3 program statements

All programming languages have specific statements that allow you to perform basic operations. These statements accomplish all programmed activity, including our interaction with objects and the definition of the services those objects provide. This chapter examines several of these programming statements as well as some additional operators. It begins by exploring the basic activities that a programmer goes through when developing software. These activities form the cornerstone of high-quality software development and represent the first step toward a disciplined development process. Finally, we use the statements we examine in this chapter to augment our ability to produce graphical output.

chapter objectives

- Discuss basic program development activities.
- Define the flow of control through a program.
- Perform decision making using `if` and `switch` statements.
- Define expressions that let us make complex decisions.
- Perform statements repetitively using `while`, `do`, and `for` statements.
- Draw with the aid of conditionals and loops.

3.0 program development

Creating software involves much more than just writing code. As you learn more about the programming language statements that you can use in your problem solutions, it is also important to develop good habits in the way you develop and validate those solutions. This section introduces some of the basic programming activities necessary for developing software.

Any proper software development effort consists of four basic *development activities*:

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

It would be nice if these activities, in this order, defined a step-by-step approach for developing software. However, although they may seem to be sequential, they are almost never completely linear in reality. They overlap and interact. Let's discuss each development stage briefly.

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

key concept

Software requirements specify *what* a program must accomplish.

The person or group who wants a software product developed (the *client*) will often provide an initial set of requirements. However, these initial requirements are often incomplete, ambiguous, or even contradictory. The software developer must work with the client to refine the requirements until all key decisions about what the system will do have been addressed.

Requirements often address user interface issues such as output format, screen layouts, and graphical interface components. Essentially, the requirements establish the characteristics that make the program useful for the end user. They may also apply constraints to your program, such as how fast a task must be performed. They may also impose restrictions on the developer such as deadlines.

A *software design* indicates *how* a program will accomplish its requirements. The design specifies the classes and objects needed in a program and defines how they interact. A detailed design might even specify the individual steps that parts of the code will follow.

key concept

A software design specifies *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 no less essential. Many problems that occur in software are directly attributable to a lack of good design effort. Alternatives need to be considered and explored. Often, the first attempt at a design is not the best solution. Fortunately, changes are relatively easy to make during the design stage.

One of the most fundamental design issues 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 implements one or more algorithms. Every software developer should spend time thinking about the algorithms involved before writing any code.

An algorithm is often described using *pseudocode,* which is a mixture of code statements and English phrases. Pseudocode provides enough structure to show how the code will operate without getting bogged down in the syntactic details of a particular programming language and without being prematurely constrained by the characteristics of particular programming constructs.

key concept

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

When developing an algorithm, it's important to analyze all of the requirements involved with that part of the problem. This ensures that the algorithm takes into account all aspects of the problem. The design of a program is often revised many times before it is finalized.

Implementation is the process of writing the source code that will solve the problem. More precisely, implementation is the act of translating the design into a particular programming language. Too many programmers focus on implementation exclusively when actually it should be the least creative of all development activities. The important decisions should be made when establishing the requirements and creating the design.

Testing a program includes running it multiple times with various inputs and carefully scrutinizing the results. Testing might also include hand-tracing program code, in which the developer mentally plays the role of the computer to see where the program logic goes awry.

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 that error is much higher. Taking the time to uncover problems as early as possible is always worth the effort.

Running a program with specific input and producing the correct results establishes only that the program works for that particular input. As more and more test cases execute without revealing errors, our confidence in the program rises, but we can never really be sure that all errors have been eliminated. There could always be another error still undiscovered. Because of that, it is important to thoroughly test a program with various kinds of input. When one problem is fixed, we should run previous tests again to make sure that while fixing the problem we didn't create another. 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.

Various models have been proposed that describe the specific way in which requirements analysis, design, implementation, and testing should be accomplished. For now we will simply keep these general activities in mind as we learn to develop programs.

3.1 control flow

The order in which statements are executed in a running program is called the *flow of control*. Unless otherwise specified, the execution of a program proceeds in a linear fashion. That is, 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 executing 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 alters the flow of control. When a method is called, control jumps to the code defined for that method. When the method completes, control returns to the place in the calling method where the invocation was made and processing continues from there. In our examples thus 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 separate classes and methods in Chapter 4.

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

key concept

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

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

For example, the cost of life insurance might be dependent on whether the insured person is a smoker. If the person smokes, we calculate the cost using a 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*, allows us to execute a programming 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, and the `for` statement. Each type of loop statement has unique characteristics that distinguish it from the others.

Conditionals and loops are fundamental to controlling the flow through a method and are necessary 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);
```

An `if` statement consists of the reserved word `if` followed by a boolean expression, or condition. The condition is enclosed in parentheses and must key concept

An if statement allows a program to choose whether to execute a particular statement.

evaluate to 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 this processing.

Note that the assignment statement in this example is indented under the header line of the `if` statement. This communicates that the assignment statement is part of the `if` statement; it implies that the `if` statement governs whether the assignment statement will be executed. This indentation is extremely important for the human reader.

key concept

Even though the compiler does not care about indentation, proper 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 makes a decision as to whether to print a particular sentence based on the age that is entered.

The `Age` program echoes 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.

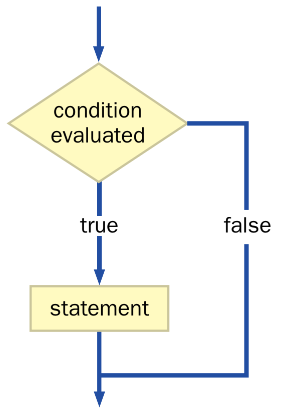


figure 3.1 The logic of an if statement

listing
3.1

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

import cs1.Keyboard;

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

      System.out.print ("Enter your age: ");
      int age = Keyboard.readInt();

      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.
```

equality and relational operators

Boolean expressions evaluate to either true or false and are fundamental to our ability to make decisions. 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, respectively. Note that the equality operator consists of 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 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.

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

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

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 approximately 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:

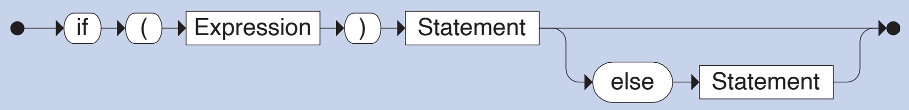
```
if (height <= MAX)
   adjustment = 0;
else
   adjustment = MAX – height;
```

If the condition is true, the first assignment statement is executed; if the condition is false, the second assignment statement is executed. Only one or the other will be executed because a boolean condition will evaluate to either true or false. Note that proper indentation is used again to communicate that the statements are part of the governing `if` statement.

key concept

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

If Statement



An `if` statement tests the boolean Expression and, if true, executes the first Statement. The optional `else` clause identifies 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;
```

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

In the `Wages` program, if an employee works over 40 hours in a week, the payment amount takes into account 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 makes use of an object called `roster`. Even without knowing what `roster` represents, or from what class it was created, we can see that it has at least three methods: `getSize`, `expand`, and `addName`. The condition of the `if`

listing
3.2

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

import java.text.NumberFormat;
import cs1.Keyboard;

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;

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

      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
```

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.

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 to delimit the main method and a class definition. The program called Guessing, shown in Listing 3.3, uses an if-else statement in which the statement of the else clause is a block statement.

If the guess entered by the user equals the randomly chosen answer, an appropriate acknowledgement is printed. However, if the answer is incorrect, two statements are printed, one that states that the guess is wrong and one that prints the actual answer. A programming project at the end of this chapter expands the concept of this example into the Hi-Lo game, which can only be done after we explore loops in more detail.

Note that if the block braces were not used, the sentence stating that the answer is incorrect would be printed if the answer 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 means nothing except to the human reader. Statements that are not blocked properly can lead to the programmer making improper assumptions about 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) implies 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
//
//  Demonstrates the use of a block statement in an if-else.
//********************************************************************

import cs1.Keyboard;
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;

      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 = Keyboard.readInt();

      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` portion 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 obtained 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 allows us to make another decision after determining the results of a previous decision. The program in Listing 3.4, called `MinOfThree`, uses nested `if` statements to determine the smallest of three integer values entered by the user.

Carefully trace the logic of the `MinOfThree` program, using various input sets with the minimum value in all three positions, to see how it determines 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
//
//  Demonstrates the use of nested if statements.
//********************************************************************

import cs1.Keyboard;

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;

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

      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 implies 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 preceded 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 crucial.

key concept

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

Braces can be used to specify the if statement to which an else clause belongs. For example, if the previous example should have been structured so that the string "Bravo!" is printed if code is not equal to 'R', we could force that relationship (and properly indent) as follows:

```
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 the switch statement

Another conditional statement in Java is called the *switch statement*, which causes the executing program to follow one of several paths based on a single value. We also discuss the *break statement* in this section because it is usually used with a switch statement.

The switch statement evaluates an expression to determine a value and then matches that value with one of several possible *cases*. Each case has statements associated with it. After evaluating the expression, control jumps to the statement associated with the first case that matches the value. Consider the following example:

```
switch (idChar)
{
   case 'A':
      aCount = aCount + 1;
      break;
   case 'B':
      bCount = bCount + 1;
      break;
   case 'C':
      cCount = cCount + 1;
      break;
   default:
      System.out.println ("Error in Identification Character.");
}
```

First, the expression is evaluated. In this example, the expression is a simple `char` variable. Execution then transfers to the first statement identified by the case value that matches the result of the expression. Therefore, if `idChar` contains an `'A'`, the variable `aCount` is incremented. If it contains a `'B'`, the case for `'A'` is skipped and processing continues where `bCount` is incremented.

If no case value matches that of the expression, execution continues with the optional *default case,* indicated by the reserved word `default`. If no default case exists, no statements in the `switch` statement are executed and processing continues with the statement after the switch. It is often a good idea to include a default case, even if you don't expect it to be executed.

When a `break` statement is encountered, processing jumps to the statement following the `switch` statement. A `break` statement is usually used to break out of each case of a `switch` statement. Without a `break` statement, processing continues into the next case of the `switch`. Therefore if the `break` statement at the end of the `'A'` case in the previous example was not there, both the `aCount` and `bCount` variables would be incremented when the `idChar` contains an `'A'`. Usually we want to perform only one case, so a `break` statement is almost always used. Occasionally, though, the "pass through" feature comes in handy.

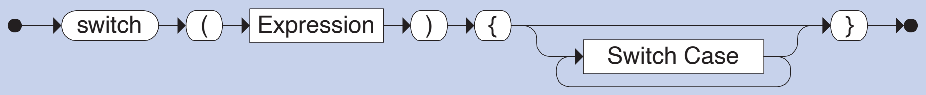
key concept

A `break` statement is usually used at the end of each case alternative of a `switch` statement to jump to the end of the switch.

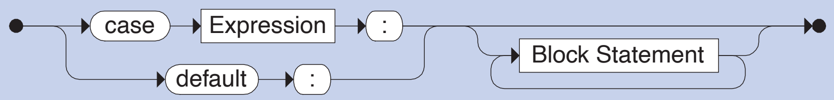
The expression evaluated at the beginning of a `switch` statement must be an *integral type*, meaning that it is either an `int` or a `char`. It cannot evaluate to a `boolean` or floating point value, and even other integer types (`byte`, `short`, and `long`) cannot be used. Furthermore, each case value must be a constant; it cannot be a variable or other expression.

Note that the implicit boolean condition of a `switch` statement is based on equality. The expression at the beginning of the statement is compared to each case value to determine which one it equals. A `switch` statement cannot be used to determine other relational operations (such as less than), unless some preliminary processing is done. For example, the `GradeReport` program in Listing 3.5 prints a comment based on a numeric grade that is entered by the user.

Switch Statement



Switch Case



The `switch` statement evaluates the initial Expression and matches its value with one of the cases. Processing continues with the Statement corresponding to that case. The optional `default` case will be executed if no other case matches.

Example:

```
switch (numValues)
{
   case 0:
      System.out.println ("No values were entered.");
      break;
   case 1:
      System.out.println ("One value was entered.");
      break;
   case 2:
      System.out.println ("Two values were entered.");
      break;
   default:
      System.out.println ("Too many values were entered.");
}
```

listing 3.5

```
//********************************************************************
//  GradeReport.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a switch statement.
//********************************************************************

import cs1.Keyboard;

public class GradeReport
{
   //-----------------------------------------------------------------
   //  Reads a grade from the user and prints comments accordingly.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int grade, category;

      System.out.print ("Enter a numeric grade (0 to 100): ");
      grade = Keyboard.readInt();

      category = grade / 10;

      System.out.print ("That grade is ");

      switch (category)
      {
         case 10:
            System.out.println ("a perfect score. Well done.");
            break;
         case 9:
            System.out.println ("well above average. Excellent.");
            break;
         case 8:
            System.out.println ("above average. Nice job.");
            break;
         case 7:
            System.out.println ("average.");
            break;
         case 6:
            System.out.println ("below average. You should see the");
            System.out.println ("instructor to clarify the material "
                                + "presented in class.");
            break;
```

listing 3.5 continued

```
            default:
               System.out.println ("not passing.");
         }
      }
   }
```

output

```
Enter a numeric grade (0 to 100): 86
That grade is above average. Nice job.
```

In `GradeReport`, the category of the grade is determined by dividing the grade by 10 using integer division, resulting in an integer value between 0 and 10 (assuming a valid grade is entered). This result is used as the expression of the `switch`, which prints various messages for grades 60 or higher and a default sentence for all other values.

key concept

A `switch` statement could be implemented as a series of `if-else` statements, but the `switch` is sometimes a more convenient and readable construct.

Note that any `switch` statement could be implemented as a set of nested `if` statements. However, nested `if` statements quickly become difficult for a human reader to understand and are error prone to implement and debug. But because a `switch` can evaluate only equality, sometimes nested `if` statements are necessary. It depends on the situation.

3.4 boolean expressions revisited

Let's examine a few more options regarding the use 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.

The `!` operator is used to perform the *logical NOT* operation, which is also called the *logical complement*. The logical complement of a boolean value yields

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

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 possible 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 depicts 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

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

key concept

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

reverses that value. The value of count is either greater than MAX or it isn't. The truth table in Fig. 3.6 breaks down all of the possibilities.

An important characteristic of the && and || operators is that they are "short-circuited." That is, if their left operand is sufficient 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.

Sometimes you can capitalize on the fact that the operation is short-circuited. For example, the condition in the following if statement will not attempt to divide by zero if the left operand is false. If count has the value zero, the left side of the && operation is false; therefore the whole expression is false and the right side is not evaluated.

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

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

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

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 defines an ordering of 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 relative ordering 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);
```

The Unicode character set is structured so that all lowercase alphabetic characters ('a' through 'z') are contiguous and in alphabetical order. The same is true of uppercase alphabetic characters ('A' through 'Z') and characters that represent digits ('0' through '9'). The digits precede the uppercase alphabetic characters, which precede the lowercase alphabetic characters. Before, after, and in between these groups are other characters. (See the chart in Appendix C.)

key concept

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

These relationships make 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 inherent relationships among characters in the character set.

However, you should not use the equality or relational operators to compare String objects. The String class contains a method called equals that returns a boolean value that is true if the two strings being compared contain exactly the same characters, and false otherwise. 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 an exact match. Because both objects were created from the String class, they both respond to the equals message. Therefore the condition could have been written as name2.equals(name1) and the same result would occur.

It is valid to 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 issue 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 versatile than the equals method. Instead of returning a boolean value, the compareTo method returns an integer. The return value is negative if the String object through which the method is invoked precedes (is less than) the string that is passed in as a parameter. The return value is zero if the two strings contain the same characters. The return value is positive if the String object through which the method is invoked follows (is greater than) the string that is passed in as a parameter. 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 C). This is called a *lexicographic ordering*. If all alphabetic characters are in the same case (upper or lower), the lexicographic ordering will be alphabetic ordering as well. 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 can be used to determine the relative order of strings. It determines lexicographic order, which does not correspond exactly to alphabetical order.

comparing floats

Another interesting situation occurs when comparing floating point data. 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, it may be unlikely that they are exactly equal even if they are close enough for the specific situation.

A better way to check for floating point equality is to compute 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 are willing to consider them equal. Comparing two floating point values, `f1` and `f2`, could be accomplished 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.

3.5 more operators

Before moving on to repetition statements, let's examine a few more Java operators to give us even more flexibility in the way we 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 that make up the operator cannot be separated by white space. The *decrement operator* (--) is similar except that it subtracts 1 from the value. They are both unary operators because they operate on only one operand. The following statement causes the value of `count` to be incremented.

```
count++;
```

The result is stored back into the variable `count`. Therefore it is functionally equivalent to the following statement:

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

The increment and decrement operators can be applied after the variable (such as `count++ or count--`), creating what is called the *postfix form* of the operator. They can also be applied before the variable (such as `++count` or `--count`), in what is called the *prefix form*. When used alone in a statement, the prefix and postfix forms are functionally equivalent. That is, it doesn't matter if you write

```
count++;
```

or

```
++count;
```

However, when such a form is written as a statement by itself, it is usually written in its postfix form.

When the increment or decrement operator is used in a larger expression, it can yield different results depending on the form used. For example, if the variable `count` currently contains the value `15`, the following statement assigns the value `15` to `total` and the value `16` to `count`:

```
total = count++;
```

However, the following statement assigns the value `16` to both `total` and `count`:

```
total = ++count;
```

The value of `count` is incremented in both situations, but the value used in the larger expression depends on whether a prefix or postfix form of the increment operator is used, as described in Fig. 3.7.

Expression	Operation	Value of Expression
`count++`	add 1 to `count`	the original value of `count`
`++count`	add 1 to `count`	the new value of `count`
`count--`	subtract 1 from `count`	the original value of `count`
`--count`	subtract 1 from `count`	the new value of `count`

figure 3.7 Prefix and postfix forms of the increment and decrement operators

Because of the subtle differences between the prefix and postfix forms of the increment and decrement operators, they should be used with care. As always, favor the side of readability.

key concept

The prefix and postfix increment and decrement operators have subtle effects on programs because of differences in when they are evaluated.

assignment operators

As a convenience, several *assignment operators* have been defined in Java that combine a basic operation with assignment. For example, the += operator can be used as follows:

```
total += 5;
```

It performs the same operation 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. Therefore, the following statement:

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

is equivalent to:

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

Many similar assignment operators are defined in Java, as listed in Fig. 3.8. (Appendix E discusses additional operators.)

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

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

is equivalent to:

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

Likewise, the following statement:

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

is equivalent to:

```
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`
`<<=`	left shift, then assignment	`x <<= y`	`x = x << y`
`>>=`	right shift with sign, then assignment	`x >>= y`	`x = x >> y`
`>>>=`	right shift with zero, then assignment	`x >>>= y`	`x = x >>> y`
`&=`	bitwise AND, then assignment	`x &= y`	`x = x & y`
`&=`	boolean AND, then assignment	`x &= y`	`x = x & y`
`^=`	bitwise XOR, then assignment	`x ^= y`	`x = x ^ y`
`^=`	boolean XOR, then assignment	`x ^= y`	`x = x ^ y`
`|=`	bitwise OR, then assignment	`x |= y`	`x = x | y`
`|=`	boolean OR, then assignment	`x |= y`	`x = x | y`

figure 3.8 Java assignment operators

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

the conditional operator

The Java *conditional operator* is a *ternary operator* because it requires three operands. The symbol for the conditional operator is usually written `?:`, but it is not like other operators in that the two symbols that make it up are always separated. The following is an example of an expression that contains the conditional operator:

```
(total > MAX) ? total + 1 : total * 2;
```

Preceding the `?` is a boolean condition. Following the `?` are two expressions separated by the `:` symbol. The entire conditional expression returns the value of the first expression if the condition is true, and the value of the second expression if the condition is false. Keep in mind that this is an expression that returns a value, and usually we want to do something with that value, such as assign it to a variable:

```
total = (total > MAX) ? total + 1 : total * 2;
```

In many ways, the `?:` operator serves like an abbreviated `if-else` statement. Therefore the previous statement is functionally equivalent to, but sometimes more convenient than, the following:

```
if (total > MAX)
   total = total + 1;
else
   total = total * 2;
```

The two expressions that define the larger conditional expression must evaluate to the same type. Consider the following declaration:

```
int larger = (num1 > num2) ? num1 : num2;
```

If `num1` is greater than `num2`, the value of `num1` is returned and used to initialize the variable `larger`. If not, the value of `num2` is returned and used. Similarly, the following statement prints the smaller of the two values:

```
System.out.println ("Smaller: " + ((num1 < num2) ? num1 : num2));
```

The conditional operator is occasionally helpful to evaluate a short condition and return a result. It is not a replacement for an `if-else` statement, however, because the operands to the `?:` operator are expressions, not necessarily full statements. Even when the conditional operator is a viable alternative, you should use it sparingly because it is often less readable than an `if-else` statement.

3.6 the while statement

As we discussed earlier in this chapter, a repetition statement (or loop) allows us to execute another statement multiple times. 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 key concept

A while statement allows a program to execute the same statement multiple times.

true, the body is executed again. This repetition continues until the condition becomes false; then processing continues with the statement after the body of the while loop. Figure 3.9 shows this processing.

The Counter program shown in Listing 3.6 simply prints the values from 1 to 5. Each iteration through the loop prints one value, then increments the counter. A constant called LIMIT is used to hold the maximum value that count is allowed to reach.

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

Let's look at another program that uses a while loop. The Average program shown in Listing 3.7 reads a series of integer values from the user, sums 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 indicate that the user is done entering numbers. In this program, we designate zero to be a *sentinel value* that indicates the end of the input. 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 the average. A sentinel value must always be outside the normal range of values entered.

Note that in the Average program, a variable called sum is used to maintain a *running sum,* which means it is the sum of the values entered thus far. The variable sum is initialized to zero, and each value read is added to and stored back into sum.

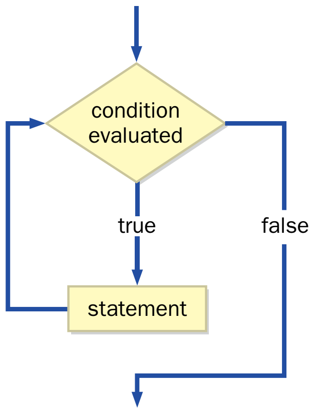


figure 3.9 The logic of a while loop

listing 3.6

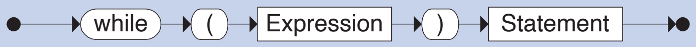
```
//********************************************************************
//  Counter.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a while loop.
//********************************************************************

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
```

We also have to count the number of values that are entered so that after the loop concludes we can divide by the appropriate value to compute the average. Note that the sentinel value is not counted. Consider the unusual situation in which 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.

While Statement



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

Example:

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

Let's examine yet another program that uses a `while` loop. The `WinPercentage` program shown in Listing 3.8 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 continues to execute, repeatedly prompting 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 potential problems as elegantly as possible.

listing
3.7

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

import java.text.DecimalFormat;
import cs1.Keyboard;

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;

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

      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 = Keyboard.readInt();
      }

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

      average = (double)sum / count;

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

listing 3.7 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
```

infinite loops

It is the programmer's responsibility to ensure that the condition of a loop will eventually become false. If it doesn't, the loop body will execute 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 body of the loop must eventually make the loop condition false.

The program shown in Listing 3.9 demonstrates an infinite loop. If you execute this program, be prepared to interrupt it. On most systems, pressing the Control-C keyboard combination (hold down the Control key and press C) terminates a running program.

listing
3.8

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

import java.text.NumberFormat;
import cs1.Keyboard;

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;

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

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

      ratio = (double)won / NUM_GAMES;

      NumberFormat fmt = NumberFormat.getPercentInstance();

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

listing 3.8 continued

output

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

Winning percentage: 58%
```

In the `Forever` program, the initial value of `count` is 1 and it is 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.

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` is initialized to 1 and is moving in a positive direction. However, note that it is being incremented 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 correct 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 similar to one we discussed earlier in this chapter when we explored the idea of comparing floating point values in the condition of an `if` statement. Because of the way the values are represented in binary, minute computational deficiencies occur internally that make a direct comparison of floating point values (for equality) problematic.

listing 3.9

```
//********************************************************************
//  Forever.java       Author: Lewis/Loftus
//
//  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
```

nested loops

The body of a loop can contain another loop. This situation is called a *nested loop*. Keep in mind that for each iteration of the outer loop, 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");
      count1++;
   }
   count2++;
}
```

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. For each iteration of the outer loop, the inner loop executes completely. Therefore the println statement is executed 500 times.

As with any loop situation, we must be careful to scrutinize 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 3.10 tests to see whether a string is a palindrome. The user may test as many strings as desired.

listing 3.10

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

import cs1.Keyboard;

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

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

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

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

listing 3.10 continued

```
            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 = Keyboard.readString();
      }
   }
}
```

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
```

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. They initially indicate the characters on either end of the string. Each iteration 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.

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

the `StringTokenizer` class

Let's examine another useful class from the Java standard class library. The types of problems this class helps us solve are inherently repetitious. Therefore the solutions almost always involve loops.

To the Java compiler, a string is just a series of characters, but often we can identify separate, important elements within a string. Extracting and processing the data contained in a string is a common programming activity. The individual elements that comprise the string are referred to as *tokens*, and therefore the process of extracting these elements is called *tokenizing* the string. The characters that are used to separate one token from another are called *delimiters*.

For example, we may want to separate a sentence such as the following into individual words:

```
"The quick brown fox jumped over the lazy dog"
```

In this case, each word is a token and the space character is the delimiter. As another example, we may want to separate the elements of a URL such as:

```
"www.csc.villanova.edu/academics/courses"
```

The delimiters of interest in this case are the period (`.`) and the slash (`/`). In yet another situation we may want to extract individual data values from a string, such as:

```
"75.43 190.49 69.58 140.77"
```

The delimiter in this case is once again the space character. A second step in processing this data is to convert the individual token strings into numeric values. This kind of processing is performed by the code inside the `Keyboard` class. When we invoke a `Keyboard` method such as `readDouble` or `readInt`, the data is initially read as a string, then tokenized, and finally converted into the appropriate numeric form. If there are multiple values on one line, the `Keyboard` class keeps track of them and extracts them as needed. We discuss `Keyboard` class processing in more detail in Chapter 5.

The `StringTokenizer` class, which is part of the `java.util` package in the Java standard class library, is used to separate a string into tokens. The default delimiters used by the `StringTokenizer` class are the space, tab, carriage return, and newline characters. Figure 3.10 lists some methods of the `StringTokenizer` class. Note that the second constructor provides a way to specify another set of delimiters for separating tokens. Once the `StringTokenizer` object is created, a call to the `nextToken` method returns the next token from the string. The `hasMoreTokens` method, which returns a `boolean` value, is often used in the condition of a loop to determine whether more tokens are left to process in the string.

The `CountWords` program shown in Listing 3.11 uses the `StringTokenizer` class and a nested `while` loop to analyze several lines of text. The user types in as many lines of text as desired, terminating them with a line that contains only the word `"DONE"`. Each iteration of the outer loop processes one line of text. The inner loop extracts and processes the tokens in the current line. The program counts the total number of words and the total number of characters in the words. After the sentinel value (which is not counted) is entered, the results are displayed.

`StringTokenizer (String str)`
  Constructor: creates a new `StringTokenizer` object to parse the specified string `str` based on white space.

`StringTokenizer (String str, String delimiters)`
  Constructor: creates a new `StringTokenizer` object to parse the specified string `str` based on the specified set of `delimiters`.

`int countTokens ()`
  Returns the number of tokens still left to be processed in the string.

`boolean hasMoreTokens ()`
  Returns true if there are tokens still left to be processed in the string.

`String nextToken ()`
  Returns the next token in the string.

figure 3.10 Some methods of the `StringTokenizer` class

Note that the punctuation characters in the strings are included with the tokenized words because the program uses only the default delimiters of the `StringTokenizer` class. Modifying this program to ignore punctuation is left as a programming project.

other loop controls

We've seen how the `break` statement can be used to break out of the cases of a `switch` statement. The `break` statement can also be placed in the body of any loop, even though this is usually inappropriate. Its effect on a loop is similar to its effect on a `switch` statement. The execution of the loop is stopped, and the statement following the loop is executed.

It is never necessary to use a `break` statement in a loop. An equivalent loop can always be written without it. Because the `break` statement causes program flow to jump from one place to another, using a `break` in a loop is not good practice. Its use is tolerated in a `switch` statement because an equivalent `switch` statement cannot be written without it. However, you can and should avoid it in a loop.

A *continue statement* has a similar effect on loop processing. The `continue` statement is similar to a `break`, but the loop condition is evaluated again, and the loop body is executed again if it is still true. Like the `break` statement, the `continue` statement can always be avoided in a loop, and for the same reasons, it should be.

listing
3.11

```java
//********************************************************************
//  CountWords.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the StringTokenizer class and nested
//  loops.
//********************************************************************

import cs1.Keyboard;
import java.util.StringTokenizer;

public class CountWords
{
   //-----------------------------------------------------------------
   //  Reads several lines of text, counting the number of words
   //  and the number of non-space characters.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int wordCount = 0, characterCount = 0;
      String line, word;
      StringTokenizer tokenizer;

      System.out.println ("Please enter text (type DONE to quit):");

      line = Keyboard.readString();
      while (!line.equals("DONE"))
      {
         tokenizer = new StringTokenizer (line);
         while (tokenizer.hasMoreTokens())
         {
            word = tokenizer.nextToken();
            wordCount++;
            characterCount += word.length();
         }
         line = Keyboard.readString();
      }

      System.out.println ("Number of words: " + wordCount);
      System.out.println ("Number of characters: " + characterCount);
   }
}
```

listing 3.11 continued

output

```
Please enter text (type DONE to quit):
Mary had a little lamb; its fleece was white as snow.
And everywhere that Mary went, the fleece shed all
over and made quite a mess. Little lambs do not make
good house pets.
DONE
Number of words: 34
Number of characters: 141
```

web bonus

The book's Web site contains a discussion of the `break` and `continue` statements, but in general their use should be avoided.

3.7 the do statement

key concept

A `do` statement executes its loop body at least once.

The *do statement* is similar to the `while` statement except that its termination condition is at the end of the loop body. Like the `while` loop, the `do` loop executes the statement in the loop body until the condition becomes false. The condition is written at the end of the loop to indicate that it is not evaluated until the loop body is executed. Note that the body of a `do` loop is always executed at least once. Figure 3.11 shows this processing.

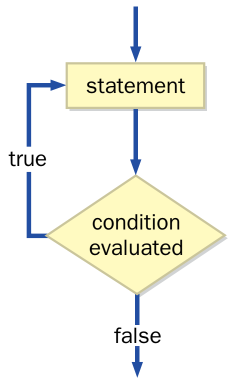
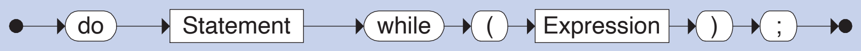


figure 3.11 The logic of a `do` loop

Do Statement



The do loop repeatedly executes the specified Statement as long as the boolean Expression is true. The Statement is executed at least once, then the Expression is evaluated to determine whether the Statement should be executed again.

Example:

```
do
{
   System.out.print ("Enter a word:");
   word = Keyboard.readString();
   System.out.println (word);
}
while (!word.equals("quit"));
```

The program Counter2 shown in Listing 3.12 uses a do loop to print the numbers 1 to 5, just as we did in an earlier version of this program with a while loop.

Note that the do loop begins simply with the reserved word do. The body of the do loop continues until the *while clause* that contains the boolean condition that determines whether the loop body will be executed again. Sometimes it is difficult to determine whether a line of code that begins with the reserved word while is the beginning of a while loop or the end of a do loop.

Let's look at another example of the do loop. The program called ReverseNumber, shown in Listing 3.13, reads an integer from the user and reverses its digits mathematically.

The do loop in the ReverseNumber program uses the remainder operation to determine the digit in the 1's position, then adds it into the reversed number, then truncates that digit from the original number using integer division. The do loop terminates when we run out of digits to process, which corresponds to the point when the variable number reaches the value zero. Carefully trace the logic of this program with a few examples to see how it works.

If you know you want to perform the body of a loop at least once, then you probably want to use a do statement. A do loop has many of the same properties as a while statement, so it must also be checked for termination conditions to avoid infinite loops.

listing 3.12

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

public class Counter2
{
   //-----------------------------------------------------------------
   //  Prints integer values from 1 to a specific limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int LIMIT = 5;
      int count = 0;

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

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

output

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

listing 3.13

```java
//********************************************************************
//  ReverseNumber.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a do loop.
//********************************************************************

import cs1.Keyboard;

public class ReverseNumber
{
   //-----------------------------------------------------------------
   //  Reverses the digits of an integer mathematically.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int number, lastDigit, reverse = 0;

      System.out.print ("Enter a positive integer: ");
      number = Keyboard.readInt();

      do
      {
         lastDigit = number % 10;
         reverse = (reverse * 10) + lastDigit;
         number = number / 10;
      }
      while (number > 0);

      System.out.println ("That number reversed is " + reverse);
   }
}
```

output

```
Enter a positive integer: 2846
That number reversed is 6482
```

3.8 the for statement

The while and the do statements are good to use when you don't initially know how many times you want to execute the loop body. The *for statement* is another repetition statement that is particularly well suited for executing the body of a loop a specific number of times that can be determined before the loop is executed.

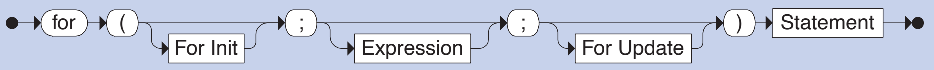
A for statement is usually used when a loop will be executed a set number of times.

key concept

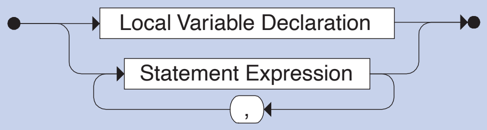
The Counter3 program shown in Listing 3.14 once again prints the numbers 1 through 5, except this time we use a for loop to do it.

The header of a for loop contains three parts separated by semicolons. Before the loop begins, the first part of the header, called the *initialization*, is executed. The second part of the header is the boolean condition, which is evaluated before the loop body (like the while loop). If true, the body of the loop is executed, followed by the execution of the third part of the header, which is called the *increment*. Note that the initialization part is executed only once, but the

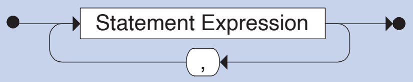
For Statement



For Init



For Update



The for statement repeatedly executes the specified Statement as long as the boolean Expression is true. The For Init portion of the header is executed only once, before the loop begins. The For Update portion 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
3.14

```
//********************************************************************
//  Counter3.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a for loop.
//********************************************************************

public class Counter3
{
   //-----------------------------------------------------------------
   //  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
```

increment part is executed after each iteration of the loop. Figure 3.12 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 map to the equivalent parts of the original `Counter` program that uses a `while` loop. The initialization por-

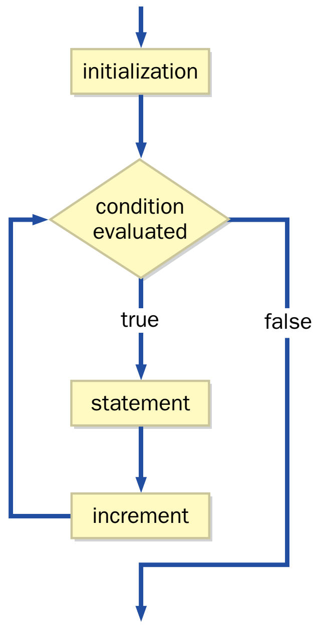


figure 3.12 The logic of a `for` loop

tion of the `for` loop header is used to declare the variable `count` as well as to give it an initial value. We are not required to declare a variable there, but it is common practice in situations where the variable is not needed outside of the loop. Because `count` is declared in the `for` loop header, it exists only inside the loop body and cannot be referenced elsewhere. The loop control variable is set up, checked, and modified by the actions in the loop header. It can be referenced inside the loop body, but it should not be modified except by the actions defined in the loop header.

The increment portion of the `for` loop header, despite 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 portion of the `for` loop can perform any calculation, not just a simple increment or decrement. Consider the program shown in Listing 3.15, which prints multiples of a particular value up to a particular limit.

listing
3.15

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

import cs1.Keyboard;

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;

      System.out.print ("Enter a positive value: ");
      value = Keyboard.readInt();

      System.out.print ("Enter an upper limit: ");
      limit = Keyboard.readInt();

      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();
      }
   }
}
```

listing 3.15 continued

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 increment portion of the `for` loop in the `Multiples` program adds the value entered by the user after each iteration. 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.

The `Stars` program in Listing 3.16 shows the use of nested `for` loops. The output is a triangle shape made of asterisk characters. The outer loop executes exactly 10 times. Each iteration of the outer loop prints one line of the output. The inner loop performs a different number of iterations depending on the line value controlled by the outer loop. Each iteration of the inner loop prints one star on the current line. Writing programs that print variations on this triangle configuration are included in the programming projects at the end of the chapter.

comparing loops

The three loop statements (`while`, `do`, and `for`) are functionally equivalent. Any particular loop written using one type of loop can be written using either of the other two loop types. Which type of loop we use depends on the situation.

listing 3.16

```
//********************************************************************
//  Stars.java       Author: Lewis/Loftus
//
//  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

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

As we mentioned earlier, the primary difference between a `while` loop and a `do` loop is when the condition is evaluated. If we know we want to execute the loop body at least once, a `do` loop is usually the better choice. The body of a while loop, on the other hand, might not be executed at all if the condition is initially false. Therefore we say that the body of a while loop is executed zero or more times, but the body of a do loop is executed one or more times.

A for loop is like a while loop in that the condition is evaluated before the loop body is executed. Figure 3.13 shows the general structure of equivalent for and while loops.

We generally use a for loop when the number of times we want to iterate through a loop is fixed or can be easily calculated. In many situations, it is simply more convenient to separate the code that sets up and controls the loop iterations inside the for loop header from the body of the loop.

3.9 program development revisited

Now that we've added several more programming language statements and operators to our repertoire, let's apply them to the program development activities that we discussed at the beginning of this chapter. Suppose an instructor wants a program that will analyze exam scores. The initial requirements are 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 requirements analysis. The initial requirements raise questions that need to be answered before we can design a suitable solution. Clarifying

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

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

figure 3.13 The general structure of equivalent for and while loops requirements often involves an extended dialog with the client. The client may very well have a clear vision about what the program should do, but this list of requirements does not provide enough detail.

For example, how many test scores should be processed? Is this program intended to handle a particular class size or should it handle varying size classes? Is the input stored in a data file or should it be entered interactively? Should the average be computed to a specific degree of accuracy? Should the output be presented in any particular format?

Let's assume that after conferring with the client, we establish that the program needs to handle an arbitrary number of test scores each time it is run and that the input should be entered interactively. Furthermore, the client wants the average presented to two decimal places, but otherwise allows us (the developer) to specify the details of the output 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 situation in several possible ways. The program could prompt the user after each grade is entered, asking if there are more grades to process. Or the program could prompt the user initially for the total number of grades that will be entered, then read exactly that many grades. A third option: When prompted for a grade, the instructor could enter a sentinel value that indicates that there are no more grades to be entered.

The first option requires a lot more input from the user and therefore is too cumbersome a solution. The second option seems reasonable, but it forces the user to have an exact count of the number of grades to enter and therefore may not be convenient. The third option is reasonable, but before we can pick an appropriate sentinel value to end the input, we must ask additional questions. What is the range of valid grades? What would be an appropriate value to use as a sentinel value? After conferring with the client again, we establish that a student cannot receive a negative grade, therefore the use of 1 as a sentinel value in this situation will work.

Let's sketch out an initial algorithm for this program. The pseudocode for a program that reads in a list of grades and computes their average might be expressed as follows:

```
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 addresses only the calculation of the average grade. Now we must refine the algorithm to compute the highest and lowest grade. Further, the algorithm does not deal elegantly 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 augmented pseudocode is now as follows:

```
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 initial algorithm for the program, the implementation can proceed. Consider the solution to this problem shown in Listing 3.17.

Let's examine how this program accomplishes the stated requirements and critique the implementation. After the variable declarations in the `main` method, we prompt the user to enter the value of the first grade. Prompts should provide information about any special input requirements. In this case, we inform the user that entering a value of 1 will indicate the end of the input.

The variables `max` and `min` are initially set to the first value entered. Note that this is accomplished 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`. In the unusual case that no larger or smaller grade is ever entered, the initial values of `max` and `min` will not change.

listing
3.17

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

import java.text.DecimalFormat;
import cs1.Keyboard;

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;

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

      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 = Keyboard.readInt ();
      }
```

listing
3.17 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
```

The `while` loop condition specifies that the loop body will be executed as long as the current grade being processed is greater than zero. Therefore, in this implementation, any negative value will indicate the end of the input, even though the prompt suggests a specific value. This change is a slight variation on the original design and ensures that no negative values will be counted as grades.

The implementation uses a nested `if` structure to determine if the new grade is a candidate for 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 was a maximum.

If at least one positive grade was entered, then `count` is not equal to zero after the loop, and the `else` portion 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 prevents us from attempting 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 erroneous 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.

3.10 drawing using conditionals and loops

Although they are not specifically related to graphics, conditionals and loops greatly enhance our ability to generate interesting graphics.

The program called `Bullseye` shown in Listing 3.18 uses a loop to draw a specific number of rings of a target. The `Bullseye` program uses an `if` statement to alternate the colors between black and white. Note that each ring is actually 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, creating the ring effect. At the end, a final red circle is drawn for the bull's-eye.

Listing 3.19 shows the `Boxes` applet, in which several randomly sized rectangles are drawn in random locations. If the width of a rectangle is below a certain thickness (5 pixels), the box is filled with the color yellow. If the height is less than the same minimal thickness, the box is filled with the color green. Otherwise, the box is drawn, unfilled, in white.

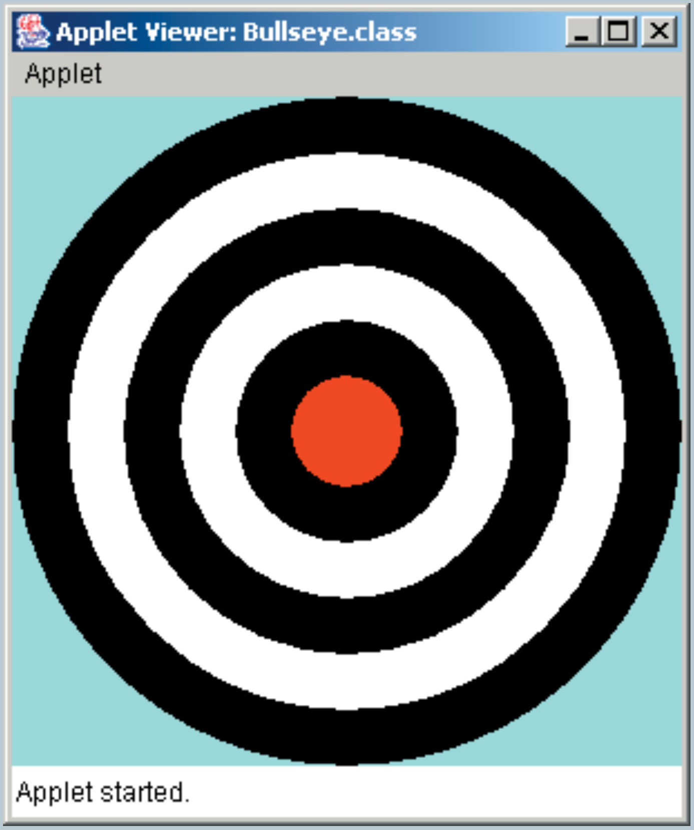
listing
3.18

```
//********************************************************************
//  Bullseye.java       Author: Lewis/Loftus
//
//  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;
      }
```

listing 3.18 continued

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

display



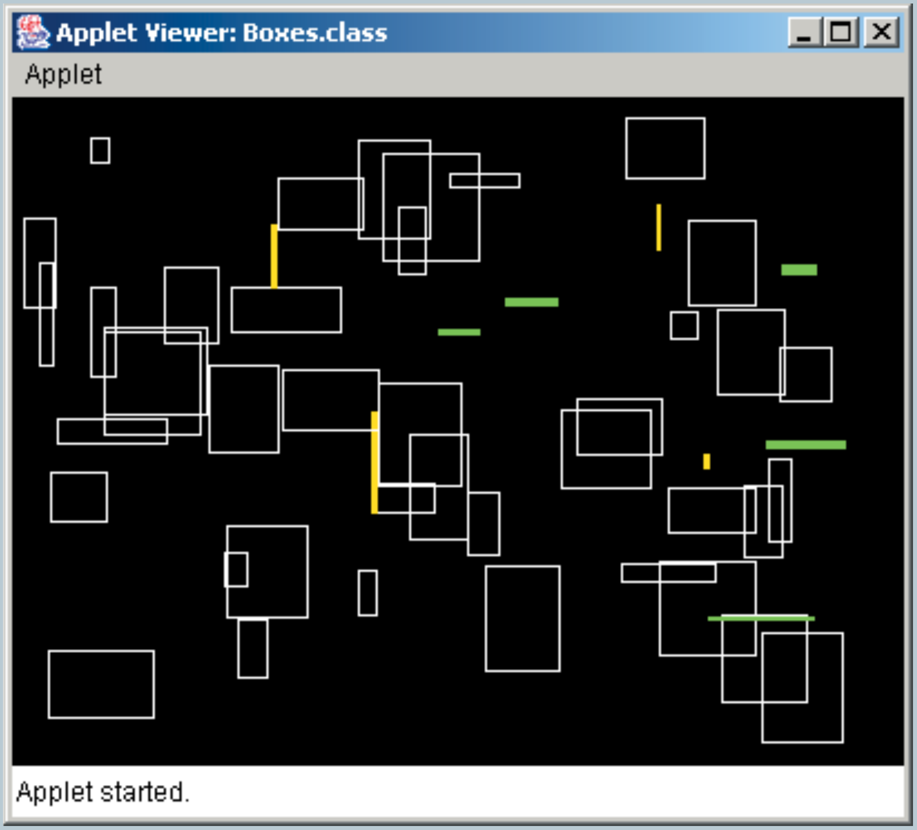
listing
3.19

```
//********************************************************************
//  Boxes.java       Author: Lewis/Loftus
//
//  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
```

listing 3.19 continued

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

display



Note that in the Boxes program, the color is decided before each rectangle is drawn. In the BarHeights applet, shown in Listing 3.20, 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
3.20

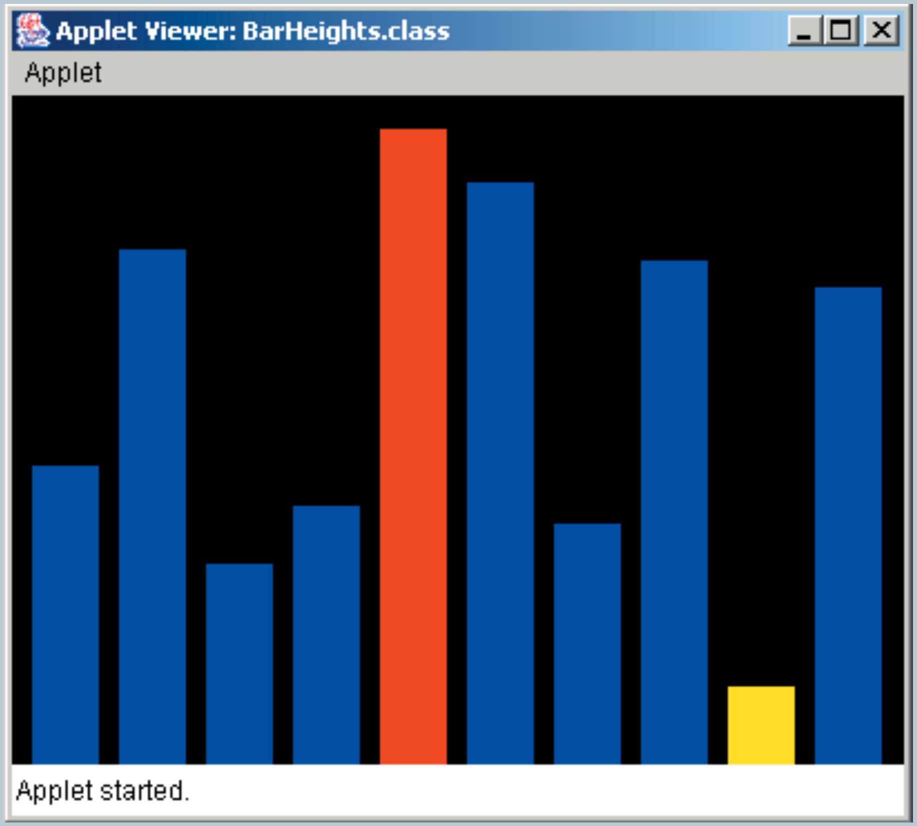
```
//********************************************************************
//  BarHeights.java       Author: Lewis/Loftus
//
//  Demonstrates the use of conditionals and loops to guide drawing.
//********************************************************************

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)
```

listing 3.20 continued

```
            {
               tallX = x;
               tallest = height;
            }

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

            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);
      }
   }
```

listing 3.20 continued

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 appropriate color.

summary of key concepts

- Software requirements specify *what* a program must accomplish.
- A software design specifies *how* a program will accomplish its requirements.
- An algorithm is a step-by-step process for solving a problem, often expressed 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 allow us to control the flow of execution through a method.
- An `if` statement allows a program to choose whether to execute a particular statement.
- Even though the compiler does not care about indentation, proper indentation is important for human readability; it shows the relationship between one statement and another.
- An `if-else` statement allows a program to 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`.
- A `break` statement is usually used at the end of each case alternative of a `switch` statement to jump to the end of the switch.
- A `switch` statement could be implemented as a series of `if-else` statements, but the `switch` is sometimes a more convenient and readable construct.
- Logical operators return a boolean value and are often used to construct sophisticated conditions.
- The relative order of characters in Java is defined by the Unicode character set.
- The `compareTo` method can be used to determine the relative order of strings. It determines lexicographic order, which does not correspond exactly to alphabetical order.

- The prefix and postfix increment and decrement operators have subtle effects on programs because of differences in when they are evaluated.
- A `while` statement allows a program to execute the same statement multiple times.
- We must design our programs carefully to avoid infinite loops. The body of the loop must eventually make the loop condition false.
- A `do` statement executes its loop body at least once.
- 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 in the construction of conditionals and loops?

3.8 What happens if a case in a `switch` does not end with a `break` statement?

3.9 What is a truth table?

3.10 How do we compare strings for equality?

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

3.12 What is an assignment operator?

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

3.14 Compare and contrast a `while` loop and a `do` loop.

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

exercises

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 Write four different program statements that increment the value of an integer variable `total`.

3.3 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");
```

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

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

3.5 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");
```

3.6 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");
```

3.7 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 C.

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

3.8 What output is produced by the following code fragment?

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

3.9 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++;
}
```

3.10 What output is produced by the following code fragment?

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

3.11 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);
```

3.12 Transform the following `while` loop into an equivalent `do` loop (make sure it produces the same output).

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

3.13 Transform the `while` loop from the previous exercise into an equivalent `for` loop (make sure it produces the same output).

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

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

3.15 Write a `while` loop that verifies that the user enters a positive integer value.

3.16 Write a `do` loop that verifies that the user enters an even integer value.

3.17 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.

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

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

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

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

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

3.23 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 modified version of the `Average` program 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 from 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 Modify the solution to the previous project so that the user can evaluate multiple years. Allow the user to terminate the program using an appropriate sentinel value. Validate each input value to ensure 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 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 Modify the `CountWords` program so that it does not include punctuation characters in its character count. *Hint*: This requires changing the set of delimiters used by the `StringTokenizer` class.

3.9 Create a revised version of the `Counter2` program such that the `println` statement comes before the counter increment in the body of the loop. Make sure the program still produces the same output.

3.10 Design and implement an application that prints the first few verses of the traveling song "One Hundred Bottles of Beer." Use a loop such 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

3.11 Design and implement an application that plays the Hi-Lo guessing game with numbers. The program should pick a random number between 1 and 100 (inclusive), then repeatedly prompt 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. Continue accepting guesses until the user guesses correctly or chooses to quit. 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), prompt to determine whether the user wants to play again. Continue playing games until the user chooses to stop.

3.12 Create a modified version of the `PalindromeTester` program so that the spaces, punctuation, and changes in uppercase and lowercase are not considered when determining whether a string is a palindrome. *Hint*: These issues can be handled in several ways. Think carefully about your design.

3.13 Create modified versions of the `Stars` program 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.      *
  *********             **     *********         ***
  ********             ***      ********        *****
  *******             ****       *******       *******
  ******             *****        ******      *********
  *****             ******         *****      *********
  ****             *******          ****       *******
  ***             ********           ***        *****
  **             *********            **         ***
  *             **********             *          *
```

3.14 Design and implement an application that prints a table showing a subset of the Unicode characters and their numeric values. Print five number/character pairs per line, separated by tab characters. Print the table for numeric values from 32 (the space character) to 126 (the ~ character), which corresponds to the printable ASCII subset of the Unicode character set. Compare your output to the table in Appendix C. Unlike the table in Appendix C, the values in your table can increase as they go across a row.

3.15 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 nonvowel characters.

3.16 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 prompt for the user's selection. At that point, the program reveals both choices and prints a statement indicating if the user won, the computer won, or if it was a tie. Continue playing until the user chooses to stop, then print the number of user wins, losses, and ties.

CODEMATE

3.17 Design and implement an application that prints the verses of the song “The Twelve Days of Christmas,” in which each verse adds one line. The first two verses of the song are:

On the 1st day of Christmas my true love gave to me

A partridge in a pear tree.

On the 2nd day of Christmas my true love gave to me

Two turtle doves, and

A partridge in a pear tree.

Use a `switch` statement in a loop to control which lines get printed. *Hint:* Order the cases carefully and avoid the `break` statement. Use a separate `switch` statement to put the appropriate suffix on the day number (1st, 2nd, 3rd, etc.). The final verse of the song involves all 12 days, as follows:

On the 12th day of Christmas, my true love gave to me

Twelve drummers drumming,

Eleven pipers piping,

Ten lords a leaping,

Nine ladies dancing,

Eight maids a milking,

Seven swans a swimming,

Six geese a laying,

Five golden rings,

Four calling birds,

Three French hens,

Two turtle doves, and

A partridge in a pear tree.

3.18 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 an appropriate statement if all three of the numbers are the same, or if any two of the numbers are the same. Continue playing until the user chooses to stop.

3.19 Create a modified version of the `ExamGrades` program to validate the grades entered to make sure they are in the range 0 to 100, inclusive. Print an error message if a grade is not valid, then continue to collect grades. Continue to use the sentinel value to indicate the end of the input, but do not print an error message when the sentinel value is entered. Do not count an invalid grade or include it as part of the running sum.

3.20 Design and implement an applet that draws 20 horizontal, evenly spaced parallel lines of random length.

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

3.22 Design and implement an applet that draws 100 circles of random color and random diameter in random locations. Ensure that in each case the entire circle appears in the visible area of the applet.

3.23 Design and implement an applet that draws 10 concentric circles of random radius.

3.24 Design and implement an applet that draws a brick wall pattern in which each row of bricks is offset from the row above and below it.

3.25 Design and implement an applet that draws a quilt in which a simple pattern is repeated in a grid of squares.

3.26 Design and implement an applet that draws a simple fence with vertical, equally spaced slats backed by two horizontal support boards. Behind the fence show a simple house in the background. Make sure the house is visible between the slats in the fence.

3.27 Design and implement an applet that draws a rainbow. Use tightly spaced concentric arcs to draw each part of the rainbow in a particular color.

3.28 Design and implement an applet that draws 20,000 points in random locations within the visible area of the applet. Make the points on the left half of the applet appear in red and the points on the right half of the applet appear in green. Draw a point by drawing a line with a length of only one pixel.

3.29 Design and implement an applet that draws 10 circles of random radius in random locations. Fill in the largest circle in red.

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 solution in source 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 expressed in pseudocode, a loose combination of English and code-like terms used to capture the basic processing steps informally.

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

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` occurs when the statement inside an `if` or `else` clause is an `if` statement. A nested `if` lets the programmer make a series of decisions. Similarly, 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 multiple things based on the boolean condition.

3.8 If a case does not end with a `break` statement, processing continues into the statements of the next case. We usually want to use `break` statements in order to jump to the end of the `switch`.

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

3.10 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.

3.11 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.

3.12 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.

3.13 An infinite loop is a repetition statement that never terminates. Specifically, the body of the loop never causes the condition to become false.

3.14 A `while` loop evaluates the condition first. If it is true, it executes the loop body. The `do` loop executes the body first and then evaluates the condition. Therefore the body of a `while` loop is executed zero or more times, and the body of a `do` loop is executed one or more times.

3.15 A `for` loop is usually used when we know, or can calculate, how many times we want to iterate through the loop body. A `while` loop handles a more generic situation.