3 program statements

All programming languages have statements that help you perform basic operations. These statements handle all programmed activity. This chapter looks at several of these programming statements as well as some additional operators. It begins by exploring the basic steps that a programmer takes when developing software. These activities are the basis of high-quality software development and a disciplined development process. Finally, we use some of the statements we have learned to produce graphical output.

chapter objectives

- Discuss basic program development steps.
- Define the flow of control through a program.
- Learn to use `if` statements.
- Define expressions that let us make complex decisions.
- Learn to use `while` and `for` statements.
- Use conditionals and loops to draw graphics.

3.0 program development

Creating software involves much more than just writing code. As you learn about programming language statements you should develop good programming habits. This section introduces some of the basic programming steps in developing software.

Software development involves four basic *development activities*:

- establishing the requirements
- creating a design
- implementing the code
- testing the implementation

It would be nice if these activities always happened in this order, but they almost never do. Instead, they often overlap. Let's discuss each development stage briefly.

key concept: Software requirements specify *what* a program must accomplish.

Software requirements are the things that a program must accomplish. They are the tasks that a program should do, not how it should do them. You may recall from Chapter 1 that programming is really about solving a problem. Requirements are the clear expression of that problem. Until we know what problem we are trying to solve, we can't solve it.

The person or group who wants a software product developed (the *client*) will usually give you a set of requirements. However, these requirements are often incomplete, ambiguous, or even contradictory. You must work with the client until you both agree on what the system will do.

Requirements often have to do with user interfaces such as output format, screen layouts, and graphics. These are the things that make the program useful for the end user. Requirements may also apply constraints to your program, such as how fast a task must be performed. They may also impose restrictions such as deadlines.

A *software design* describes *how* a program will meet the requirements. The design spells out the classes and objects needed in a program and how they work with each other. A detailed design might even list the steps that parts of the code will follow.

key concept: A software design spells out *how* a program will accomplish its requirements.

A civil engineer would never consider building a bridge without designing it first. The design of software is just as important. Many software problems are the result of poor or sloppy design. You need to consider all the different ways of meeting the requirements, not jump on the first idea. Often, the first attempt at a design is not the best solution. Luckily, changes are easy to make during the design stage.

One basic design issue is defining the *algorithms* to be used in the program. An algorithm is a step-by-step process for solving a problem. A recipe is like an algorithm. Travel directions are like an algorithm. Every program uses one or more algorithms. Every software developer should spend time thinking about the algorithms before writing any code.

An algorithm is often written in *pseudocode,* which is a mixture of code statements and English phrases sort of like a rough draft of an essay. Pseudocode helps you decide how the code will operate without getting bogged down in the details of a particular programming language.

key concept
An algorithm is a step-by-step process for solving a problem, often expressed in pseudocode.

When you develop an algorithm, you should study all of the requirements involved with that part of the problem. This ensures that the algorithm takes into account all aspects of the problem. You should be willing to revise many times before you're done.

Implementation is the process of writing the source code, in a particular programming language. Too many programmers focus on implementation, when actually it should be the least creative part of development. The important decisions should be made when the requirements are established and the design is created. During implementation it is often best to start at the highest, or top, level and work your way down to the smaller and more detailed pieces. This is called *top-down* development. For example, you may implement the main algorithm of your program first and then move on to the smaller pieces that the main algorithm calls on.

Testing a program includes running it many times with different inputs and carefully studying the results. Testing might also include hand-tracing program code, in which you mentally play the role of the computer to see where the program logic might fail.

key concept
Implementation should be the least creative of all development activities.

The goal of testing is to find errors. By finding errors and fixing them, we improve the quality of our program. It's likely that later on someone else will find errors that remained hidden during development, when the cost of fixing that error is much higher. Taking the time to uncover problems as early as possible is always worth the effort.

Running a program and getting the correct results only means that the program works for the data you put in. The more times you test, with different input, the more confident you will feel. But you can never really be sure that you've caught all the errors. There could always be an error you didn't find. Because of that, it is important to thoroughly test a program with many different kinds of input. When one problem is fixed, you should run your tests over again to make sure that when you fixed the problem you didn't create a new problem. This technique is called *regression testing.*

key concept
The goal of testing is to find errors. We can never really be sure that all errors have been found.

3.1 control flow

The order in which statements are executed is called the *flow of control.* Most of the time, a running program starts at the first programming statement and moves down one statement at a time until the program is complete. A Java application begins with the first line of the `main` method and proceeds step by step until it gets to the end of the `main` method.

Invoking a method changes the flow of control. When a method is called, control jumps to the code for that method. When the method finishes, control returns to the place where the method was called and processing continues from there. In our examples so far, we've invoked methods in classes and objects using the Java libraries, and we haven't been concerned about the code that defines those methods. We discuss how to write our own classes and methods in Chapter 4.

key concept

Conditionals and loops let us control the flow of execution through a method.

Within a given method, we can changes the flow of control through the code by using certain types of programming statements. Statements that control the flow of execution through a method fall into two categories: conditionals and loops.

A *conditional statement* is sometimes called a *selection statement* because it lets us choose which statement will be executed next. The conditional statements in Java that we will study are the `if` statement and the `if-else` statement. These statements let us decide which statement to execute next. Each decision is based on a *boolean expression* (also called a *condition*), which says whether something is true or false. The result of the expression determines which statement is executed next.

For example, the cost of life insurance might depend on whether the insured person is a smoker. If the person smokes, we calculate the cost using one particular formula; if not, we calculate it using another. The role of a conditional statement is to evaluate a boolean condition (whether the person smokes) and then to execute the proper calculation accordingly.

A *loop,* or *repetition statement,* lets us execute the same statement over and over again. Like a conditional, a loop is based on a boolean expression that determines how many times the statement is executed.

For example, suppose we wanted to calculate the grade point average of every student in a class. The calculation is the same for each student; it is just performed on different data. We would set up a loop that repeats the calculation for each student until there are no more students to process.

Java has three types of loop statements:

- the `while` statement
- the `do` statement
- the `for` statement

Each type of loop statement has unique characteristics. We will study the `while` and `for` statements in this book. Information on the `do` statement (which is not in the AP* subset) can be found on the Web site.

Conditionals and loops control the flow through a method and are needed in many situations. This chapter explores conditional and loop statements as well as some additional operators.

3.2 the if statement

The *if statement* is a conditional statement found in many programming languages, including Java. The following is an example of an `if` statement:

```
if (total > amount)
   total = total + (amount + 1);
```

key concept

An `if` statement lets a program choose whether to execute a particular statement.

An `if` statement consists of the reserved word `if` followed by a boolean expression, or condition, followed by a statement. The condition is enclosed in parentheses and must be either true or false. If the condition is true, the statement is executed and processing continues with the next statement. If the condition is false, the statement is skipped and processing continues immediately with the next statement. In this example, if the value in `total` is greater than the value in `amount`, the assignment statement is executed; otherwise, the assignment statement is skipped. Figure 3.1 shows how this works.

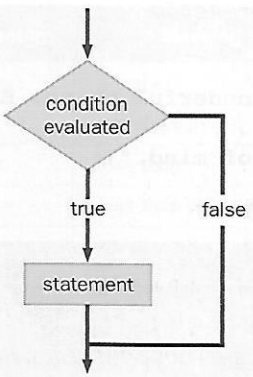


figure 3.1 The logic of an `if` statement

Note that the assignment statement in this example is indented under the header line of the `if` statement. This tells us that the assignment statement is part of the `if` statement; it means that the `if` statement controls whether the assignment statement will be executed. This indentation is extremely important for the people who read the code.

key concept Indentation is important for human readability. It shows the relationship between one statement and another.

The example in Listing 3.1 reads the age of the user and then decides which sentence to print, based on the age that is entered.

listing 3.1

```
//********************************************************************
//  Age.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of an if statement.
//********************************************************************

import java.util.Scanner;

public class Age
{
   //-----------------------------------------------------------------
   //  Reads the user's age and prints comments accordingly.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int MINOR = 21;
      Scanner scan = new Scanner (System.in);

      System.out.print ("Enter your age: ");
      int age = scan.nextInt();

      System.out.println ("You entered: " + age);

      if (age < MINOR)
         System.out.println ("Youth is a wonderful thing. Enjoy.");

      System.out.println ("Age is a state of mind.");
   }
}
```

output

```
Enter your age: 35
You entered: 35
Age is a state of mind.
```

The Age program in Listing 3.1 echoes (reads back) the age value that is entered in all cases. If the age is less than the value of the constant MINOR, the statement about youth is printed. If the age is equal to or greater than the value of MINOR, the println statement is skipped. In either case, the final sentence about age being a state of mind is printed.

equality and relational operators

Boolean expressions evaluate to either true or false. Java has several operators that produce a true or false result. The == and != operators are called *equality operators;* they test if two values are equal (==) or not equal (!=). Note that the equality operator is two equal signs side by side and should not be mistaken for the assignment operator that uses only one equal sign.

The following if statement prints a sentence only if the variables total and sum contain the same value:

```
if (total == sum)
   System.out.println ("total equals sum");
```

Likewise, the following if statement prints a sentence only if the variables total and sum do *not* contain the same value:

```
if (total != sum)
   System.out.println ("total does NOT equal sum");
```

In the Age program in Listing 3.1 we used the < operator to decide whether one value was less than another. The less than operator is one of several *relational operators* that let us decide the relationships between values. Figure 3.2 lists the Java equality and relational operators.

Operator	Meaning
==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

figure 3.2 Java equality and relational operators

The equality and relational operators have precedence lower than the arithmetic operators. This means that arithmetic operations are evaluated first, followed by equality and relational operations. As always, parentheses can be used to specify the order of evaluation.

Let's look at a few more examples of basic `if` statements.

```
if (size >= MAX)
   size = 0;
```

This `if` statement causes the variable `size` to be set to zero if its current value is greater than or equal to the value in the constant `MAX`.

The condition of the following `if` statement first adds three values together, then compares the result to the value stored in `numBooks`.

```
if (numBooks < stackCount + inventoryCount + duplicateCount)
   reorder = true;
```

If `numBooks` is less than the other three values combined, the boolean variable `reorder` is set to `true`. The addition operations are performed before the less than operator because the arithmetic operators have a higher precedence than the relational operators.

The following `if` statement compares the value returned from a call to `nextInt` to the calculated result of dividing the constant `HIGH` by 5. The odds of this code picking a winner are 1 in 5.

```
if (generator.nextInt(HIGH) < HIGH / 5)
   System.out.println ("You are a randomly selected winner!");
```

the if-else statement

Sometimes we want to do one thing if a condition is true and another thing if that condition is false. We can add an *else clause* to an `if` statement, making it an *if-else statement*, to handle this kind of situation. The following is an example of an `if-else` statement:

```
if (height <= MAX)
   adjustment = 0;
else
   adjustment = MAX - height;
```

If the condition is true, the first assignment statement is executed; if the condition is false, the second statement is executed. Only one or the other will be executed because a boolean condition will evaluate to either true or false.

Note that we indented to show that the statements are part of the `if` statement.

key concept

An `if-else` statement tells a program to do one thing if a condition is true and another thing if the condition is false.

The `Wages` program shown in Listing 3.2 uses an `if-else` statement to compute the payment for an employee.

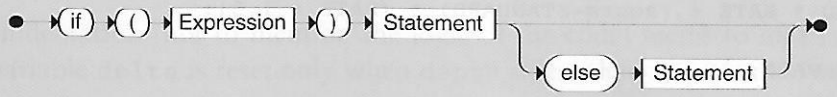
In the `Wages` program, if an employee works over 40 hours in a week, the payment amount includes the overtime hours. An `if-else` statement is used to determine whether the number of hours entered by the user is greater than 40. If it is, the extra hours are paid at a rate one and a half times the normal rate. If there are no overtime hours, the total payment is based simply on the number of hours worked and the standard rate.

Let's look at another example of an `if-else` statement:

```
if (roster.getSize() == FULL)
   roster.expand();
else
   roster.addName (name);
```

This example uses an object called `roster`. Even without knowing what `roster` is, we can see that it has at least three methods: `getSize`, `expand`, and `addName`. The condition of the `if` statement calls `getSize` and compares the result to the constant `FULL`. If the condition is true, the `expand` method is invoked (apparently to expand the size of the roster). If the roster is not yet full, the variable `name` is passed as a parameter to the `addName` method.

If Statement



An `if` statement tests the boolean Expression. If it is true, the program executes the first Statement. The optional `else` clause shows the Statement that should be executed if the Expression is false.

Examples:

```
if (total < 7)
   System.out.println ("Total is less than 7.");

if (firstCh != 'a')
   count++;
else
   count = count / 2;
```

listing 3.2

```
//********************************************************************
//  Wages.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of an if-else statement.
//********************************************************************

import java.text.NumberFormat;
import java.util.Scanner;

public class Wages
{
   //-----------------------------------------------------------------
   //  Reads the number of hours worked and calculates wages.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final double RATE = 8.25;  // regular pay rate
      final int STANDARD = 40;   // standard hours in a work week

      double pay = 0.0;
      Scanner scan = new Scanner (System.in);

      System.out.print ("Enter the number of hours worked: ");
      int hours = scan.nextInt();

      System.out.println ();

      // Pay overtime at "time and a half"
      if (hours > STANDARD)
         pay = STANDARD * RATE + (hours-STANDARD) * (RATE * 1.5);
      else
         pay = hours * RATE;

      NumberFormat fmt = NumberFormat.getCurrencyInstance();
      System.out.println ("Gross earnings: " + fmt.format(pay));
   }
}
```

output

```
Enter the number of hours worked: 46

Gross earnings: $404.25
```

using block statements

We may want to do more than one thing as the result of evaluating a boolean condition. In Java, we can replace any single statement with a *block statement*. A block statement is a collection of statements enclosed in braces. We've already seen these braces used with the `main` method and a class definition. The program called `Guessing`, shown in Listing 3.3, uses an `if-else` statement with the statement of the `else` clause in a block statement.

If the user's guess equals the answer, the sentences "You got it! Good guessing" are printed. If the guess doesn't match two statements are printed, one that says that the guess is wrong and one that prints the actual answer. A programming project at the end of this chapter expands this into the Hi-Lo game.

Note that if we didn't use the block braces, the sentence stating that the guess is incorrect would be printed if the guess was wrong, but the sentence revealing the correct answer would be printed in all cases. That is, only the first statement would be considered part of the `else` clause.

Remember that indentation is only for people reading the code. Statements that are not blocked properly can cause the programmer to misunderstand how the code will execute. For example, the following code is misleading:

```
if (depth > 36.238)
   delta = 100;
else
   System.out.println ("WARNING: Delta is being reset to ZERO");
   delta = 0;  // not part of the else clause!
```

The indentation (not to mention the logic of the code) seems to mean that the variable `delta` is reset only when `depth` is less than `36.238`. However, without using a block, the assignment statement that resets `delta` to zero is not governed by the `if-else` statement at all. It is executed in either case, which is clearly not what is intended.

listing 3.3

```
//********************************************************************
//  Guessing.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of a block statement in an if-else.
//********************************************************************

import java.util.Scanner;
import java.util.Random;

public class Guessing
{
   //-----------------------------------------------------------------
   //  Plays a simple guessing game with the user.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int MAX = 10;
      int answer, guess;
      Scanner scan = new Scanner (System.in);

      Random generator = new Random();
      answer = generator.nextInt(MAX) + 1;

      System.out.print ("I'm thinking of a number between 1 and "
                        + MAX + ". Guess what it is: ");
      guess = scan.nextInt();

      if (guess == answer)
         System.out.println ("You got it! Good guessing!");
      else
      {
         System.out.println ("That is not correct, sorry.");
         System.out.println ("The number was " + answer);
      }
   }
}
```

output

```
I'm thinking of a number between 1 and 10. Guess what it is: 7
That is not correct, sorry.
The number was 4
```

A block statement can be used anywhere a single statement is called for in Java syntax. For example, the `if` part of an `if-else` statement could be a block, or the `else` portion could be a block (as we saw in the `Guessing` program), or both parts could be block statements. For example:

```
if (boxes != warehouse.getCount())
{
   System.out.println ("Inventory and warehouse do NOT match.");
   System.out.println ("Beginning inventory process again!");
   boxes = 0;
}
else
{
   System.out.println ("Inventory and warehouse MATCH.");
   warehouse.ship();
}
```

In this `if-else` statement, the value of `boxes` is compared to a value that we got by calling the `getCount` method of the `warehouse` object (whatever that is). If they do not match exactly, two `println` statements and an assignment statement are executed. If they do match, a different message is printed and the `ship` method of `warehouse` is invoked.

nested if statements

The statement executed as the result of an `if` statement could be another `if` statement. This situation is called a *nested if*. It lets us make another decision after getting the results of a previous decision. The program in Listing 3.4, called `MinOfThree`, uses nested `if` statements to find the smallest of three integer values entered by the user.

Carefully trace the logic of the `MinOfThree` program, using different sets of numbers, with the smallest number in a different position each time, to see how the program chooses the lowest value.

An important situation arises with nested `if` statements. It may seem that an `else` clause after a nested `if` could apply to either `if` statement. For example:

```
if (code == 'R')
   if (height <= 20)
      System.out.println ("Situation Normal");
   else
      System.out.println ("Bravo!");
```

listing 3.4

```
//********************************************************************
//  MinOfThree.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of nested if statements.
//********************************************************************

import java.util.Scanner;

public class MinOfThree
{
   //-----------------------------------------------------------------
   //  Reads three integers from the user and determines the smallest
   //  value.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int num1, num2, num3, min = 0;
      Scanner scan = new Scanner (System.in);

      System.out.println ("Enter three integers: ");
      num1 = scan.nextInt();
      num2 = scan.nextInt();
      num3 = scan.nextInt();

      if (num1 < num2)
         if (num1 < num3)
            min = num1;
         else
            min = num3;
      else
         if (num2 < num3)
            min = num2;
         else
            min = num3;

      System.out.println ("Minimum value: " + min);
   }
}
```

output

```
Enter three integers:
45   22   69
Minimum value: 22
```

Is the `else` clause matched to the inner `if` statement or the outer `if` statement? The indentation in this example seems to mean that it is part of the inner `if` statement, and that is correct. An `else` clause is always matched to the closest unmatched `if` that came before it. However, if we're not careful, we can easily mismatch it in our mind and imply our intentions, but not reality, by misaligned indentation. This is another reason why accurate, consistent indentation is so important.

Braces can be used to show which `if` statement belongs with which `else` clause. For example, if our example had been written so that the string `"Bravo!"` is printed if `code` is not equal to `'R'`, we could force that relationship (and properly indent) as follows:

key concept

In a nested `if` statement, an `else` clause is matched to the closest unmatched `if`.

```
if (code == 'R')
{
   if (height <= 20)
      System.out.println ("Situation Normal");
}
else
   System.out.println ("Bravo!");
```

By using the block statement in the first `if` statement, we establish that the `else` clause belongs to it.

3.3 boolean expressions revisited

Let's look at a few more uses of boolean expressions.

logical operators

In addition to the equality and relational operators, Java has three *logical operators* that produce boolean results. They also take boolean operands. Figure 3.3 lists and describes the logical operators.

Operator	Description	Example	Result
!	logical NOT	! a	true if a is false and false if a is true
&&	logical AND	a && b	true if a and b are both true and false otherwise
||	logical OR	a || b	true if a or b or both are true and false otherwise

figure 3.3 Java logical operators

The ! operator is used to perform the *logical NOT* operation, which is also called the *logical complement*. The logical complement of a boolean value gives its opposite value. That is, if a boolean variable called found has the value false, then !found is true. Likewise, if found is true, then !found is false. The logical NOT operation does not change the value stored in found.

A logical operation can be described by a *truth table* that lists all the combinations of values for the variables involved in an expression. Because the logical NOT operator is unary, there are only two possible values for its one operand, true or false. Figure 3.4 shows a truth table that describes the ! operator.

The && operator performs a *logical AND* operation. The result is true if both operands are true, but false otherwise. Since it is a binary operator and each operand has two possible values, there are four combinations to consider.

The result of the *logical OR* operator (||) is true if one or the other or both operands are true, but false otherwise. It is also a binary operator. Figure 3.5 is a truth table that shows both the && and || operators.

The logical NOT has the highest precedence of the three logical operators, followed by logical AND, then logical OR.

Logical operators are often used as part of a condition for a selection or repetition statement. For example, consider the following if statement:

```
if (!done && (count > MAX))
   System.out.println ("Completed.");
```

Under what conditions would the println statement be executed? The value of the boolean variable done is either true or false, and the NOT operator reverses that value. The value of count is either greater than MAX or it isn't. The truth table in Figure 3.6 shows all of the possibilities.

a	!a
false	true
true	false

figure 3.4 Truth table describing the logical NOT operator

a	b	a && b	a || b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

figure 3.5 Truth table describing the logical AND and OR operators

An important characteristic of the `&&` and `||` operators is that they are "short-circuited." That is, if their left operand is enough to decide the boolean result of the operation, the right operand is not evaluated. This situation can occur with both operators but for different reasons. If the left operand of the `&&` operator is false, then the result of the operation will be false no matter what the value of the right operand is. Likewise, if the left operand of the `||` is true, then the result of the operation is true no matter what the value of the right operand is.

key concept

Logical operators return a boolean value and are often used to build sophisticated conditions.

This can be very useful. For example, the condition in the following `if` statement will not try to divide by zero if the left operand is false. If `count` has the value zero, the left side of the `&&` operation is false; so the whole expression is false and the right side is not evaluated.

```
if (count != 0 && total/count > MAX)
   System.out.println ("Testing.");
```

Be careful when you use these programming language characteristics. Not all programming languages work the same way. As we have mentioned several times, you should always make it clear to the reader exactly how the logic of your program works.

done	count > MAX	!done	!done && (count > MAX)
false	false	true	false
false	true	true	true
true	false	false	false
true	true	false	false

figure 3.6 A truth table for a specific condition

comparing characters and strings

We know what it means when we say that one *number* is less than another, but what does it mean to say one *character* is less than another? As we discussed in Chapter 2, characters in Java are based on the Unicode character set, which orders all possible characters that can be used. Because the character `'a'` comes before the character `'b'` in the character set, we can say that `'a'` is less than `'b'`.

We can use the equality and relational operators on character data. For example, if two character variables, `ch1` and `ch2`, hold the values of two characters, we might determine their order in the Unicode character set with an `if` statement as follows:

```
if (ch1 > ch2)
   System.out.println (ch1 + " is greater than " + ch2);
else
   System.out.println (ch1 + " is NOT greater than " + ch2);
```

key concept

The order of characters in Java is defined by the Unicode character set.

In the Unicode character set all lowercase alphabetic characters (`'a'` through `'z'`) are in alphabetical order. The same is true of uppercase alphabetic characters (`'A'` through `'Z'`) and digits (`'0'` through `'9'`). The digits come before the uppercase alphabetic characters, which come before the lowercase alphabetic characters. Before, after, and in between these groups are other characters. (See the chart in Appendix B.)

This makes it easy to sort characters and strings of characters. If you have a list of names, for instance, you can put them in alphabetical order based on the relationships in the character set.

However, you should not use the equality or relational operators to compare `String` objects. The `String` class has a method called `equals` that returns a `boolean` value that is true if the two strings contain exactly the same characters, and false if they do not. For example:

```
if (name1.equals(name2))
   System.out.println ("The names are the same.");
else
   System.out.println ("The names are not the same.");
```

Assuming that `name1` and `name2` are `String` objects, this condition determines whether the characters they contain are exactly the same. Because both objects were created from the `String` class, they both respond to the `equals` message. Therefore we could have written the condition as `name2.equals(name1)` and gotten the same result.

We could test the condition `(name1 == name2)`, but that actually tests to see whether both reference variables refer to the same `String` object.

That is, the == operator tests whether both reference variables contain the same address. That's different than testing to see whether two different `String` objects contain the same characters. We discuss this in more detail later in the book.

To determine the relative ordering of two strings, use the `compareTo` method of the `String` class. The `compareTo` method is more flexible than the `equals` method. Instead of returning a `boolean` value, the `compareTo` method returns a number. The return value is negative if the first `String` object (`name1`) is less than the second string (`name2`). The return value is zero if the two strings contain the same characters. The return value is positive if the first `String` object is greater than the second string. For example:

```
int result = name1.compareTo(name2);
if (result < 0)
   System.out.println (name1 + " comes before " + name2);
else
   if (result == 0)
      System.out.println ("The names are equal.");
   else
      System.out.println (name1 + " follows " + name2);
```

Keep in mind that comparing characters and strings is based on the Unicode character set (see Appendix B). This is called a *lexicographic ordering*. If all alphabetic characters are in the same case (upper or lower), the lexicographic ordering will be alphabetic. However, when comparing two strings, such as `"able"` and `"Baker"`, the `compareTo` method will conclude that `"Baker"` comes first because all of the uppercase letters come before all of the lowercase letters in the Unicode character set. A string that is the prefix of another, longer string is considered to precede the longer string. For example, when comparing two strings such as `"horse"` and `"horsefly"`, the `compareTo` method will conclude that `"horse"` comes first.

key concept

The `compareTo` method determines lexicographic order, which does not correspond exactly to alphabetical order.

comparing floating point values

Another interesting situation occurs when floating point data is compared. Specifically, you should rarely use the equality operator (==) when comparing floating point values. Two floating point values are equal, according to the == operator, only if all the binary digits of their underlying representations match. If the compared values are the results of computation, they may not be exactly equal. For example, 5.349 is not equal to 5.3490001.

A better way to check for floating point equality is to get the absolute value of the difference between the two values and compare the result to some tolerance level. For example, we may choose a tolerance level of `0.00001`. If the two floating point values are so close that their difference is less than the tolerance, then we may consider them equal. For example, two floating point values, `f1` and `f2`, could be compared as follows:

```
if (Math.abs(f1 - f2) < TOLERANCE)
   System.out.println ("Essentially equal.");
```

The value of the constant `TOLERANCE` should be appropriate for the situation.

3.4 more operators

Let's look at a few more Java operators to give us even more ways to express our program commands. Some of these operators are commonly used in loop processing.

increment and decrement operators

The *increment operator* (++) adds 1 to any integer or floating point value. The two plus signs cannot be separated by white space. The *decrement operator* (--) is similar except that it subtracts 1 from the value. The increment and decrement operators are both unary operators because they operate on only one operand. The following statement causes the value of `count` to be increased by one, or *incremented*.

```
count++;
```

The result is stored back in the variable `count`. Therefore this statement is the same as the following statement:

```
count = count + 1;
```

assignment operators

Several *assignment operators* in Java combine a basic operation with assignment. For example, the += operator can be used as follows:

```
total += 5;
```

This does the thing as the following statement:

```
total = total + 5;
```

The right-hand side of the assignment operator can be a full expression. The expression on the right-hand side of the operator is evaluated, then that result is added to the current value of the variable on the left-hand side, and that value is stored in the variable. So the following statement:

```
total += (sum - 12) / count;
```

is the same as:

```
total = total + ((sum - 12) / count);
```

Many similar Java assignment operators are listed in Figure 3.7.

All of the assignment operators evaluate the expression on the right-hand side first, then use the result as the right operand of the other operation. So the following statement:

```
result *= count1 + count2;
```

is the same as:

```
result = result * (count1 + count2);
```

Likewise, the following statement:

```
result %= (highest - 40) / 2;
```

is the same as:

```
result = result % ((highest - 40) / 2);
```

Operator	Description	Example	Equivalent Expression
=	assignment	x = y	x = y
+=	addition, then assignment	x += y	x = x + y
+=	string concatenation, then assignment	x += y	x = x + y
-=	subtraction, then assignment	x -= y	x = x - y
*=	multiplication, then assignment	x *= y	x = x * y
/=	division, then assignment	x /= y	x = x / y
%=	remainder, then assignment	x %= y	x = x % y

figure 3.7 Java assignment operators

Some assignment operators have special functions depending on the types of the operands, just as regular operators do. For example, if the operands to the += operator are strings, then the assignment operator performs string concatenation.

3.5 the while statement

As we discussed earlier in this chapter, a repetition statement (or loop) lets us execute a statement as many times as we need to. A `while` *statement* is a loop that evaluates a boolean condition—just like an `if` statement does—and executes a statement (called the *body* of the loop) if the condition is true. However, unlike the `if` statement, after the body is executed, the condition is evaluated again. If it is still true, the body is executed again. This repeats until the condition becomes false; then processing continues with the statement after the body of the `while` loop. Figure 3.8 shows this processing.

key concept
A `while` statement lets a program execute the same statement many times.

The `Counter` program shown in Listing 3.5 simply prints the values from 1 to 5. Each turn through the loop prints one value, then increases the counter by one. A constant called `LIMIT` holds the maximum value that `count` is allowed to reach. The condition of the `while` loop, (`count <= LIMIT`), means that the loop will keep going as long as `count` is less than or equal to `LIMIT`. Once `count` reaches the limit, the condition is false an the loop quits.

Note that the body of the `while` loop is a block containing two statements. Because the value of `count` is increased by one each time, we are guaranteed that `count` will eventually reach the value of `LIMIT`.

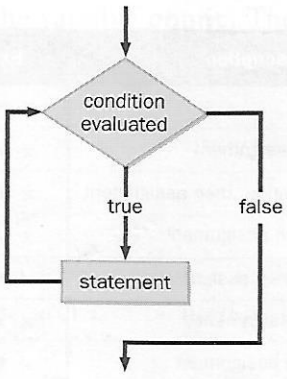


figure 3.8 The logic of a `while` loop

listing 3.5

```
//********************************************************************
//  Counter.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of a while loop.
//********************************************************************

public class Counter
{
   //-----------------------------------------------------------------
   //  Prints integer values from 1 to a specific limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int LIMIT = 5;
      int count = 1;

      while (count <= LIMIT)
      {
         System.out.println (count);
         count = count + 1;
      }

      System.out.println ("Done");
   }
}
```

output

```
1
2
3
4
5
Done
```

Let's look at another program that uses a `while` loop. The `Average` program shown in Listing 3.6 reads integer values from the user, adds them up, and computes their average.

We don't know how many values the user may enter, so we need to have a way to show that the user is done. In this program, we pick zero to be a *sentinel value*, which is a value that shows the end of the input the way a sentinel stands guard at the gate of a fort or perimeter of an army's camp. The `while` loop continues to process input values until the user enters zero. This assumes that zero is not one of the valid numbers that should contribute to

listing 3.6

```
//********************************************************************
//  Average.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of a while loop, a sentinel value, and a
//  running sum.
//********************************************************************

import java.text.DecimalFormat;
import java.util.Scanner;

public class Average
{
   //-----------------------------------------------------------------
   //  Computes the average of a set of values entered by the user.
   //  The running sum is printed as the numbers are entered.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int sum = 0, value, count = 0;
      double average;
      Scanner scan = new Scanner (System.in);

      System.out.print ("Enter an integer (0 to quit): ");
      value = scan.nextInt();

      while (value != 0)  // sentinel value of 0 to terminate loop
      {
         count++;

         sum += value;
         System.out.println ("The sum so far is " + sum);

         System.out.print ("Enter an integer (0 to quit): ");
         value = scan.nextInt();
      }

      System.out.println ();
      System.out.println ("Number of values entered: " + count);

      average = (double)sum / count;

      DecimalFormat fmt = new DecimalFormat ("0.###");
```

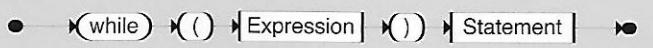
Listing 3.6 continued

```
      System.out.println ("The average is " + fmt.format(average));
   }
}
```

output

```
Enter an integer (0 to quit): 25
The sum so far is 25
Enter an integer (0 to quit): 164
The sum so far is 189
Enter an integer (0 to quit): -14
The sum so far is 175
Enter an integer (0 to quit): 84
The sum so far is 259
Enter an integer (0 to quit): 12
The sum so far is 271
Enter an integer (0 to quit): -35
The sum so far is 236
Enter an integer (0 to quit): 0
Number of values entered: 6
The average is 39.333
```

While Statement



The `while` loop executes the Statement over and over as long as the boolean Expression is true. The Expression is evaluated first; so the Statement might not be executed at all. The Expression is evaluated again after each execution of the Statement until the Expression becomes false.

Example:

```
while (total > max)
{
   total = total / 2;
   System.out.println ("Current total: " + total);
}
```

the average. A sentinel value must always be outside the normal range of values entered.

Note that in the `Average` program in Listing 3.6, a variable called `sum` is used to keep a *running sum,* which means it is the total of the values entered so far. The variable `sum` starts at zero, and each value read is added to and stored back into `sum`.

We also have to count the number of values that are entered so that after the loop finishes we can divide by the right number to get the average. Note that the sentinel value is not counted. But what if the user immediately enters the sentinel value before entering any valid values? The value of `count` in this case will still be zero and the computation of the average will result in a runtime error. Fixing this problem is left as a programming project.

Let's look at another program that uses a `while` loop. The `WinPercentage` program shown in Listing 3.7 computes the winning percentage of a sports team based on the number of games won.

We use a `while` loop in the `WinPercentage` program to *validate the input,* meaning we guarantee that the user enters a value that we consider to be valid. In this example, that means that the number of games won must be greater than or equal to zero and less than or equal to the total number of games played. The `while` loop keeps executing, repeatedly asking the user for valid input, until the entered number is indeed valid.

Validating input data, avoiding errors such as dividing by zero, and performing other actions that guarantee proper processing are important design steps. We generally want our programs to be *robust,* which means that they handle errors—even user errors—well.

infinite loops

The programmer must make sure that the condition of a loop will eventually become false. If it doesn't, the loop will keep going forever, or at least until the program is interrupted. This situation, called an *infinite loop,* is a common mistake.

key concept

We must design our programs carefully to avoid infinite loops. The loop condition must eventually become false.

The program shown in Listing 3.8 has an infinite loop. If you execute this program, you will have to interrupt it to make it stop. On most systems, pressing the Control-C keyboard combination (hold down the Control key and press C) stops a running program.

In the `Forever` program in Listing 3.8, the starting value of `count` is 1 and it is subtracted from, or decremented, in the loop body. The `while` loop will continue as long as `count` is less than or equal to 25. Because `count` gets smaller with each iteration, the condition will always be true.

listing 3.7

```
//********************************************************************
//  WinPercentage.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of a while loop for input validation.
//********************************************************************

import java.text.NumberFormat;
import java.util.Scanner;

public class WinPercentage
{
   //-----------------------------------------------------------------
   //  Computes the percentage of games won by a team.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int NUM_GAMES = 12;
      int won;
      double ratio;
      Scanner scan = new Scanner (System.in);

      System.out.print ("Enter the number of games won (0 to "
                        + NUM_GAMES + "): ");
      won = scan.nextInt();

      while (won < 0 || won > NUM_GAMES)
      {
         System.out.print ("Invalid input. Please reenter: ");
         won = scan.nextInt();
      }

      ratio = (double)won / NUM_GAMES;

      NumberFormat fmt = NumberFormat.getPercentInstance();

      System.out.println ();
      System.out.println ("Winning percentage: " + fmt.format(ratio));
   }
}
```

output

```
Enter the number of games won (0 to 12): -5
Invalid input. Please reenter: 13
Invalid input. Please reenter: 7

Winning percentage: 58%
```

listing 3.8

```
//********************************************************************
//  Forever.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates an INFINITE LOOP.  WARNING!!
//********************************************************************

public class Forever
{
   //-----------------------------------------------------------------
   //  Prints ever-decreasing integers in an INFINITE LOOP!
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int count = 1;

      while (count <= 25)
      {
         System.out.println (count);
         count = count - 1;
      }

      System.out.println ("Done");  // this statement is never reached
   }
}
```

output

```
1
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
and so on until interrupted
```

Let's look at some other examples of infinite loops:

```
int count = 1;
while (count != 50)
   count += 2;
```

In this code fragment, the variable count begins at 1 and moves in a positive direction. However, note that it is increased by 2 each time. This loop will never terminate because count will never equal 50. It begins at 1 and then changes to 3, then 5, and so on. Eventually it reaches 49, then changes to 51, then 53, and continues forever.

Now consider the following situation:

```
double num = 1.0;
while (num != 0.0)
   num = num - 0.1;
```

Once again, the value of the loop control variable seems to be moving in the right direction. And, in fact, it seems like num will eventually take on the value 0.0. However, this loop is infinite (at least on most systems) because num will never have a value *exactly* equal to 0.0. This situation is like the one we discussed earlier in this chapter when we compared floating point values in the condition of an if statement. Because of the way the values are represented in binary, tiny differences make comparing floating point values (for equality) a problem.

nested loops

The body of a loop can contain another loop. This situation is called a *nested loop*. Keep in mind that each time the outer loop executes once, the inner loop executes completely. Consider the following code fragment. How many times does the string "Here again" get printed?

```
int count1, count2;
count1 = 1;
while (count1 <= 10)
{
   count2 = 1;
   while (count2 <= 50)
   {
      System.out.println ("Here again");
      count2++;
   }
   count1++;
}
```

The println statement is inside the inner loop. The outer loop executes 10 times, as count1 iterates between 1 and 10. The inner loop executes 50 times, as count2 iterates between 1 and 50. Each time the outer loop executes, the inner loop executes completely. So the println statement is executed 500 times.

As with any loop, we must study the conditions of the loops and the initializations of variables. Let's consider some small changes to this code. What if the condition of the outer loop were `(count1 < 10)` instead of `(count1 <= 10)`? How would that change the total number of lines printed? Well, the outer loop would execute 9 times instead of 10, so the `println` statement would be executed 450 times. What if the outer loop were left as it was originally defined, but `count2` were initialized to 10 instead of 1 before the inner loop? The inner loop would then execute 40 times instead of 50, so the total number of lines printed would be 400.

Let's look at another example of a nested loop. A *palindrome* is a string of characters that reads the same forward or backward. For example, the following strings are palindromes:

- radar
- drab bard
- ab cde xxxx edc ba
- kayak
- deified
- able was I ere I saw elba

Note that some palindromes have an even number of characters, whereas others have an odd number of characters. The `PalindromeTester` program shown in Listing 3.9 tests to see whether a string is a palindrome. Users may test as many strings as they want.

The code for `PalindromeTester` contains two loops, one inside the other. The outer loop controls how many strings are tested, and the inner loop scans through each string, character by character, until it determines whether the string is a palindrome.

The variables `left` and `right` store the indexes of two characters. At first they indicate the characters on either end of the string. Each execution of the inner loop compares the two characters indicated by `left` and `right`. We fall out of the inner loop when either the characters don't match, meaning the string is not a palindrome, or when the value of `left` becomes equal to or greater than the value of `right`, which means the entire string has been tested and it is a palindrome.

listing 3.9

```
//********************************************************************
//  PalindromeTester.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of nested while loops.
//********************************************************************

import java.util.Scanner;

public class PalindromeTester
{
   //-----------------------------------------------------------------
   //  Tests strings to see if they are palindromes.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      String str, another = "y";
      int left, right;
      Scanner scan = new Scanner (System.in);

      while (another.equalsIgnoreCase("y")) // allows y or Y
      {
         System.out.println ("Enter a potential palindrome:");
         str = scan.nextLine();

         left = 0;
         right = str.length() - 1;

         while (str.charAt(left) == str.charAt(right) && left < right)
         {
            left++;
            right--;
         }

         System.out.println();

         if (left < right)
            System.out.println ("That string is NOT a palindrome.");
         else
            System.out.println ("That string IS a palindrome.");

         System.out.println();
         System.out.print ("Test another palindrome (y/n)? ");
         another = scan.nextLine();
      }
   }
}
```

Listing 3.9 continued

output

```
Enter a potential palindrome:
radar

That string IS a palindrome.

Test another palindrome (y/n)? y
Enter a potential palindrome:
able was I ere I saw elba

That string IS a palindrome.

Test another palindrome (y/n)? y
Enter a potential palindrome:
abcddcba

That string IS a palindrome.

Test another palindrome (y/n)? y
Enter a potential palindrome:
abracadabra

That string is NOT a palindrome.

Test another palindrome (y/n)? n
```

Note that the following phrases would not be considered palindromes by the current version of the program:

- A man, a plan, a canal, Panama.
- Dennis and Edna sinned.
- Rise to vote, sir.
- Doom an evil deed, liven a mood.
- Go hang a salami; I'm a lasagna hog.

These strings fail our rules for a palindrome because of the spaces, punctuation marks, and changes in uppercase and lowercase. However, if these were removed or ignored, these strings read the same forward and backward. Consider how the program could be changed to handle these situations. These changes are left as a programming project at the end of the chapter.

3.6 iterators

An *iterator* is an object that has methods that allow you to process a collection of items one at a time. That is, an iterator lets you step through each item and interact with it as needed. For example, your goal may be to compute the dues for each member of a club, or print the distinct parts of a URL. The key is that an iterator provides a consistent and simple mechanism for systematically processing a group of items. Since it is inherently a repetitive process, it is closely related to the idea of loops.

key concept
An iterator is an object that helps you process a group of related items.

Technically an iterator object in Java is defined using the `Iterator` interface; interfaces are discussed in Chapter 5. For now it is simply helpful to know that such objects exist and that they can make the processing of a collection of items easier.

Every iterator object has a method called `hasNext` that returns a `boolean` value indicating if there is at least one more item to process. Therefore the `hasNext` method can be used as a condition of a loop to control the processing of each item. An iterator also has a method called `next` to retrieve the next item in the collection to process.

There are several classes in the Java standard class library that define iterator objects. One of these is `Scanner`, a class we've used several times in previous examples to help us read data from the user. The `hasNext` method of the `Scanner` class returns true if there is another input token to process. And, as we've seen previously, it has a `next` method that returns the next input token as a string.

The `Scanner` class also has specific variations of the `hasNext` method, such as the `hasNextInt` and `hasNextDouble` methods, which allow you to determine if the next input token is a particular type. Likewise, as we've seen, there are variations of the `next` method, such as `nextInt` and `nextDouble`, that retrieve values of specific types.

When reading input interactively from the standard input stream, the `hasNext` method of the `Scanner` class will wait until there is input available, then return true. That is, when input is read from the keyboard it is always thought to have more data to process—it just hasn't arrived yet until

the user types it in. That's why in previous examples we've used special sentinel values to determine the end of interactive input.

However, the fact that a `Scanner` object is an iterator is particularly helpful when the scanner is being used to process input from a source that has a specific end point, such as processing the lines of a data file or processing the parts of a character string. Let's examine an example of this type of processing.

reading text files

Suppose we have an input file called `urls.inp` that contains a list of URLs that we want to process in some way. The following are the first few lines of `urls.inp`:

```
www.google.com
java.sun.com/j2se/1.5
www.linux.org/info/gnu.html
duke.csc.villanova.edu/lewis/
www.csc.villanova.edu/academics/index.jsp
```

The program shown in Listing 3.10 reads the URLs from this file and dissects them to show the various parts of the path. It uses a `Scanner` object to process the input. In fact, it uses multiple `Scanner` objects—one to read the lines of the data file, and another to process each URL string.

listing 3.10

```
//********************************************************************
//  URLDissector.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of Scanner to read file input and parse it
//  using alternative delimiters.
//********************************************************************

import java.util.Scanner;
import java.io.*;

public class URLDissector
{
   //-----------------------------------------------------------------
   //  Reads urls from a file and prints their path components.
   //-----------------------------------------------------------------
```

Listing 3.10 continued

```
   public static void main (String[] args) throws IOException
   {
      String url;
      Scanner fileScan, urlScan;

      fileScan = new Scanner (new File("urls.inp"));

      // Read and process each line of the file
      while (fileScan.hasNext())
      {
         url = fileScan.nextLine();
         System.out.println ("URL: " + url);

         urlScan = new Scanner (url);
         urlScan.useDelimiter("/");

         //  Print each part of the url
         while (urlScan.hasNext())
            System.out.println ("    " + urlScan.next());

         System.out.println();
      }
   }
}
```

output

```
URL: www.google.com
   www.google.com

URL: java.sun.com/j2se/1.5
   java.sun.com
   j2se
   1.5

URL: www.linux.org/info/gnu.html
   www.linux.org
   info
   gnu.html

URL: duke.csc.villanova.edu/lewis/
   duke.csc.villanova.edu
   lewis

URL: www.csc.villanova.edu/academics/index.jsp
   www.csc.villanova.edu
   academics
   index.jsp
```

There are two `while` loops in this program, one nested within the other. The outer loop processes each line in the file, and the inner loop processes each token in the current line.

The variable `fileScan` is created as a scanner that operates on the input file named `urls.inp`. Instead of passing `System.in` into the `Scanner` constructor, we instantiate a `File` object that represents the input file and pass it into the `Scanner` constructor. At that point, the `fileScan` object is ready to read and process input from the input file.

If for some reason there is a problem finding or opening the input file, the attempt to create a `File` object will throw an `IOException`, which is why we've added the `throws IOException` clause to the `main` method header. The details of file I/O, however, are beyond the scope of this book; exceptions will be introduced in Chapter 5.

The body of the outer `while` loop will be executed as long as the `hasNext` method of the input file scanner returns true—that is, as long as there is more input in the data file to process. Each iteration through the loop reads one line (one URL) from the input file and prints it out.

For each URL, a new `Scanner` object is set up to parse the pieces of the URL string, which is passed to the `Scanner` constructor when instantiating the `urlScan` object. The inner `while` loop prints each token of the URL on a separate line.

Recall that, by default, a `Scanner` object assumes that white space (spaces, tabs, and new lines) is used as the delimiters separating the input tokens. If the default delimiters do not suffice, as in the processing of a URL in this example, they can be changed.

In this case, we are interested in each part of the path separated by the slash (`/`) character. A call to the `useDelimiter` method of the scanner sets the delimiter to a slash prior to processing the URL string.

3.7 the for statement

key concept

A `for` statement is usually used when we know how many times a loop will be executed.

The `while` statement is good to use when you don't know how many times you want to execute the loop body. The *for statement* is a repetition statement that works well when you *do* know exactly how many times you want to execute the loop.

The `Counter2` program shown in Listing 3.11 once again prints the numbers 1 through 5, except this time we use a `for` loop to do it.

The header of a `for` loop has three parts, separated by semicolons. Before the loop begins, the first part of the header, called the *initialization*,

listing 3.11

```
//********************************************************************
//  Counter2.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of a for loop.
//********************************************************************

public class Counter2
{
   //-----------------------------------------------------------------
   //  Prints integer values from 1 to a specific limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int LIMIT = 5;

      for (int count=1; count <= LIMIT; count++)
         System.out.println (count);

      System.out.println ("Done");
   }
}
```

output

```
1
2
3
4
5
Done
```

is executed. The second part of the header is the boolean condition. If the condition is true, the body of the loop is executed, followed by the third part of the header, which is called the *increment*. Note that the initialization part is executed only once, but the increment part is executed each time. Figure 3.9 shows this processing.

A `for` loop can be a bit tricky to read until you get used to it. The execution of the code doesn't follow a "top to bottom, left to right" reading. The increment code executes after the body of the loop, even though it is in the header.

Note how the three parts of the `for` loop header match the parts of the original `Counter` program that uses a `while` loop. The initialization part of

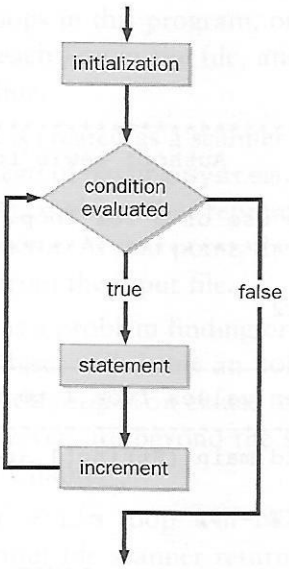


figure 3.9 The logic of a for loop

the for loop header declares the variable count as well as gives it a beginning value. We don't have to declare a variable there, but it is common practice when the variable is not needed outside the loop. Because count is declared in the for loop header, it exists only inside the loop body and can't be referenced elsewhere. The loop control variable is set up, checked, and changed by the actions in the loop header. It can be referenced inside the loop body, but it should not be changed except by the actions defined in the loop header.

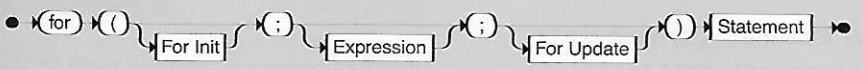
The increment part of the for loop header, in spite of its name, could decrement a value rather than increment it. For example, the following loop prints the integer values from 100 down to 1:

```
for (int num = 100; num > 0; num--)
   System.out.println (num);
```

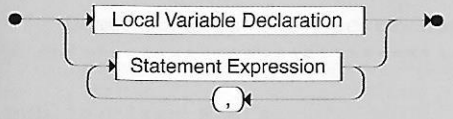
In fact, the increment part of the for loop can do any calculation, not just a simple increment or decrement. Look at the program in Listing 3.12, which prints multiples of a particular value up to a limit.

The increment part of the for loop adds the value entered by the user. The number of values printed per line is controlled by counting the values printed and then moving to the next line whenever count is evenly divisible by the PER_LINE constant.

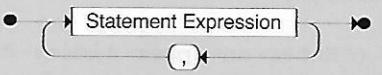
For Statement



For Init



For Update



The for statement executes the specified Statement over and over, as long as the boolean Expression is true. The For Init part of the header is executed only once, before the loop begins. The For Update part executes after each execution of Statement.

Examples:

```
for (int value=1; value < 25; value++)
   System.out.println (value + " squared is " + value*value);

for (int num=40; num > 0; num-=3)
   sum = sum + num;
```

listing 3.12

```
//********************************************************************
//  Multiples.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of a for loop.
//********************************************************************

import java.util.Scanner;

public class Multiples
{
   //-----------------------------------------------------------------
   //  Prints multiples of a user-specified number up to a user-
   //  specified limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int PER_LINE = 5;
      int value, limit, mult, count = 0;
```

listing 3.12 continued

```
      Scanner scan = new Scanner (System.in);

      System.out.print ("Enter a positive value: ");
      value = scan.nextInt();

      System.out.print ("Enter an upper limit: ");
      limit = scan.nextInt();

      System.out.println ();
      System.out.println ("The multiples of " + value + " between " +
                          value + " and " + limit + " (inclusive) are:");

      for (mult = value; mult <= limit; mult += value)
      {
         System.out.print (mult + "\t");

         // Print a specific number of values per line of output
         count++;
         if (count % PER_LINE == 0)
            System.out.println();
      }
   }
}
```

output

```
Enter a positive value: 7
Enter an upper limit: 400

The multiples of 7 between 7 and 400 (inclusive) are:
7       14      21      28      35
42      49      56      63      70
77      84      91      98      105
112     119     126     133     140
147     154     161     168     175
182     189     196     203     210
217     224     231     238     245
252     259     266     273     280
287     294     301     308     315
322     329     336     343     350
357     364     371     378     385
392     399
```

The `Stars` program in Listing 3.13 shows the use of nested `for` loops. The output is a triangle made of asterisks. The outer loop executes exactly 10 times, each time printing one line of asterisks. The inner loop has a

listing 3.13

```
//********************************************************************
//  Stars.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of nested for loops.
//********************************************************************

public class Stars
{
   //-----------------------------------------------------------------
   //  Prints a triangle shape using asterisk (star) characters.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int MAX_ROWS = 10;

      for (int row = 1; row <= MAX_ROWS; row++)
      {
         for (int star = 1; star <= row; star++)
            System.out.print ("*");

         System.out.println();
      }
   }
}
```

output

```
*
**
***
****
*****
******
*******
********
*********
**********
```

different number of iterations depending on the line value controlled by the outer loop. Each time it executes, the inner loop prints one star on the current line. Variations on this triangle program are included in the projects at the end of the chapter.

iterators and for loops

In section 3.6 we discussed that some objects are considered to be iterators, which have `hasNext` and `next` methods to process each item from a group. A variation of the `for` loop lets us process the items in an iterator without the complicated syntax.

For example, if `bookList` is an iterator object that manages `Book` objects, we can use a `for` loop to process each `Book` object in the iterator as follows:

```
for (Book myBook : bookList)
   System.out.println (myBook);
```

This version of the `for` loop is referred to as a *foreach statement.* It processes each object in the iterator in turn. It is equivalent to the following:

```
Book myBook;
while (bookList.hasNext())
{
   myBook = bookList.next();
   System.out.println (myBook);
}
```

This version of the `for` loop can also be used on arrays, which are discussed in Chapter 6, and on enumerated types. The `IceCreamShop` program in Listing 3.14 shows the use of a `foreach` loop to print all the possible values of an enumerated type. As we mentioned in Chapter 2, enumerated types are actually a special kind of class. The `values` method is used to get the list of all possible values of the enumerated type (`Flavor` in this case), which the `foreach` loop then iterates through.

comparing loops

The `while` and `for` loop statements are about the same: any loop written using one type of loop statement can be written using the other loop type. Which type of statement we use depends on the situation.

A `for` loop is like a `while` loop in that the condition is evaluated before the loop body is executed. Figure 3.10 shows the general structure of `for` and `while` loops.

We generally use a `for` loop when we know how many times we want to go through a loop. Most of the time it is easier to put the code that sets up and controls the loop in the `for` loop header.

listing 3.14

```
//********************************************************************
//  IceCreamShop.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of a foreach loop on an enumerated type.
//********************************************************************

public class IceCreamShop
{
   enum Flavor {vanilla, chocolate, strawberry, fudgeRipple, coffee,
                rockyRoad, mintChocolateChip, cookieDough}

   //-----------------------------------------------------------------
   //  Prints the flavors of ice cream available.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      System.out.println ("Welcome to the Ice Cream Shop!");
      System.out.println ();

      System.out.println ("We have the following flavors:");

      for (Flavor f: Flavor.values())
      {
         System.out.println(f);
      }
   }
}
```

output

```
Welcome to the Ice Cream Shop!

We have the following flavors:
vanilla
chocolate
strawberry
fudgeRipple
coffee
rockyRoad
mintChocolateChip
cookieDough
```

```
for (initialization; condition; increment)
   statement;
```

```
initialization;
while (condition)
{
   statement;
   increment;
}
```

figure 3.10 The general structure of equivalent `for` and `while` loops

3.8 program development revisited

Now let's apply what we know to program development. Suppose a teacher wants a program that will analyze exam scores. The requirements are first given as follows. The program will:

- accept a series of test scores as input
- compute the average test score
- determine the highest and lowest test scores
- display the average, highest, and lowest test scores

Our first task is to look at the requirements. The requirements raise questions that need to be answered before we can design a solution. Understanding the requirements often means talking with the client. The client may very well have a clear idea of what the program should do, but this list of requirements does not provide enough detail.

For example, how many test scores should be processed? Will this program handle only one class size or should it handle different size classes? Is the input stored in a data file or will it be entered by the teacher, using the keyboard? What degree of accuracy does the teacher expect: two decimal places? Three? None? Should the output be in any particular format?

Let's assume we know that the program needs to handle a different number of test scores each time it is run and that the input will be entered by the teacher. The teacher wants the average presented to two decimal places, but lets us (the developer) pick the format.

Now let's consider some design questions. Because there is no limit to the number of grades that can be entered, how should the user indicate that there are no more grades? We can address this several ways. The program could ask the user, after each grade is entered, if there are more grades to process. Or the program could begin by asking the user for the total number of grades

that will be entered, then read exactly that many grades. Or, when prompted for a grade, the teacher could enter a sentinel value to say that there are no more grades to be entered.

The first option requires a lot more input from the user, which is too awkward. The second option means the user must know exactly how many grades to enter and better not make any mistakes. The third option is reasonable, but before we can pick a sentinel value to end the input, we must ask more questions. What is the range of valid grades? What would be a good sentinel value? Talking with the client again, we learn that a student cannot get a negative grade, so we can use –1 as a sentinel value.

Let's sketch out an algorithm for this program. The pseudocode for a program that reads in a list of grades and computes their average might look like this:

```
prompt for and read the first grade
while (grade does not equal -1)
{
   increment count
   sum = sum + grade
   prompt for and read another grade
}
average = sum / count
print average
```

This algorithm only calculates the average grade. Now we must change the algorithm to compute the highest and lowest grade. Further, the algorithm does not deal well with the unusual case of entering –1 for the first grade. We can use two variables, `max` and `min`, to keep track of the highest and lowest scores. The new pseudocode looks like this:

```
prompt for and read the first grade
max = min = grade
while (grade does not equal -1)
{
   increment count
   sum = sum + grade
   if (grade > max)
      max = grade
   if (grade < min)
      min = grade
   prompt for and read another grade
}
if (count is not zero)
{
   average = sum / count
   print average, highest, and lowest grades
}
```

Having planned out an algorithm for the program, we can start implementing it. Consider the solution to this problem shown in Listing 3.15.

listing 3.15

```
//********************************************************************
//  ExamGrades.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of various control structures.
//********************************************************************

import java.text.DecimalFormat;
import java.util.Scanner;

public class ExamGrades
{
   //-----------------------------------------------------------------
   //  Computes the average, minimum, and maximum of a set of exam
   //  scores entered by the user.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int grade, count = 0, sum = 0, max, min;
      double average;
      Scanner scan = new Scanner (System.in);

      //  Get the first grade and give max and min that initial value
      System.out.print ("Enter the first grade (-1 to quit): ");
      grade = scan.nextInt();

      max = min = grade;

      //  Read and process the rest of the grades
      while (grade >= 0)
      {
         count++;
         sum += grade;

         if (grade > max)
            max = grade;
         else
            if (grade < min)
               min = grade;

         System.out.print ("Enter the next grade (-1 to quit): ");
         grade = scan.nextInt ();
      }
```

Listing 3.15 continued

```
      //  Produce the final results
      if (count == 0)
         System.out.println ("No valid grades were entered.");
      else
      {
         DecimalFormat fmt = new DecimalFormat ("0.##");
         average = (double)sum / count;
         System.out.println();
         System.out.println ("Total number of students: " + count);
         System.out.println ("Average grade: " + fmt.format(average));
         System.out.println ("Highest grade: " + max);
         System.out.println ("Lowest grade: " + min);
      }
   }
}
```

output

```
Enter the first grade (-1 to quit): 89
Enter the next grade (-1 to quit): 95
Enter the next grade (-1 to quit): 82
Enter the next grade (-1 to quit): 70
Enter the next grade (-1 to quit): 98
Enter the next grade (-1 to quit): 85
Enter the next grade (-1 to quit): 81
Enter the next grade (-1 to quit): 73
Enter the next grade (-1 to quit): 69
Enter the next grade (-1 to quit): 77
Enter the next grade (-1 to quit): 84
Enter the next grade (-1 to quit): 82
Enter the next grade (-1 to quit): -1

Total number of students: 12
Average grade: 82.08
Highest grade: 98
Lowest grade: 69
```

Let's look at how this program does what the teacher wanted. After the variable declarations in the `main` method, we ask the user to enter the first grade. Prompts should tell the user about any special input requirements. In this case, we tell the user that entering –1 will indicate the end of the input.

The variables `max` and `min` are set to the first value entered. This is done using *chained assignments*. An assignment statement returns a value and can

be used as an expression. The value returned by an assignment statement is the value that gets assigned. Therefore, the value of `grade` is first assigned to `min`, then that value is assigned to `max`. If no larger or smaller grade is ever entered, the values of `max` and `min` will not change.

The `while` loop condition says that the loop body will be executed as long as the grade being processed is greater than or equal to zero. Therefore, any negative value will indicate the end of the input, even though the prompt tells the user that only –1 will end the input. This change is a slight variation on the original design and makes sure that no negative values will be counted as grades.

We use a nested `if` structure to decide if the new grade should be the highest or lowest grade. It cannot be both, so using an `else` clause is slightly more efficient. There is no need to ask whether the grade is a minimum if we already know it is a maximum.

If at least one positive grade was entered, then `count` is not equal to zero after the loop, and the `else` part of the `if` statement is executed. The average is computed by dividing the sum of the grades by the number of grades. Note that the `if` statement keeps us from trying to divide by zero in situations where no valid grades are entered. As we've mentioned before, we want to design robust programs that handle unexpected or wrong input without causing a runtime error. The solution for this problem is robust up to a point because it processes any numeric input without a problem, but it will fail if a nonnumeric value (like a string) is entered at the grade prompt.

AP* case study

To work with the AP* Case Study section for this chapter, go to www.aw.com/cssupport and look under author: Lewis/Loftus/Cocking.

3.9 drawing using conditionals and loops

Conditionals and loops can help us create interesting graphics.

The program called `Bullseye`, shown in Listing 3.16, uses a loop to draw the rings of a target. The `Bullseye` program uses an `if` statement to alternate the colors between black and white. Each ring is drawn as a filled circle (an oval of equal width and length). Because we draw the circles on top of each other, the inner circles cover the inner part of the larger circles, so they look like rings. At the end, a final red circle is drawn for the bull's-eye.

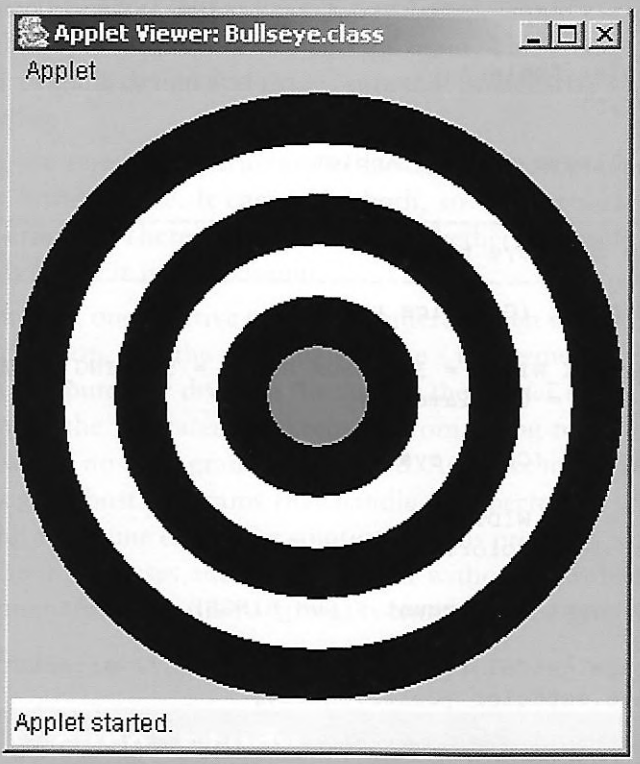
GRAPHICS TRACK

listing 3.16

```
//********************************************************************
//  Bullseye.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of conditionals and loops to guide drawing.
//********************************************************************

import java.applet.Applet;
import java.awt.*;

public class Bullseye extends Applet
{
   //-----------------------------------------------------------------
   //  Paints a bullseye target.
   //-----------------------------------------------------------------
   public void paint (Graphics page)
   {
      final int MAX_WIDTH = 300, NUM_RINGS = 5, RING_WIDTH = 25;
      int x = 0, y = 0, diameter;

      setBackground (Color.cyan);

      diameter = MAX_WIDTH;
      page.setColor (Color.white);

      for (int count = 0; count < NUM_RINGS; count++)
      {
         if (page.getColor() == Color.black)  // alternate colors
            page.setColor (Color.white);
         else
            page.setColor (Color.black);

         page.fillOval (x, y, diameter, diameter);

         diameter -= (2 * RING_WIDTH);
         x += RING_WIDTH;
         y += RING_WIDTH;
      }

      // Draw the red bullseye in the center
      page.setColor (Color.red);
      page.fillOval (x, y, diameter, diameter);
   }
}
```

listing 3.16 continued

display



Listing 3.17 shows the `Boxes` applet, in which several randomly sized rectangles are drawn in random locations. If the width of a rectangle is less than 5 pixels, the box is filled with the color yellow. If the height is less than 5 pixels, the box is filled with the color green. Otherwise, the box is drawn, unfilled, in white.

Note that in the `Boxes` program, the color is decided before each rectangle is drawn. In the `BarHeights` applet, shown in Listing 3.18, we handle the situation differently. The goal of `BarHeights` is to draw 10 vertical bars of random heights, coloring the tallest bar in red and the shortest bar in yellow.

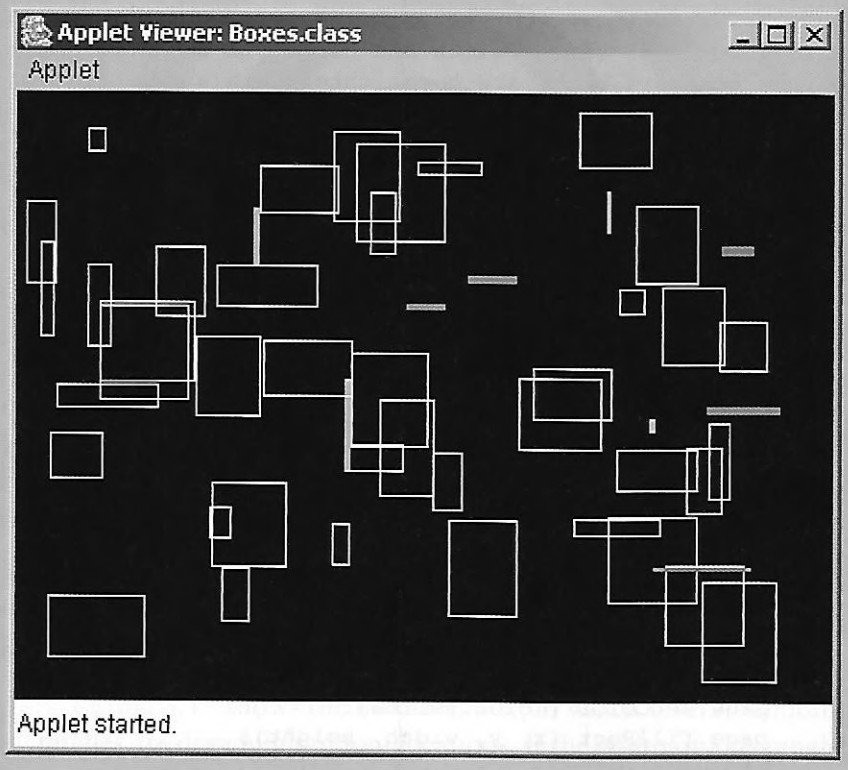
listing 3.17

```
//********************************************************************
//  Boxes.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of conditionals and loops to guide drawing.
//********************************************************************

import java.applet.Applet;
import java.awt.*;
import java.util.Random;

public class Boxes extends Applet
{
   //-----------------------------------------------------------------
   //  Paints boxes of random width and height in a random location.
   //  Narrow or short boxes are highlighted with a fill color.
   //-----------------------------------------------------------------
   public void paint(Graphics page)
   {
      final int NUM_BOXES = 50, THICKNESS = 5, MAX_SIDE = 50;
      final int MAX_X = 350, MAX_Y = 250;
      int x, y, width, height;

      setBackground (Color.black);
      Random generator = new Random();

      for (int count = 0; count < NUM_BOXES; count++)
      {
         x = generator.nextInt (MAX_X) + 1;
         y = generator.nextInt (MAX_Y) + 1;

         width = generator.nextInt (MAX_SIDE) + 1;
         height = generator.nextInt (MAX_SIDE) + 1;

         if (width <= THICKNESS)  // check for narrow box
         {
            page.setColor (Color.yellow);
            page.fillRect (x, y, width, height);
         }
         else
            if (height <= THICKNESS)  // check for short box
            {
               page.setColor (Color.green);
               page.fillRect (x, y, width, height);
            }
```

listing 3.17 continued

```
            else
            {
               page.setColor (Color.white);
               page.drawRect (x, y, width, height);
            }
      }
   }
}
```

display



In the `BarHeights` program, we don't know if the bar we are about to draw is either the tallest or the shortest because we haven't created them all yet. Therefore we keep track of the position of both the tallest and shortest bars as they are drawn. After all the bars are drawn, the program goes back and redraws these two bars in the right color.

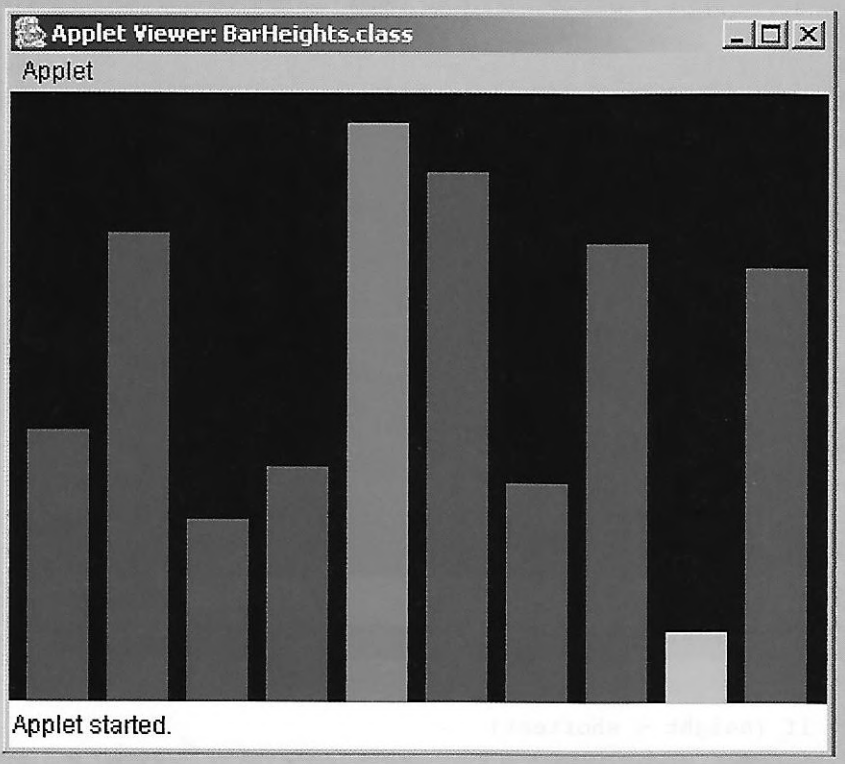
Listing 3.18

```
//********************************************************************
//  BarHeights.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of conditionals and loops to guide drawing.
//********************************************************************

import java.applet.Applet;
import java.awt.*;
import java.util.Random;

public class BarHeights extends Applet
{
   //-----------------------------------------------------------------
   //  Paints bars of varying heights, tracking the tallest and
   //  shortest bars, which are redrawn in color at the end.
   //-----------------------------------------------------------------
   public void paint (Graphics page)
   {
      final int NUM_BARS = 10, WIDTH = 30, MAX_HEIGHT = 300, GAP =9;
      int tallX = 0, tallest = 0, shortX = 0, shortest = MAX_HEIGHT;
      int x, height;

      Random generator = new Random();
      setBackground (Color.black);

      page.setColor (Color.blue);
      x = GAP;

      for (int count = 0; count < NUM_BARS; count++)
      {
         height = generator.nextInt(MAX_HEIGHT) + 1;
         page.fillRect (x, MAX_HEIGHT-height, WIDTH, height);

         // Keep track of the tallest and shortest bars
         if (height > tallest)
         {
            tallX = x;
            tallest = height;
         }

         if (height < shortest)
         {
            shortX = x;
            shortest = height;
         }
```

listing 3.18 continued

```
            x = x + WIDTH + GAP;
        }

        // Redraw the tallest bar in red
        page.setColor (Color.red);
        page.fillRect (tallX, MAX_HEIGHT-tallest, WIDTH, tallest);

        // Redraw the shortest bar in yellow
        page.setColor (Color.yellow);
        page.fillRect (shortX, MAX_HEIGHT-shortest, WIDTH, shortest);
    }
}
```

display



summary of key concepts

- Software requirements tell us *what* a program must do.
- A software design tells us *how* a program will fill its requirements.
- An algorithm is a step-by-step process for solving a problem, often written in pseudocode.
- Implementation should be the least creative of all development activities.
- The goal of testing is to find errors. We can never really be sure that all errors have been found.
- Conditionals and loops let us control the flow of execution through a method.
- An `if` statement lets a program choose whether to execute a particular statement.
- The compiler does not care about indentation. Indentation is important for human readers because it shows the relationship between one statement and another.
- An `if-else` statement lets a program do one thing if a condition is true and another thing if the condition is false.
- In a nested `if` statement, an `else` clause is matched to the closest unmatched `if`.
- Logical operators return a boolean value (true or false) and are often used for sophisticated conditions.
- The order of characters in Java is defined by the Unicode character set.
- The `compareTo` method determines the lexicographic order of strings, which is not necessarily alphabetical order.
- A `while` statement lets a program execute the same statement over and over.
- We must design our programs carefully to avoid infinite loops. The body of the loop must eventually make the loop condition false.
- An iterator is an object that helps you process a group of related items.
- A `for` statement is usually used when a loop will be executed a set number of times.

self-review questions

3.1 Name the four basic activities that are involved in a software development process.

3.2 What is an algorithm? What is pseudocode?

3.3 What is meant by the flow of control through a program?

3.4 What type of conditions are conditionals and loops based on?

3.5 What are the equality operators? The relational operators?

3.6 What is a nested `if` statement? A nested loop?

3.7 How do block statements help us construct conditionals and loops?

3.8 What is a truth table?

3.9 How do we compare strings for equality?

3.10 Why must we be careful when comparing floating point values for equality?

3.11 What is an assignment operator?

3.12 What is an infinite loop? Specifically, what causes it?

3.13 When would we use a `for` loop instead of a `while` loop?

multiple choice

3.1 Which of the following statements increase the value of `x` by 1?

I. `x++;`

II. `x = x + 1;`

III. `x += 1;`

a. I only

b. II only

c. I and III

d. II and III

e. I, II, and III

3.2 What will be printed by the following code segment?

```
boolean flag = true;
int x = -1;
if (flag && (x > 0))
   System.out.println("yes");
else if (x == 0)
   System.out.println("maybe");
else if (!flag)
   System.out.println("sometimes");
else
   System.out.println("no");
```

a. yes

b. maybe

c. sometimes

d. no

e. There will be an error because you can't mix integers and booleans in the same expression.

3.3. The expression `!f || g` is the same as which of the following?

a. `f || !g`

b. `!(f || g)`

c. `!(f && g)`

d. `!(!f && !g)`

e. `!(f && !g)`

3.4 In the following code, what value should go in the blank so that there will be exactly six lines of output?

```
for (int x = 0; x < _____; x = x + 2)
   System.out.println("-");
```

a. 5

b. 6

c. 10

d. 11

e. 13

3.5 What will be the largest value printed by the following code?

```
for (int x=5; x > 0; x--)
   for (int y=0; y < 8; y++)
      System.out.println(x*y);
```

a. 5

b. 8

c. 35

d. 40

e. 64

3.6 Assume `x` is an integer and has been initialized to some value. Consider the code

```
for (int a = 1; a < 20; a++)
   if (x < 0)
      x = a;
```

Which statement will have the same effect on the value of `x`?

a. `if (x < 0)  x = 1;`

b. `if (x < 20)  x = 19;`

c. `if (x < 0)  x = 19;`

d. `if (x < 20)  x = 20;`

e. `x = 1;`

3.7 Assume `num` and `max` are integer variables. Consider the code

```
while (num < max)
   num++;
```

Which values of `num` and `max` will cause the body of the loop to be executed exactly once?

a. `num = 1, max = 1;`

b. `num = 1, max = 2;`

c. `num = 2, max = 2;`

d. `num = 2, max = 1;`

e. `num = 1, max = 3;`

3.8 Which `for` loop is equivalent to this `while` loop?

```
int y = 5;
while (y >= 0)
{
   System.out.println(y);
   y--;
}
```

a.
```
for (int y = 0; y < 5; y++)
      System.out.println(y);
```

b.
```
for (int y = 5; y > 0; y--)
      System.out.println(y);
```

c.
```
for (int y = 5; y >= 0; y--)
      System.out.println(y);
```

d.
```
for (int y = 0; y > 5; y++)
      System.out.println(y);
```

e.
```
for (int y = 0; y > 5; y--)
      System.out.println(y);
```

3.9 Which expression tests to make sure the grade is between 0 and 100 inclusive?

a. `(grade <= 100) || (graph <= 0)`

b. `(grade <= 100) || (graph >= 0)`

c. `(grade < 101) || (graph > -1)`

d. `(grade <= 100) && (graph >= 0)`

e. `(grade >= 100) && (graph <= 0)`

3.10 Which values of `x`, `y`, `a`, or `b` will cause the `if` statement to be short-circuited?

```
if ((x > y) && (a || b))
   statement;
```

a. `x = 1, y = 1`

b. `x = 5, y = 1`

c. `x = 2, y = 1, a = true, b = false`

d. `a = false, b = true`

e. `a = false, b = false`

true/false

3.1 An `if` statement may be used to make a decision in a program.

3.2 The expression `x > 0` is the same as the expression `0 <= x`.

3.3 The operators `+=`, `*=`, `-=`, and `/=` may only be used with integers.

3.4 The expression `a || b` is the same as `a && !b`.

3.5 If the expression `a && !b` evaluates to true, then the expression `a || b` will evaluate to true.

3.6 The expression `a || b` will be short-circuited if `a` is false.

3.7 Any loop written using a `for` statement can be written using a `while` statement.

3.8 An algorithm is a step-by-step process for solving a problem.

3.9 Once an initial design is created, it should never be revised.

short answer

3.1 What happens in the `MinOfThree` program if two or more of the values are equal? If exactly two of the values are equal, does it matter whether the equal values are lower or higher than the third?

3.2 What is wrong with the following code fragment? Rewrite it so that it produces correct output.

```
if (total == MAX)
   if (total < sum)
      System.out.println ("total == MAX and is < sum.");
else
   System.out.println ("total is not equal to MAX");
```

3.3 What is wrong with the following code fragment? Will this code compile if it is part of a valid program? Explain.

```
if (length = MIN_LENGTH)
   System.out.println ("The length is minimal.");
```

3.4 What output is produced by the following code fragment?

```
int num = 87, max = 25;
if (num >= max*2)
   System.out.println ("apple");
   System.out.println ("orange");
System.out.println ("pear");
```

3.5 What output is produced by the following code fragment?

```
int limit = 100, num1 = 15, num2 = 40;
if (limit <= limit)
{
   if (num1 == num2)
      System.out.println ("lemon");
   System.out.println ("lime");
}
System.out.println ("grape");
```

3.6 Put the following list of strings in lexicographic order as if determined by the `compareTo` method of the `String` class. Consult the Unicode chart in Appendix B.

```
"fred"
"Ethel"
"?-?-?-?"
"{([])}"
"Lucy"
"ricky"
"book"
"******"
"12345"
"       "
"HEPHALUMP"
"bookkeeper"
"6789"
";+<?"
"^^^^^^^^^^"
"hephalump"
```

3.7 What output is produced by the following code fragment?

```
int num = 1, max = 20;
while (num < max)
{
   if (num%2 == 0)
      System.out.println (num);
   num++;
}
```

3.8 What output is produced by the following code fragment?

```
for (int num = 0; num <= 200; num += 2)
   System.out.println (num);
```

3.9 What output is produced by the following code fragment?

```
for (int val = 200; val >= 0; val -= 1)
   if (val % 4 != 0)
      System.out.println (val);
```

3.10 Transform the following `while` loop into a `for` loop (make sure it produces the same output).

```
int num = 1;
while (num < 20)
{
   num++;
   System.out.println (num);
}
```

3.11 What is wrong with the following code fragment? What are three ways it could be changed to remove the flaw?

```
count = 50;
while (count >= 0)
{
   System.out.println (count);
   count = count + 1;
}
```

3.12 Write a `while` loop that makes sure the user enters a positive integer value.

3.13 Write a code fragment that reads and prints integer values entered by a user until a particular sentinel value (stored in `SENTINEL`) is entered. Do not print the sentinel value.

3.14 Write a `for` loop to print the odd numbers from 1 to 99 (inclusive).

3.15 Write a `for` loop to print the multiples of 3 from 300 down to 3.

3.16 Write a `foreach` loop that prints all possible values of an enumerated type called `Month`.

3.17 Write a code fragment that reads 10 integer values from the user and prints the highest value entered.

3.18 Write a code fragment that determines and prints the number of times the character `'a'` appears in a `String` object called `name`.

3.19 Write a code fragment that prints the characters stored in a `String` object called `str` backward.

3.20 Write a code fragment that prints every other character in a `String` object called `word` starting with the first character.

programming projects

3.1 Create a new version of the `Average` program (Listing 3.6) that prevents a runtime error when the user immediately enters the sentinel value (without entering any valid values).

3.2 Design and implement an application that reads an integer value representing a year input by the user. The purpose of the program is to determine if the year is a leap year (and therefore has 29 days in February) in the Gregorian calendar. A year is a leap year if it is divisible by 4, unless it is also divisible by 100 but not 400. For example, the year 2003 is not a leap year, but 2004 is. The year 1900 is not a leap year because it is divisible by 100, but the year 2000 is a leap year because even though it is divisible by 100, it is also divisible by 400. Produce an error message for any input value less than 1582 (the year the Gregorian calendar was adopted).

3.3 Change the solution to the Programming Project 3.2 so that the user can enter more than one year. Let the user end the program by entering a sentinel value. Validate each input value to make sure it is greater than or equal to 1582.

3.4 Design and implement an application that reads an integer value and prints the sum of all even integers between 2 and the input value, inclusive. Print an error message if the input value is less than 2. Prompt the user accordingly.

3.5 Design and implement an application that reads a string from the user and prints it one character per line.

3.6 Design and implement an application that determines and prints the number of odd, even, and zero digits in an integer value read from the keyboard.

3.7 Design and implement an application that produces a multiplication table, showing the results of multiplying the integers 1 through 12 by themselves.

3.8 Create a new version of the `Counter` program (Listing 3.5) such that the `println` statement comes after the counter increment in the body of the loop. Make sure the program still produces the same output.

3.9 Design and implement an application that prints the first few verses of the traveling song "One Hundred Bottles of Beer." Use a loop so that each iteration prints one verse. Read the number of verses to print from the user. Validate the input. The following are the first two verses of the song:

100 bottles of beer on the wall

100 bottles of beer

If one of those bottles should happen to fall

99 bottles of beer on the wall

99 bottles of beer on the wall

99 bottles of beer

If one of those bottles should happen to fall

98 bottles of beer on the wall

3.10 Design and implement an application that plays the Hi-Lo guessing game with numbers (Listing 3.3). The program should pick a random number between 1 and 100 (inclusive), then keep asking the user to guess the number. On each guess, report to the user that he or she is correct or that the guess is high or low. Keep accepting guesses until the user guesses correctly or quits. Use a sentinel value to determine whether the user wants to quit. Count the number of guesses and report that value when the user guesses correctly. At the end of each game (by quitting or a correct guess), ask whether the user wants to play again. Keep playing games until the user chooses to stop.

3.11 Create a new version of the `PalindromeTester` program (Listing 3.9) so that the spaces, punctuation, and changes in uppercase and lowercase are not considered when determining whether a string is a palindrome. *Hint*: You can handle this in several ways. Think carefully about your design.

3.12 Create new versions of the `Stars` program (Listing 3.13) to print the following patterns. Create a separate program to produce each pattern. *Hint:* Parts b, c, and d require several loops, some of which print a specific number of spaces.

```
a. **********  b.          *  c. **********  d.        *
   *********               **     *********       ***
   ********               ***     ********      *****
   *******               ****      *******     *******
   ******               *****       ******   *********
   *****               ******        *****   *********
   ****               *******         ****    *******
   ***                ********         ***     *****
   **                 *********         **       ***
   *                  **********        *         *
```

3.13 Design and implement an application that reads a string from the user, then determines and prints how many of each lowercase vowel (a, e, i, o, and u) appear in the entire string. Have a separate counter for each vowel. Also count and print the number of consonants, spaces, and punctuation marks.

3.14 Design and implement an application that plays the rock-paper-scissors game against the computer. When played between two people, each person picks one of three options (usually shown by a hand gesture) at the same time, and a winner is determined. In the game, rock beats scissors, scissors beats paper, and paper beats rock. The program should randomly choose one of the three options (without revealing it), then ask for the user's selection. At that point, the program reveals both choices and prints a statement indicating that the user won, that the computer won, or that it was a tie. Keep playing until the user chooses to stop, then print the number of user wins, losses, and ties.

3.15 Design and implement an application that simulates a simple slot machine in which three numbers between 0 and 9 are randomly selected and printed side by side. Print a statement saying all three of the numbers are the same, or any two of the numbers are the same, when this happens. Keep playing until the user chooses to stop.

3.16 Design and implement an applet that draws the side view of stair steps from the lower left to the upper right.

3.17 Design and implement an applet that draws 100 circles of random color and random diameter in random locations. Make sure that in each case the whole circle appears in the visible area of the applet.

AP*-style multiple choice

3.1 Consider the following output.

```
10 9 8 7 6 5 4 3 2 1
```

Which of the following loops will produce this output?

(A)
```
for (int i = 0; i < 10; i--)
    System.out.print(i + " ");
```

(B)
```
for (int i = 10; i >= 0; i--)
    System.out.print(i + " ");
```

(C)
```
for (int i = 0; i <= 10; i++)
    System.out.print((10 - i) + " ");
```

(D)
```
for (int i = 0; i < 10; i++)
    System.out.print((10 - i) + " ");
```

(E)
```
for (int i = 10; i > 0; i--)
    System.out.print((10 - i) + " ");
```

3.2 A program has been written to process the scores of soccer games. Consider the following code segment, which is intended to assign an appropriate string to `outcome` based on the number of points scored by each of two teams. (The team with the greater number of points gains the victory.)

```
if (team1Points == team2Points)
   outcome = "Tie game";
if (team1Points > team2Points)
   outcome = "Victory for Team 1";
else
   outcome = "Victory for Team 2";
```

The code does not work properly. For which of the following cases will the code assign the **wrong** string to `outcome`?

The case where

I. both teams score the same number of points

II. Team 1 scores more points than Team 2

III. Team 2 scores more points than Team 1

(A) I only

(B) II only

(C) III only

(D) I and III only

(E) II and III only

3.3 Consider the following code segment.

```
for (int i = 1; i < 5; i++)
   for (int k = i; k > 2; k--)
      System.out.print(k + " ");
```

What is printed as a result of executing the code segment?

(A) `3 4 3`

(B) `3 4 4`

(C) `1 2 3 4 3`

(D) `2 3 2 4 3 2`

(E) Many digits are printed due to an infinite loop.

3.4 Consider the following code segment.

```
for (int i = 1; i < 25; i = i + 5)
   if (i % 5 == 0)
      System.out.print(i + " ");
```

What is printed as a result of executing the code segment?

(A) `5 10 15 20`

(B) `5 10 15 20 25`

(C) `5 15`

(D) `6 11 16 21`

(E) Nothing is printed.

3.5 Consider the following `while` loop.

```
int k = 8;
while (k > 0)
{
   k = k - 2;
   System.out.println(k);
}
```

Which of the following `for` loops produces the same output as the `while` loop?

(A)
```
for (int k=8; k >= 0; k = k – 2)
   { System.out.println(k); }
```

(B)
```
for (int k=8; k > 0; k = k – 2)
   { System.out.println(k); }
```

(C)
```
for (int k=8; k > 2; k = k – 2)
   { System.out.println(k); }
```

(D)
```
for (int k=6; k >= 0; k = k – 2)
   { System.out.println(k); }
```

(E)
```
for (int k=6; k > 0; k = k – 2)
   { System.out.println(k); }
```

3.6 Consider the following variable declarations.

```
boolean a = true, b = false;
int k = 7;
```

In which expression will short-circuiting occur?

(A) `a && b`

(B) `a && (k > 5)`

(C) `a || (k < 7)`

(D) `b || a`

(E) `(k == 7) && a`

answers to self-review questions

3.1 The four basic activities in software development are requirements analysis (deciding what the program should do), design (deciding how to do it), implementation (writing the code), and testing (validating the implementation).

3.2 An algorithm is a step-by-step process that describes the solution to a problem. Every program can be described in algorithmic terms. An algorithm is often written in pseudocode, a loose combination of English and code-like terms used to capture the basic processing steps.

3.3 The flow of control through a program determines the program statements that will be executed when the program is run.

3.4 Each conditional and loop is based on a boolean condition that evaluates to either true or false.

3.5 The equality operators are equal (==) and not equal (!=). The relational operators are less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=).

3.6 A nested `if` is an `if` statement inside an `if` or `else` clause. A nested `if` lets the programmer make a series of decisions. A nested loop is a loop within a loop.

3.7 A block statement groups several statements together. We use them to define the body of an `if` statement or loop when we want to do several things based on the boolean condition.

3.8 A truth table shows all possible results of a boolean expression, given all possible combinations of variables and conditions.

3.9 We compare strings for equality using the `equals` method of the `String` class, which returns a boolean result. The `compareTo` method of the `String` class can also be used to compare strings. It returns a positive, 0, or negative integer result depending on the relationship between the two strings.

3.10 Because they are stored internally as binary numbers, comparing floating point values for exact equality will be true only if they are the same bit-by-bit. It's better to use a reasonable tolerance value and consider the difference between the two values.

3.11 An assignment operator combines an operation with assignment. For example, the += operator performs an addition, then stores the value back into the variable on the right-hand side.

3.12 An infinite loop is a repetition statement that never ends. The body of the loop never causes the condition to become false.

3.13 A `for` loop is usually used when we know, or can calculate, how many times we want to iterate through the loop body. We use a `while` loop when we don't know how many times the loop should execute.