CHAPTER 7

CONTROL STATEMENTS CONTINUED

OBJECTIVES

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

- Construct complex Boolean expressions using the logical operators `&&` (AND), `||` (OR), and `!` (NOT).
- Construct truth tables for Boolean expressions.
- Understand the logic of nested `if` statements and extended `if` statements.
- Test `if` statements in a comprehensive manner.
- Construct nested loops.
- Create appropriate test cases for `if` statements and loops.
- Understand the purpose of assertions, invariants, and loop verification.

Estimated Time: 5 hours

VOCABULARY

Arithmetic overflow
Boundary condition
Combinatorial explosion
Complete code coverage
Equivalence class
Extended `if` statement
Extreme condition
Input assertion
Logical operator
Loop invariant
Loop variant
Nested `if` statement
Nested loop
Output assertion
Quality assurance
Robust
Truth table

This chapter explores more advanced aspects of the control statements introduced in Chapter 4. Topics include logical operators, nested `if` statements, and nested loops. The chapter also describes strategies for testing programs that contain control statements. Programmers try to write programs that are free of logic errors, but they seldom succeed. Consequently, they must test their programs thoroughly before releasing them—and even so, errors will still slip through. Notice that we say "will" instead of "might." Software is so incredibly complex that no significant software product has ever been released free of errors; however, the situation would be much worse if we stopped emphasizing the importance of testing.

7.1 Logical Operators

Java includes three logical operators equivalent in meaning to the English words AND, OR, and NOT. These operators are used in the Boolean expressions that control the behavior of `if`,

`while`, and `for` statements. Before we examine how these operators are used in Java, we review their usage in English. For instance, consider the following sentences:

1. If the sun is shining AND it is 8 a.m. then let's go for a walk; else let's stay home.
2. If the sun is shining OR it is 8 a.m. then let's go for a walk; else let's stay home.
3. If NOT the sun is shining then let's go for a walk; else let's stay home.

The structure of all three sentences is similar, but their meanings are very different. For clarity we have emphasized keywords. In these sentences, the phrases "the sun is shining" and "it is 8 a.m." are operands and the words AND, OR, and NOT are operators. At any particular moment, the value of a condition (true or false) depends on the values of the operands (also true or false) and the operator's meaning. For instance,

- In the first sentence, the operator is AND. Consequently, if both operands are true, the condition as a whole is true. If either or both are false, the condition is false.
- In the second sentence, which uses OR, the condition is false only if both operands are false; otherwise, it is true.
- In the third sentence, the operator NOT has been placed before the operand, as it would be in Java. This looks a little strange in English but is still understandable. If the operand is true, then the NOT operator makes the condition as a whole false.

We summarize these observations in the three parts of Table 7-1. Each part is called a truth table, and it shows how the value of the overall condition depends on the values of the operands. All combinations of values are considered. When there is one operand, there are two possibilities. For two operands, there are four; and for three operands, there are eight possibilities. In general there are 2^n combinations of true and false for n operands.

TABLE 7-1
Truth tables for three example sentences

THE SUN IS SHINING	IT IS 8 A.M.	THE SUN IS SHINING AND IT IS 8 A.M.	ACTION TAKEN
true	true	true	go for a walk
true	false	false	stay at home
false	true	false	stay at home
false	false	false	stay at home
THE SUN IS SHINING	**IT IS 8 A.M.**	**THE SUN IS SHINING OR IT IS 8 A.M.**	**ACTION TAKEN**
true	true	true	go for a walk
true	false	true	go for a walk
false	true	true	go for a walk
false	false	false	stay at home
THE SUN IS SHINING	**NOT THE SUN IS SHINING**	**ACTION TAKEN**	
true	false	stay at home	
false	true	go for a walk	

Dropping the column labeled "action taken," we can combine the information in the three truth tables in Table 7-1 into one table of general rules, as illustrated in Table 7-2. The letters P and Q represent the operands.

TABLE 7-2
General rules for AND, OR, and NOT

P	Q	P AND Q	P OR Q	NOT P
true	true	true	true	false
true	false	false	true	
false	true	false	true	true
false	false	false	false	

Three Operators at Once

Now that we know the rules, it is easy to construct and understand more complex conditions. Consider the following sentences:

A. If (the sun is shining AND it is 8 a.m.) OR (NOT your brother is visiting) then let's go for a walk; else let's stay at home.

We have added parentheses to remove ambiguity. As usual, expressions inside parentheses are evaluated before those that are not. So now when do we go for a walk? The answer is at 8 a.m. on sunny days or when your brother does not visit; however, rearranging the parentheses changes the meaning of the sentence:

B. If the sun is shining AND (it is 8 a.m. OR (NOT your brother is visiting)) then let's go for a walk; else let's stay at home.

Now before we go for a walk, the sun must be shining. In addition, one of two things must be true. Either it is 8 a.m. or your brother is not visiting. It does get a little confusing. Making truth tables for these sentences would make their meanings completely clear.

Java's Logical Operators and Their Precedence

In Java, the operators AND, OR, and NOT are represented by `&&`, `||`, and `!`, respectively. Before writing code that uses these operators, we must consider their precedence, as shown in Table 7-3. Observe that NOT (`!`) has the same high precedence as other unary operators, whereas AND (`&&`) and OR (`||`) have low precedence, with OR below AND.

TABLE 7-3
Positions of the logical and relational operators in the precedence scheme

OPERATION	SYMBOL	PRECEDENCE (FROM HIGHEST TO LOWEST)	ASSOCIATION
Grouping	`( )`	1	Not applicable
Method selector	`.`	2	Left to right
Unary plus	`+`	3	Not applicable
Unary minus	`-`	3	Not applicable
Not	`!`	3	Not applicable
Multiplication	`*`	4	Left to right
Division	`/`	4	Left to right
Remainder or modulus	`%`	4	Left to right
Addition	`+`	5	Left to right
Subtraction	`-`	5	Left to right
Relational operators	`< <= > >= == !=`	6	Not applicable
And	`&&`	8	Left to right
Or	`||`	9	Left to right
Assignment operators	`= *= /= %= += -=`	10	Right to left

A complete table of operator precedence can be found in Appendix C.

Examples Using Logical Operators

Following are some illustrative examples based on the employment practices at ABC Company. The company screens all new employees by making them take two written tests. A program then analyzes the scores and prints a list of jobs for which the applicant is qualified. Following is the relevant code:

```
Scanner reader = new Scanner(System.in);
int score1, score2;
System.out.print("Enter the first test score: ");
score1 = reader.nextInt();
System.out.print("Enter the second test score: ");
score2 = reader.nextInt();
```

```
// Managers must score well (90 or above) on both tests.
if (score1 >= 90 && score2 >= 90)
   System.out.println("Qualified to be a manager");

// Supervisors must score well (90 or above) on just one test
if (score1 >= 90 || score2 >= 90)
   System.out.println("Qualified to be a supervisor");

// Clerical workers must score moderately well on one test
// (70 or above), but not badly (below 50) on either.
if ((score1 >= 70 || score2 >= 70) &&
   !(score1 < 50 || score2 < 50))
   System.out.println("Qualified to be a clerk");
```

Boolean Variables

The complex Boolean expressions in the preceding examples can be simplified by using Boolean variables. A Boolean variable can be true or false and is declared to be of type `boolean`. Now we rewrite the previous examples using Boolean variables:

```
Scanner reader = new Scanner(System.in);
int score1, score2;
boolean bothHigh, atLeastOneHigh, atLeastOneModerate, noLow;
System.out.print("Enter the first test score: ");
score1 = reader.nextInt();
System.out.print("Enter the second test score: ");
score2 = reader.nextInt();

bothHigh           =  (score1 >= 90 && score2 >= 90); // parentheses
atLeastOneHigh     =  (score1 >= 90 || score2 >= 90); // optional
atLeastOneModerate =  (score1 >= 70 || score2 >= 70); // here
noLow              = !(score1 <  50 || score2 <  50);
if (bothHigh)
   System.out.println("Qualified to be a manager");
if (atLeastOneHigh)
   System.out.println("Qualified to be a supervisor");
if (atLeastOneModerate && noLow)
   System.out.println("Qualified to be a clerk");
```

Rewriting Complex Boolean Expressions

A complex `if` statement is sometimes so confusing that it is better rewritten as a series of simpler ones. Here is an example in a mixture of English and Java that we call Javish:

```
if (the sun shines && (you have the time || it is Sunday))
   let's go for a walk;
else
   let's stay home;
```

To rewrite the previous code, we first create a truth table for the complex `if` statement, as shown in Table 7-4.

TABLE 7-4
Truth table for complex `if` statement

P: THE SUN SHINES	Q: YOU HAVE TIME	R: IT IS SUNDAY	P && (Q || R)	ACTION TAKEN
true	true	true	true	walk
true	true	false	true	walk
true	false	true	true	walk
true	false	false	false	stay home
false	true	true	false	stay home
false	true	false	false	stay home
false	false	true	false	stay home
false	false	false	false	stay home

Then implement each line of the truth table with a separate `if` statement involving only `&&` (AND) and `!` (NOT). Applying the technique here yields

```
if ( the sun shines &&  you have time &&  it is Sunday) walk;
if ( the sun shines &&  you have time && !it is Sunday) walk;
if ( the sun shines && !you have time &&  it is Sunday) walk;
if ( the sun shines && !you have time && !it is Sunday) stay home;
if (!the sun shines &&  you have time &&  it is Sunday) stay home;
if (!the sun shines &&  you have time && !it is Sunday) stay home;
if (!the sun shines && !you have time &&  it is Sunday) stay home;
if (!the sun shines && !you have time && !it is Sunday) stay home;
```

In this particular example, the verbosity can be reduced without reintroducing complexity by noticing that the first two `if` statements are equivalent to

```
if ( the sun shines &&  you have time) walk;
```

and the last four are equivalent to

```
if (!the sun shines) stay home;
```

Putting all this together yields

```
if ( the sun shines &&  you have time) walk;
if ( the sun shines && !you have time &&  it is Sunday) walk;
if (the sun shines && !you have time  && !it is Sunday) stay home
if (!the sun shines) stay home;
```

Of course, it is also possible to go in the other direction: that is, combine several `if` statements into a single more complex one, but no matter how we choose to represent complex conditions, truth tables are an essential tool for verifying the accuracy of the result. We can use them anytime we are uncertain about the meaning of the `if` statements we write.

Some Useful Boolean Equivalences

There is often more than one way to write a Boolean expression. For instance, the following pairs of Boolean expressions are equivalent, as truth tables readily confirm:

`!(p || q)`	equivalent to	`!p && !q`
`!(p && q)`	equivalent to	`!p || !q`
`p || (q && r)`	equivalent to	`(p || q) && (p || r)`
`p && (q || r)`	equivalent to	`(p && q) || (p && r)`

Using these equivalences sometimes enables us to rewrite a condition in a more easily understood form. Following is an example in which we display the word "reject" if `x` is not in the interval [3, 5], or alternatively, if `x` is less than 3 or greater than 5:

```
if (!(3 <= x && x <= 5))     System.out.println("reject");
if (!(3 <= x) || !(x <= 5)) System.out.println("reject");
if (x < 3 || x > 5)          System.out.println("reject");
```

Short-Circuit Evaluation

The Java virtual machine sometimes knows the value of a Boolean expression before it has evaluated all of its parts. For instance, in the expression (`p && q`), if `p` is false, then so is the expression, and there is no need to evaluate `q`. Likewise, in the expression (`p || q`), if `p` is true, then so is the expression, and again there is no need to evaluate `q`. This approach, in which evaluation stops as soon as possible, is called short-circuit evaluation. In contrast, some programming languages use complete evaluation, in which all parts of a Boolean expression are always evaluated. These two methods nearly always produce the same results; however, there are times when short-circuit evaluation is advantageous. Consider the following example:

```
Scanner reader = new Scanner(System.in);
int count, sum;
System.out.print("Enter the count: ");
count = reader.nextInt;
System.out.print("Enter the sum: ");
sum = reader.nextInt();

if (count > 0 && sum / count > 10)
   System.out.println("average > 10");
else
   System.out.println("count = 0 or average <= 10");
```

If the user enters 0 for the count, the condition (`count > 0 && sum / count > 10`) contains a potential division by zero; however, because of short-circuit evaluation the division by zero is avoided.

EXERCISE 7.1

1. Fill in the truth values in the following truth table:

P	Q	! ((P || Q) && (P && Q))

2. Assume that A is true and B is false. Write the values of the following expressions:
   - **a.** `A || B`
   - **b.** `A && B`
   - **c.** `A && ! B`
   - **d.** `! (A || B)`

3. Construct truth tables for the expressions listed under the heading "Some Useful Boolean Equivalences" in this section to show that they are equivalent.

4. List the logical operators in the order in which each one would be evaluated at run time.

5. Construct a Boolean expression that tests whether the value of variable x is within the range specified by the variables min (the smallest) and max (the largest).

Case Study 1: Compute Weekly Pay

We illustrate the use of logical operators by writing a program to compute weekly pay.

Request

Write a program to compute the weekly pay of hourly employees.

Analysis

Employees are paid at a base rate for the first 40 hours they work each week. Hours over 40 are paid at an overtime rate equal to twice the base rate. An exception is made for part-time employees, who are always paid at the regular rate, no matter how many hours they work. The hourly rate is in the range $6.75 to $30.50 and hours worked in the range 1 to 60. We use a type attribute to distinguish between full-time (type 1) and part-time (type 2) employees. Figure 7-1 shows the user interface.

We would also like the program to be ***robust***—that is, to handle invalid inputs without crashing or producing meaningless results. The easiest and best way to achieve this end is to check data values as soon as they are entered and reject those that are invalid. The user is then given another opportunity to enter a correct value, as shown in Figure 7-2.

FIGURE 7-1
Interface for the compute weekly pay program

```
Enter employee data
  Name (or blank to quit): Susan Jones
  Type (1 or 2): 1
  Hourly rate (between 6.75 and 30.50, inclusive): 10.50
  Hours worked (between 1 and 60, inclusive): 50
  The weekly pay for Susan Jones is $630.0
Enter employee data
  Name (or blank to quit): Bill Smith
  Type (1 or 2): 2
  Hourly rate (between 6.75 and 30.50, inclusive): 15.00
  Hours worked (between 1 and 60, inclusive): 60
  The weekly pay for Bill Smith is $900.0
Enter employee data
  Name (or blank to quit):
```

FIGURE 7-2
How the program responds to invalid inputs

```
Enter employee data
  Name (or blank to quit): Patricia Nelson
  Type (1 or 2): 3
  Type (1 or 2): 0
  Type (1 or 2): 1
  Hourly rate (between 6.75 and 30.50, inclusive): 99.00
  Hourly rate (between 6.75 and 30.50, inclusive): 3.75
  Hourly rate (between 6.75 and 30.50, inclusive): 20.89
  Hours worked (between 1 and 60, inclusive): 100
  Hours worked (between 1 and 60, inclusive): 25
  The weekly pay for Patricia Nelson is $522.25
```

Following the approach introduced in Chapter 6, we divide the work of the application into two classes: a user interface class (`PayrollSystemApp`) and an employee class (`Employee`). The `Employee` class has four instance variables:

- Name
- Type
- Rate
- Hours

The `Employee` class also has three responsibilities:

- Provide information about data validation rules (`getNameRules`, `getTypeRules`, etc).
- Set instance variables provided the values are valid (return `true` if valid and `false` otherwise).
- Get the name and weekly pay.

Figure 7-3 summarizes these points. The user interface class has the usual structure and has the single method `main`, so we omit a class summary.

FIGURE 7-3
Summary of the `Employee` class

Class:
`Employee`

Private Instance Variables:
```
String name
int type
double rate
int hours
```

Public Methods:
```
constructor
String getNameRules()
String getTypeRules()
String getRateRules()
String getHoursRules()
boolean setName(String nm)
boolean setType(int tp)
boolean setRate(double rt)
boolean setHours(int hrs)
String getName()
double getPay()
```

Design

Following is the pseudocode for the user interface:

```
while (true){
   read name, break if blank else set the employee name
   read type until valid and set the employee type
   read rate until valid and set the employee rate
   read hours until valid and set the employee hours
   ask the employee for name and pay and print these
}
```

An employee object computes pay as follows:

```
if (hours <= 40 || type == 2)
   pay = rate * hours;
else
   pay = rate * 40 +  rate * 2 * (hours - 40);
```

Implementation

The implementation uses two new `String` methods, `trim` and `equals`. When a `trim()` message is sent to a string, a new string that contains no leading or trailing spaces is returned:

```
String inputName, trimmedName;
System.out.print("Enter the name: ")
inputName = reader.nextLine;
trimmedName = inputName.trim();
```

The `equals` method is used to determine if two string objects contain equal strings. This is in contrast to the == operator, which determines if two string variables refer to the same object. Following is an illustration:

```
String a, b;
a = "cat";
System.out.print("What do you call a small domestic feline? ");
b = reader.nextLine;

if (a == b)
   System.out.println("a and b reference the same object.");
else
   System.out.println("a and b reference different objects.");

if (a.equals(b))
   System.out.println
   ("a and b reference objects that contain equal strings.");
else
   System.out.println
   ("a and b reference objects that contain unequal strings");
```

If the user enters "cat" when this code is run, the output is

```
a and b reference different objects.
a and b reference objects that contain equal strings.
```

We now return to the Case Study program and present the implementation, which is complicated by all the error checking we have decided to do. The user interface class restricts itself to interacting with the user, and the `Employee` class controls data validation and computation of pay:

```
/* Case Study 7.1: PayrollSystemApp.java
1. Request employee name, type, pay rate, and hours.
2. Print employee name and pay.
3. Repeat until the name is blank.*/

import java.util.Scanner;

public class PayrollSystemApp{

   public static void main (String [] args) {
      Scanner reader = new Scanner(System.in);
      Employee emp;    // employee
      String name;     //   name
      int    type;     //   type
      double rate;     //   hourly pay rate
      int    hours;    //   hours worked
      String prompt;   // user prompt;

      while (true){

         // Get the name and break if blank
         System.out.println("Enter employee data");
         System.out.print("  Name (or blank to quit): ");
         name = reader.nextLine();
```

```
            name = name.trim(); // Trim off leading and trailing spaces
            if (name.length() == 0) break;
            emp = new Employee();
            emp.setName(name);

            // Get the type until valid
            while (true){
               prompt = "  Type (" + emp.getTypeRules() + "): ";
               System.out.print(prompt);
               type = reader.nextInt();
               if (emp.setType(type)) break;
            }

            // Get the hourly pay rate until valid
            while (true){
               prompt = "  Hourly rate (" + emp.getRateRules() + "): ";
               System.out.print(prompt);
               rate = reader.nextDouble();
               if (emp.setRate(rate)) break;
            }

            // Get the hours worked until valid
            //   To illustrate the possibilities we compress
            //   into a hard-to-read set of statements.
            System.out.print("Hours worked (" +
                             emp.getHoursRules() + "): ");
            while (!emp.setHours(reader.nextInt()))
                   System.out.print("Hours worked (" +
                                    emp.getHoursRules() + "): ");

            // Consume the trailing newline
            reader.nextLine();

            // Print the name and pay
            System.out.println("  The weekly pay for " + emp.getName() +
                               " is $" + emp.getPay());
         }
      }
   }
```

```
/*  Employee.java
1. Instance variables: name, type, rate, hours
2. Methods to
   get data validation rules
   set instance variables if data are valid
   get name and pay */

public class Employee{

   // Private Instance Variables:
   private String name;
```

```
private int type;
private double rate;
private int hours;

// Public Methods:
public Employee(){
   name = "";
   type = 0;
   rate = 0;
   hours = 0;
}

public String getNameRules(){
   return "nonblank";
}

  public String getTypeRules(){
   return "1 or 2";
}

public String getRateRules(){
   return "between 6.75 and 30.50, inclusive";
}

public String getHoursRules(){
   return "between 1 and 60, inclusive";
}

public boolean setName(String nm){
   if (nm.equals(""))
      return false;
   else{
      name = nm;
      return true;
   }
}

public boolean setType(int tp){
   if (tp != 1 && tp != 2)
      return false;
   else{
      type = tp;
      return true;
   }
}

public boolean setRate(double rt){
   if (!(6.75 <= rt && rt <= 30.50))
      return false;
   else{
      rate = rt;
      return true;
   }
}
```

```
    public boolean setHours(int hrs){
       if (!(1 <= hrs && hrs <= 60))
          return false;
       else{
          hours = hrs;
          return true;
       }
    }

    public String getName(){
       return name;
    }

    public double getPay(){
       double pay;
       if (hours <= 40 || type == 2)
          pay = rate * hours;
       else
          pay = rate * 40 +  rate * 2 * (hours - 40);
       return pay;
    }
}
```

7.2 Testing `if` Statements

Quality assurance is the ongoing process of making sure that a software product is developed to the highest standards possible, subject to the ever-present constraints of time and money. As we learned in Chapter 1, faults are fixed most inexpensively early in the development life cycle; however, no matter how much care is taken at every stage during a program's development, eventually the program must be run against well-designed test data. Such data should exercise a program as thoroughly as possible. At a minimum, the test data should try to achieve ***complete code coverage***, which means that every line in a program is executed at least once. Unfortunately, this is not the same thing as testing all possible logical paths through a program, which would provide a more thorough test, but also might require considerably more test data.

We now design test data for the preceding case study. Because the program is so simple, the test data will provide complete code coverage and test all possible logical paths through the program. Varying the hourly rate has no particular significance in this problem, so we use an hourly rate of $10 for all the tests.

First, we test with an employee type of 1 and hours worked equal to 30 and 50 hours. Because we must compare the program's output with the expected results, we have chosen numbers for which it is easy to perform the calculations by hand. Having tested the program for the input 30 hours, we feel no need to test it for 29 or 31 hours because we realize that exactly the same code is executed in all three cases. Likewise, we do not feel compelled to test the program for 49 and 51 hours. All the sets of test data that exercise a program in the same manner are said to belong to the same ***equivalence class***, which means they are equivalent from the perspective of testing the same paths through the program. When the employee type is 1, test data for the payroll program fall into just two equivalence classes: hours between 0 and 40 and hours greater than 40.

The test data should also include cases that assess a program's behavior under ***boundary conditions***—that is, on or near the boundaries between equivalence classes. It is common for

programs to fail at these points. For the payroll program, this requirement means testing with hours equal to 39, 40, and 41.

We should test under ***extreme conditions***—that is, with data at the limits of validity. For this we choose hours worked equal to 0 and 168 hours.

Testing with an employee type of 2 is much simpler because the number of hours does not matter, but just to be on the safe side, we test with the hours equal to 30 and 50.

Finally, we must test the data validation rules. We need to enter values that are valid and invalid, and we must test the boundary values between the two. This suggests that we test using

- Type equal to 0, 1, 2, and 3
- Hourly rate equal to 6.74, 6.75, 10, 30.50, and 30.51
- Hours worked equal to 0, 1, 30, 60, and 61

Table 7-5 summarizes our planned tests. If from this discussion you draw the conclusion that testing is a lot of work, you are correct. Many software companies spend as much money on testing as they do on analysis, design, and implementation combined. The programs in this book, however, are fairly short and simple, and testing takes only a moderate amount of time.

For a discussion of an approach that puts testing at the center of software development, see Kent Beck, *Extreme Programming Explained: Embrace Change, Second Edition* (Boston: Addison-Wesley, 2004).

TABLE 7-5
Test data for the payroll program

TYPE OF TEST	DATA USED
Code coverage	employee type: 1 hourly rate: 10 hours worked: 30 and 50
Boundary conditions	employee type: 1 hourly rate: 10 hours worked: 39, 40, and 41
Extreme conditions	employee type: 1 hourly rate: 10 hours worked: 0 and 168
Tests when the employee type is 2	employee type: 2 hourly rate: 10 hours worked: 30 and 50
Data validation rules	type: 0, 1, 2, and 3 hourly rate: 6.74, 6.75, 10, 30.50, and 30.51 hours worked: 0, 1, 30, 60, and 61

EXERCISE 7.2

1. Describe appropriate test data for the following code segments:

   a.

```
if (number > 0)
   <action 1>
else
   <action 2>
```

   b.

```
 if (0 < number && 100 > number)
   <action 1>
else
   <action 2>
```

2. What happens when we provide complete code coverage of a program?

3. What is an equivalence class? Give an example.

4. What are boundary conditions? Give an example.

5. What are extreme conditions? Give an example.

6. Suppose a teacher uses grades from 0 to 100 and wants to discount all grades below 60 in her records. Discuss the equivalence classes, boundary conditions, and extreme conditions used to test a program that processes this information.

7.3 Nested `if` Statements

A program's logic is often complex. Logical operators (`&&`, `||`, and `!`) provide one mechanism for dealing with this complexity. Nested `if` statements offer an alternative. Following is an everyday example of nested `if`s written in Javish:

```
if (the time is after 7 PM){
   if (you have a book)
      read the book;
   else
      watch TV;
}else
   go for a walk;
```

Technology Careers

ARTIFICIAL INTELLIGENCE, ROBOTS, AND SOFTBOTS

Computers not only calculate results but also respond to conditions in their environment and take the appropriate actions. This additional capability forms the basis of a branch of computer science known as artificial intelligence, or AI. AI programmers attempt to construct computational models of intelligent human behavior. These tasks involve, among many others, interacting in English or other natural languages, recognizing objects in the environment, reasoning, creating and carrying out plans of action, and pruning irrelevant information from a sea of detail.

There are many ways to construct AI models. One way is to view intelligent behavior as patterns of production rules. Each rule contains a set of conditions and a set of actions. In this model, an intelligent agent, either a computer or a human being, compares conditions in its environment to the conditions of all of its rules. Those rules with matching conditions are scheduled to fire—meaning that their actions are triggered—according to a higher-level scheme of rules. The set of rules is either hand-coded by the AI programmer or "learned" by using a special program known as a neural net.

Among other things, AI systems have been used to control *robots*. These robots perform mundane tasks such as assembling cars.

AI systems also are embedded in software agents known as *softbots*. For example, softbots exist to filter information from electronic mail systems, to schedule appointments, and to search the World Wide Web for information.

For a detailed discussion of robots and softbots, see Rodney Brooks, "Intelligence Without Representation," in *Mind Design II*, ed. John Haugeland (Cambridge, MA: MIT Press, 1997), and Patti Maes, "Agents That Reduce Work and Information Overload," *Communications of the ACM*, Volume 37, No. 7 (July 1994): 30–40.

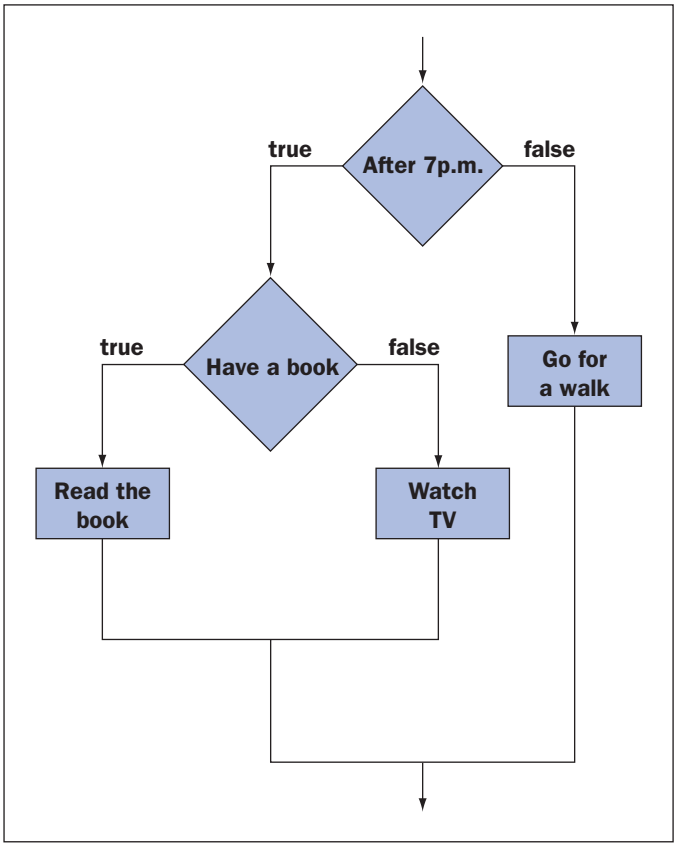
Although this code is not complicated, it is a little difficult to determine exactly what it means without the aid of the truth table illustrated in Table 7-6.

TABLE 7-6
Truth table for reading a book, watching TV, or going for a walk

AFTER 7 P.M.	HAVE A BOOK	ACTION TAKEN
true	true	read book
true	false	watch TV
false	true	walk
false	false	walk

Having made the table, we are certain that we understand the code correctly. Of course, it is better to make the table first and then write code to match. As a substitute for a truth table, we can draw a flowchart, as shown in Figure 7-4. Again, it is better to draw the flowchart before writing the code. Truth tables and flowcharts are useful design tools whenever we must deal with complex logic.

FIGURE 7-4
Flowchart for reading a book, watching TV, or going for a walk



Determine a Student's Grade

Following is a second example of nested `if` statements. The code determines a student's grade based on his test average:

```
System.out.print("Enter the test average: ");
testAverage = reader.nextInt();
if (testAverage >= 90)
   System.out.println("grade is A");
else{
   if (testAverage >= 80)
      System.out.println("grade is B");
   else{
      if (testAverage >= 70)
         System.out.println("grade is C");
      else{
         if (testAverage >= 60)
            System.out.println("grade is D");
         else{
            System.out.println("grade is F");
         }
      }
   }
}
```

Because in the absence of braces an else is associated with the immediately preceding `if`, we can drop the braces and rewrite the code as follows:

```
System.out.print("Enter the test average: ");
testAverage = reader.nextInt();
if (testAverage >= 90)
   System.out.println("grade is A");
else
   if (testAverage >= 80)
      System.out.println("grade is B");
   else
      if (testAverage >= 70)
         System.out.println("grade is C");
      else
         if (testAverage >= 60)
            System.out.println("grade is D");
         else
            System.out.println("grade is F");
```

or after changing the indentation slightly as follows:

```
System.out.print("Enter the test average: ");
testAverage = reader.nextInt();
if (testAverage >= 90)
   System.out.println("grade is A");
else if (testAverage >= 80)
   System.out.println("grade is B");
else if (testAverage >= 70)
```

```
      System.out.println("grade is C");
   else if (testAverage >= 60)
      System.out.println("grade is D");
   else
      System.out.println("grade is F");
```

This last format is very common and is used whenever a variable is compared to a sequence of threshold values. This form of the `if` statement is sometimes called an extended `if` statement or a multiway `if` statement, as compared to the two-way and one-way `if` statements we have seen earlier.

In Java, the `switch` statement provides an alternative to the extended `if` statement. See Appendix B for details.

EXERCISE 7.3

1. Construct a truth table that shows all the possible paths through the following nested `if` statement:

```
if (the time is before noon)
   if (the day is Monday)
      take the computer science quiz
   else
      go to gym class
else
   throw a Frisbee in the quad
```

2. What is the difference between a nested `if` statement and a multiway `if` statement?

7.4 Logical Errors in Nested `if` statements

It is easy to make logical errors when writing nested `if` statements. In this section, we illustrate several fairly typical mistakes.

Misplaced Braces

One of the most common mistakes involves misplaced braces. Consider how repositioning a brace affects the following code:

```
// Version 1
if (the weather is wet){
   if (you have an umbrella)
      walk;
   else
      run;
}

// Version 2
if (the weather is wet){
   if (you have an umbrella)
      walk;
}else
   run;
```

To demonstrate the differences between the two versions, we construct a truth table—as shown in Table 7-7.

TABLE 7-7
Truth table for version 1 and version 2

THE WEATHER IS WET	YOU HAVE AN UMBRELLA	VERSION 1 OUTCOME	VERSION 2 OUTCOME
true	true	walk	walk
true	false	run	none
false	true	none	run
false	false	none	run

The truth table shows exactly how different the two versions are.

Removing the Braces

This example raises an interesting question. What happens if the braces are removed? In such situations, Java pairs the `else` with the closest preceding `if`. Thus

```
if (the weather is wet)
   if (you have an umbrella)
      walk;
   else
      run;
```

Remember that indentation is just a stylistic convention intended to improve the readability of code and means nothing to the compiler. Consequently, reformatting the above code as follows does not change its meaning but will almost certainly mislead the unwary programmer:

```
if (the weather is wet)
   if (you have an umbrella)
      walk;
else
   run;
```

Introducing a Syntax Error

Now we consider a final variation:

```
if (the weather is wet)
   if (you have an umbrella)
      open umbrella;
      walk;
   else
      run;
```

This contains a compile-time error. Can you spot it? The second `if` is followed by more than one statement, so braces are required:

```
if (the weather is wet)
   if (you have an umbrella){
      open umbrella;
      walk;
   }else
      run;
```

Remembering that it is better to overuse than to underuse braces, we could rewrite the code as follows:

```
if (the weather is wet){
   if (you have an umbrella){
      open umbrella;
      walk;
   }else{
      run;
   }
}
```

Computation of Sales Commissions

We now attempt to compute a salesperson's commission and introduce a logical error in the process. Commissions are supposed to be computed as follows:

- 10% if sales are greater than or equal to $5000
- 20% if sales are greater than or equal to $10,000

Following is our first attempt at writing the corresponding code:

```
if (sales >= 5000)
   commission = sales * 0.1;       // line a
else if (sales >= 10000)
   commission = sales * 0.2;       // line b
```

To determine if the code works correctly, we check it against representative values for the sales, namely, sales that are less than $5000, equal to $5000, between $5000 and $10,000, equal to $10,000, and greater than $10,000. As we can see from Table 7-8, the code is not working correctly.

TABLE 7-8
Calculation of commissions for various sales levels

VALUE OF SALES	LINES EXECUTED	VALIDITY
1,000	neither line a nor line b	correct
5,000	line a	correct
7,000	line a	correct
10,000	line a	incorrect
12,000	line a	incorrect

Corrected Computation of Sales Commissions

After a little reflection, we realize that the conditions are in the wrong order. Here is the corrected code:

```
if (sales >= 10000)
   commission = sales * 0.2;        // line b
else if (sales >= 5000)
   commission = sales * 0.1;        // line a
```

Table 7-9 confirms that the code now works correctly.

TABLE 7-9
Corrected calculation of commissions for various sales levels

VALUE OF SALES	LINES EXECUTED	VALIDITY
1,000	neither line a nor line b	correct
5,000	line a	correct
7,000	line a	correct
10,000	line b	correct
12,000	line b	correct

Avoiding Nested `if` statements

Sometimes getting rid of nested `if`s is the best way to avoid logical errors. This is easily done by rewriting nested `if`s as a sequence of independent `if` statements. For example, consider the following code for computing sales commissions:

```
if (5000 <= sales && sales < 10000)
   commission = sales * 0.1;
if (10000 <= sales)
   commission = sales * 0.2;
```

And here is another example involving the calculation of student grades:

```
if (90 <= average                  ) grade is A;
if (80 <= average && average < 90) grade is B;
if (70 <= average && average < 80) grade is C;
if (60 <= average && average < 70) grade is D;
if (                 average < 60) grade is F;
```

The first question people usually ask when confronted with these alternatives is, "Which is faster?", by which they mean, "Which will execute most rapidly?" In nearly all situations the difference in speed is negligible, so a much better question is, "Which is easier to write and maintain correctly?" There is no hard-and-fast answer to this question, but you should always consider it when writing complex code.

EXERCISE 7.4

1. A tax table provides rates for computing tax based on incomes up to and including a given amount. For example, income above $20,000 up to and including $50,000 is taxed at 18%. Find the logic errors in the following code that determines the tax rate for a given income:

```
if (income > 10000)
   rate = 0.10;
else if (income > 20000)
   rate = 0.18;
else if (income > 50000)
   rate = 0.40;
else
   rate = 0.0;
```

2. Write a correct code segment for the problem in Question 1.

7.5 Nested Loops

There are many programming situations in which a loop is placed within another loop—this is called a ***nested loop***. The first case study in this chapter provided an example of this. We now consider another. In Chapter 4, we showed how to determine if a number is prime. The code involved a `for` loop, and by nesting this `for` loop inside another, we can compute all the primes between two limits. The outside loop feeds a sequence of numbers to the inside loop. Following is the code:

```
System.out.print("Enter the lower limit: ");
lower = reader.nextInt();
System.out.print("Enter the upper limit: ");
upper = reader.nextInt();
for (n = lower; n <= upper; n++){
   innerLimit = (int)Math.sqrt (n);
   for (d = 2; d <= innerLimit; d++){
      if (n % d == 0)
         break;
   }
   if (d > innerLimit)
      System.out.println (n + " is prime");
}
```

Following is the output when the user enters 55 and 75:

```
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
```

If the user wants to enter repeated pairs of limits, we enclose the code in yet another loop:

```
System.out.print("Enter the lower limit or -1 to quit: ");
lower = reader.nextInt();
```

```
while (lower != -1){
   System.out.print("Enter the upper limit: ");
   upper = reader.nextInt();
   for (n = lower; n <= upper; n++){
      innerLimit = (int)Math.sqrt (n);
      for (d = 2; d <= innerLimit; d++){
         if (n % d == 0)
            break;
      }
      if (d > innerLimit)
         System.out.println (n + " is prime");
   }
   System.out.print("Enter the lower limit or -1 to quit: ");
   lower = reader.nextInt();
}
```

Note of Interest

RELIABILITY OF COMPUTER SYSTEMS

The next time you step onto an airplane or lie down inside an MRI machine, you might ask yourself about the quality of the software system that helps to run them. There are several measures of software quality, such as readability, maintainability, correctness, and robustness. But perhaps the most important measure is ***reliability***. Reliability should not be confused with correctness. Software is correct if its design and implementation are consistent with its specifications. That means that the software actually does what it is supposed to do, as described in what we have called analysis. However, software can be correct in this sense yet still be unreliable.

During the analysis phase of software development, we construct a model of what the user wants the software to do, and from this model we build a model of what the software will do. Our design and implementation may reflect this second model correctly, but the software may still be unreliable. It is unreliable if we have built the wrong models during analysis—that is, if we have misunderstood the user's request (have the wrong model of the user) or we have built a model of the software that does not do what we correctly have understood the user to require.

For example, several decades ago, the Navy contracted with a software firm to build a software system to detect the movements of missiles. The software worked just fine in detecting missiles but was thrown off by the presence of the moon in certain cases.

There have been many reports of software unreliability in commercial software installations as well. One of the more tragic cases is that of the x-ray machine Therac-25, which killed several patients a few years ago.

A classic discussion of software reliability in military applications can be found in Alan Borning, "Computer System Reliability and Nuclear War," *Communications of the ACM*, Volume 30, Number 2 (February 1987):112–131. Almost every textbook on computer ethics has case studies on computer reliability in commercial applications. A good place to start is Sara Baase, *A Gift of Fire, Second Edition* (Upper Saddle River, NJ: Prentice Hall, 2002), Chapter 4.

EXERCISE 7.5

1. Write the outputs of the following code segments:

a.

```
for (int i = 1; i <= 3; i++)
   for (int j = 1; j <= 3; j++)
      System.out.print(j + " ");
```

b.

```
for (int i = 1; i <= 3; i++){
   for (int j = 1; j <= 3; j++)
      System.out.print(j + " ");
   System.out.println("");
}
```

2. Write code segments that solve the following problems:

a. Output the numbers 1 to 25 in consecutive order, using five rows of five numbers each.

b. Output five rows of five numbers. Each number is the sum of its row position and column position. The position of the first number is (1, 1).

7.6 Testing Loops

The presence of looping statements in a program increases the challenge of designing good test data. Frequently, loops do not iterate some fixed number of times, but instead iterate zero, one, or more than one time depending on a program's inputs. When designing test data, we want to cover all three possibilities. To illustrate, we develop test data for the print divisors program presented in Chapter 4. First, let's look at the code again:

```
// Display the proper divisors of a number
System.out.print("Enter a positive integer: ");
int n = reader.nextInt();
int limit = n / 2;
for (int d = 2; d <= limit; d++){
   if (n % d == 0)
      System.out.print (d + " ");
}
```

By analyzing the code, we conclude that if n equals 0, 1, 2, or 3, the limit is less than 2, and the loop is never entered. If n equals 4 or 5, the loop is entered once. If n is greater than 5, the loop is entered multiple times. All this suggests the test data shown in Table 7-10. After testing the program with this data, we feel reasonably confident that it works correctly.

TABLE 7-10
Test data for the count divisors program

TYPE OF TEST	DATA USED
No iterations	0, 1, 2, and 3
One iteration	4 and 5
Multiple iterations for a number with divisors	24
Multiple iterations for a number without divisors	29

Combinatorial Explosion

The surprisingly large amount of testing needed to validate even a small program suggests an interesting question. Suppose a program is composed of three parts and that it takes five tests to verify each part independently. Then how many tests does it take to verify the program as a whole? In the unlikely event that the three parts are independent of each other and utilize the same five sets of test data, then five tests suffice. However, it is far more likely that the behavior of each part affects the other two and also that the parts have differing test requirements. Then all possible combinations of tests should be tried, that is, 5*5*5 or 125. We call this multiplicative growth in test cases a combinatorial explosion, and it pretty much guarantees the impossibility of exhaustively testing large complex programs; however, programmers still must do their best to test their programs intelligently and well.

Robust Programs

So far, we have focused on showing that a program that uses loops produces correct results when provided with valid inputs, but, surprisingly, that is not good enough. As we learned when we were testing programs with `if` statements earlier in this chapter, we also should consider how a program behaves when confronted with invalid data. After all, users frequently make mistakes or do not fully understand a program's data entry requirements. As we have learned, a program that tolerates errors in user inputs and recovers gracefully is robust. The best and easiest way to write robust programs is to check user inputs immediately on entry and reject those that are invalid. We illustrate this technique in the next case study. At this stage, there are limits to how thoroughly we can check inputs, so in the case study, we merely make sure that inputs fall in the range specified by the prompt.

EXERCISE 7.6

1. Describe appropriate test data for the following code segments:

a.

```
while (number > 0)
   <action>
```

b.

```
while (0 < number && 100 > number)
   <action>
```

EXERCISE 7.6 Continued

2. Design test data for Project 4-6 (in Chapter 4).
3. What would be reasonable test data for a loop that does not execute a fixed number of times?
4. What is a robust program? Give an example.

Case Study 2: Fibonacci Numbers

There is a famous sequence of numbers that occurs frequently in nature. In 1202, the Italian mathematician Leonardo Fibonacci presented the following problem concerning the breeding of rabbits. He assumed somewhat unrealistically that

1. Each pair of rabbits in a population produces a new pair of rabbits each month.
2. Rabbits become fertile one month after birth.
3. Rabbits do not die.

He then considered how rapidly the rabbit population would grow on a monthly basis when starting with a single pair of newborn rabbits.

To answer the question, we proceed one month at a time:

- At the beginning of month 1, there is one pair of rabbits (total = 1 pair).
- At the beginning of month 2, our initial pair of rabbits, A, will have just reached sexual maturity, so there will be no offspring (total = 1 pair).
- At the beginning of month 3, pair A will have given birth to pair B (total = 2 pair).
- At the beginning of month 4, pair A will have given birth to pair C and pair B will be sexually mature (total = 3 pair).
- At the beginning of month 5, pairs A and B will have given birth to pairs D and E, while pair C will have reached sexual maturity (total = 5 pair).
- And so on.

If we continue in this way, we obtain the following sequence of numbers

1 1 2 3 5 8 13 21 34 55 89 144 233 ...

called the *Fibonacci numbers*. Notice that each number, after the first two, is the sum of its two predecessors. Referring back to the rabbits, see if you can demonstrate why this should be the case. Although the sequence of numbers is easy to construct, the formula for calculating the *n*th Fibonacci number is quite complex and gives rise to the following program request and our proposed solution.

Request

Write a program that can compute the *n*th Fibonacci number on demand, where *n* is a positive integer.

Analysis, Design, and Implementation

The user input should be a positive integer or -1 to quit. Other integer inputs are rejected. The proposed user interface is shown in Figure 7-5.

FIGURE 7-5
Interface for the Fibonacci program

```
Enter a positive integer or -1 to quit: 8
Fibonacci number 8 is 21
```

Here is the code:

```
// Case Study 7.2: Display the nth Fibonacci number

import java.util.Scanner;

public class Fibonacci {

   public static void main (String [] args) {
      Scanner reader = new Scanner(System.in);
      int n;          // The number entered by the user
      int fib;        // The nth Fibonacci number
      int a,b,count;  // Variables that facilitate the computation

      while (true){

         // Ask the user for the next input
         System.out.print("Enter a positive integer or -1 to quit: ");
         n = reader.nextInt();
         if (n == -1) break;
         else if (n >= 1){

            // Calculate the nth Fibonacci number
            fib = 1;             // Takes care of case n = 1 or 2
            a = 1;
            b = 1;
            count = 3;
            while (count <= n){  // Takes care of case n >= 3
               fib = a + b;      // Point p. Referred to later.
               a = b;
               b = fib;
               count = count + 1;
            }

            // Print the nth Fibonacci number
            System.out.println ("Fibonacci number " + n + " is " + fib);
         }  // end else
      }     // end while
   }        // end main
}
```

Loop Analysis

The loop in the Fibonacci program is not obvious at first glance, so to clarify what is happening, we construct Table 7-11. This table traces the changes to key variables on each pass through the loop.

TABLE 7-11
Changes to key variables on each pass through the loop

COUNT AT POINT P	A AT POINT P	B AT POINT P	FIBONACCI NUMBER AT POINT P
3	1	1	2
4	1	2	3
5	2	3	5
6	3	5	8
...	...	...	...
n	(*n* - 2)th Fibonacci number	(*n* - 1)th Fibonacci number	*n*th Fibonacci number

Test Data

We complete the case study by developing suitable test data:

- To make sure the program is robust, we try the following sequence of inputs for *n*: -3, 0, 1, 2, –1.
- To make sure the computation is correct when the second inner loop is not entered, we try *n* equal to 1 and 2.
- To make sure the computation is correct when the second inner loop is entered one or more times, we let *n* equal 3 and 6.

Because all these tests were successful, we can hardly be blamed for thinking that our program works perfectly; however, it contains a completely unexpected problem. When *n* equals 80, the program returns the value –285,007,387. The problem is due to ***arithmetic overflow***. In Java and most other programming languages, integers have a limited range (see Chapter 3) and exceeding that range leads to strange results. Adding one to the most positive integer in the range yields the most negative integer, whereas subtracting one from the most negative yields the most positive. Welcome to the strange world of computer arithmetic, where bizarre behavior is always lurking to trip the unwary. To detect the problem automatically, we could include extra lines of code that test for an unexpected switch to a negative value. A somewhat similar problem cost the French space program half a billion dollars and a great deal of embarrassment when a computer-guided rocket and its payload exploded shortly after takeoff.

7.7 Loop Verification

Loop verification is the process of guaranteeing that a loop performs its intended task, independently of testing. Some work has been done on constructing formal proofs to determine if

loops are "correct." We now examine a modified version of loop verification; a complete treatment of the issue will be the topic of subsequent course work.

The `assert` Statement

Java includes an `assert` statement that allows the programmer to evaluate a Boolean expression and halt the program with an error message if the expression's value is false. If its value is true, the program continues execution. Here are some examples of `assert` statements:

```
assert x != 0;                  // Halt if x is 0
assert x >= 0 && x <= MAX;      // Halt if x is not in this range
assert x % 2 == 0;              // Halt if x is not even
```

The general form is `assert <Boolean expression>`. Note that parentheses are omitted. At run time, if the Boolean expression evaluates to false, the JVM will halt with an error message. To enable this mechanism when running the program, use the command line

```
java -enableassertions AJavaProgram
```

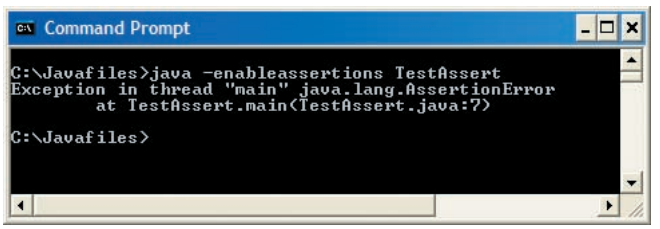
Figure 7-6 shows the output of the following short program that includes an `assert` statement:

```
// Example 7.1: Assert that x != 0

public class TestAssert{

   public static void main(String[] args){
      int x = 0;
      assert x != 0;
   }
}
```

FIGURE 7-6
The failure of an `assert` statement



Running a program with `assert` enabled will slow it down, but we do this only during development and testing.

Assertions with Loops

Input assertions state what should be true before a loop is entered. ***Output assertions*** state what should be true when the loop is exited.

To illustrate input and output assertions, we consider the mathematical problem of summing the proper divisors of a positive integer. For example, suppose we have the integers shown in Table 7-12.

TABLE 7-12
The sums of the proper divisors of some integers

INTEGER	PROPER DIVISORS	SUM
6	1, 2, 3	6
9	1, 3	4
12	1, 2, 3, 4, 6	16

The following loop performs this task:

```
divisorSum = 0;
for (trialDivisor = 1; trialDivisor <= num / 2; ++trialDivisor)
   if (num % trialDivisor == 0)
      divisorSum = divisorSum + trialDivisor;
```

Input assertions for this loop are

```
1. num is a positive integer.
2. divisorSum == 0.
```

An output assertion is

```
divisorSum is the sum of all proper divisors of num.
```

When these are placed with the previous code, we have

```
divisorSum = 0;
assert num > 0 && divisorSum == 0;
for (trialDivisor = 1; trialDivisor <= num / 2; ++trialDivisor)
   if (num % trialDivisor == 0)
      divisorSum = divisorSum + trialDivisor;
// Output assertion: divisorSum is the sum of all proper divisors of num.
```

Note that we pass the Boolean expression to the `assert` mechanism of Java so that the JVM actually establishes the truth of that assertion. However, we cannot do this with the output assertion because the sum of all the proper divisors of a number is just what we are computing in the loop!

Invariant and Variant Assertions

A ***loop invariant*** is an assertion that expresses a relationship between variables that remains constant throughout all iterations of the loop. In other words, it is a statement that is true both

before the loop is entered and after each pass through the loop. An invariant assertion for the preceding code segment could be

```
divisorSum is the sum of proper divisors of num that are less than or equal
  to trialDivisor.
```

A ***loop variant*** is an assertion whose truth changes between the first and final execution of the loop. The loop variant expression should be stated in such a way that it guarantees the loop is exited. Thus, it contains some statement about the loop variable being incremented (or decremented) during execution of the loop. In the preceding code, we could have

```
trialDivisor is incremented by 1 each time through the loop.
It eventually exceeds the value num / 2, at which point the loop is exited.
```

Variant and invariant assertions usually occur in pairs.

We now use four kinds of assertions—input, output, variant, and invariant—to produce the formally verified loop that follows:

```
divisorSum = 0;

// 1. num is a positive integer.                    (input assertion)
// 2. divisorSum == 0.

assert num > 0 && divisorSum == 0;

for (trialDivisor = 1; trialDivisor <= num / 2; ++trialDivisor)

// trialDivisor is incremented by 1 each time       (variant assertion)
// through the loop. It eventually exceeds the
// value (num / 2), at which point the loop is exited.

    if (num % trialDivisor == 0)
        divisorSum = divisorSum + trialDivisor;

// divisorSum is the sum of proper divisors of       (invariant assertion)
// num that are less than or equal to trialDivisor.

// divisorSum is the sum of                          (output assertion)
// all proper divisors of num.
```

In general, code that is presented in this text does not include formal verification of the loops. This issue is similar to that of robustness. In an introductory course, a decision must be made on the trade-off between learning new concepts and writing robust programs with formal verification of loops. We encourage the practice, but space and time considerations make it inconvenient to include such documentation at this level. We close this discussion with another example illustrating loop verification.

Consider the problem of finding the greatest common divisor (GCD) of two positive integers. Table 7-13 shows some sample inputs and outputs for this problem.

TABLE 7-13
Several examples of the greatest common divisor (GCD) of two numbers

NUM1	NUM2	GCD(NUM 1, NUM2)
8	12	4
20	10	10
15	32	1
70	40	10

A segment of code to produce the GCD of two positive integers after they have been ordered as small, large is

```
int trialGcd = small;
boolean gcdFound = false;
while (! gcdFound)
   if ((large % trialGcd == 0) && (small % trialGcd == 0)){
      gcd = trialGcd;
      gcdFound = true;
   }
   else
      trialGcd = trialGcd - 1;
```

Using assertions as previously indicated, this code would appear as

```
int trialGcd = small;
boolean gcdFound = false;

//      1. small <= large
//      2. trialGcd (small) is the first candidate for gcd
//      3. gcdFound is false

assert small <= large && small == trialGcd && ! gcdFound;

while (! gcdFound)

// trialGcd assumes integer values ranging from small
// to 1. It is decremented by 1 each time through the
// loop. When trialGcd divides both small and large,
// the loop is exited. Exit is guaranteed since 1
// divides both small and large.
```

```
   if ((large % trialGcd == 0) && (small % trialGcd == 0)){

// When trialGcd divides both large and small,
// then gcd is assigned that value.

      assert large % trialGcd == 0 && small % trialGcd == 0;

      gcd = trialGcd;
      gcdFound = true;
   }
   else
      trialGcd = trialGcd - 1;

// Output assertion: gcd is the greatest common divisor of small and large.
```

EXERCISE 7.7

1. Write appropriate input assertions and output assertions for each of the following loops.

a.

```
score = reader.nextInt();
while (score != -999){
   numScores = numScores + 1;
   sum = sum + score;
   System.out.print("Enter a score; -999 to quit: ");
   score = reader.nextInt();
}
```

b.

```
count = 0;
power2 = 1;
while (power2 < 100){
   System.out.println(power2);
   power2 = power2 * 2;
   count = count + 1;
}
```

2. Write appropriate loop invariant and loop variant assertions for each of the loops in Question 1 above.

EXERCISE 7.7 Continued

3. Consider the following loop. The user enters a number, `guess`, and the computer then displays a message indicating whether the guess is correct, too high, or too low. Add appropriate input assertions, output assertions, loop invariant assertions, and loop variant assertions to the following code.

```
correct = false;
count = 0;
while ((count < MAX_TRIES) && (! correct)){
    count = count + 1;
    System.out.print("Enter choice number " + count + ": ");
    guess = reader.nextInt();
    if (guess == choice)
    {
       correct = true;
       System.out.println("Congratulations!");
    }
    else if (guess < choice)
       System.out.println("Your guess is too low");
    else
       System.out.println("Your guess is too high");
}
```

7.8 Advanced Operations on Strings

Thus far in this book, we have used strings without manipulating their contents very much. However, most text-processing applications spend time examining the characters in strings, taking them apart, and building new strings. For example, consider the problem of extracting words from a string representing a line of text. To obtain the first word, we could copy the string's characters to a new string until we reach the first space character in the string (assuming the delimiter between words is the space) or we reach the length of the string. Following is a code segment that uses this strategy:

```
// Create a sample string
String str = "Hi there!";

// Variable to hold the first word, set to empty string
String word = "";

// Visit all the characters in the string
for (int i = 0; i < str.length(); i++){

   // Or stop when a space is found
   if (str.charAt(i) == ' ')
      break;

   // Add the non-space character to the word
   word += str.charAt(i);
}
```

As you can see, this code combines the tasks of finding the first space character and building a ***substring*** of the original string. The problem is solved much more easily by using two separate string methods that are designed for these tasks. The first method, `indexOf`, expects the target character as a parameter and returns the position of the first instance of that character or –1 if the character is not in the string. The second method, `substring`, expects two integer parameters indicating the starting and ending positions of the substring. This method returns the substring that runs from the starting position up to but not including the ending position. Here is a short program that uses these methods:

```
// Example 7.2: Test the methods indexOf and substring

public class TestStringMethods{

   public static void main(String[] args){
      String str = "Hi there!";

      // Search for the position of the first space
      int endPosition = str.indexOf(' ');

      // If there is no space, use the whole string
      if (endPosition == -1)
         endPosition = str.length();

      // Extract the first word
      String word = str.substring(0, endPosition);

      // Output the results
      System.out.println(endPosition);     // Prints 2
      System.out.println(word);            // Prints "Hi"
   }
}
```

Table 7-14 describes some commonly used `String` methods.

TABLE 7-14
Some commonly used `string` methods

METHOD	DESCRIPTION
`charAt (anIndex)` returns char	Example: `chr = myStr.charAt(4);` Returns the character at the position `anIndex`. Remember that the first character is at position 0. An exception is thrown (i.e., an error is generated) if `anIndex` is out of range (i.e., does not indicate a valid position within `myStr`).
`compareTo (aString)` returns int	Example: `i = myStr.compareTo("abc");` Compares two strings alphabetically. Returns 0 if `myStr` equals `aString`, a value less than 0 if `myStr` string is alphabetically less than `aString`, and a value greater than 0 if `myStr` string is alphabetically greater than `aString`.
`equals (aString)` returns boolean	Example: boolean = `myStr.equals("abc");` Returns true if `myStr` equals `aString`; else returns false. Because of implementation peculiarities in Java, never test for equality like this: `myStr == aString`

TABLE 7-14 Continued
Some commonly used `string` methods

METHOD	DESCRIPTION
`equalsIgnoreCase (aString)` returns boolean	Similar to equals but ignores case during the comparison.
`indexOf (aCharacter)` returns int	Example: `i = myStr.indexOf('z');` Returns the index within `myStr` of the first occurrence of `aCharacter` or –1 if `aCharacter` is absent.
`indexOf (aCharacter, beginIndex)` returns int	Example: `i = myStr.indexOf('z', 6);` Similar to the preceding method except the search starts at position `beginIndex` rather than at the beginning of `myStr`. An exception is thrown (i.e., an error is generated) if `beginIndex` is out of range (i.e., does not indicate a valid position within `myStr`).
`indexOf (aSubstring)` returns int	Example: `i = myStr.indexOf("abc");` Returns the index within `myStr` of the first occurrence of `aSubstring` or –1 if `aSubstring` is absent.
`indexOf (aSubstring, beginIndex)` returns int	Example: `i = myStr.indexOf("abc", 6);` Similar to the preceding method except the search starts at position `beginIndex` rather than at the beginning of `myStr`. An exception is thrown (i.e., an error is generated) if `beginIndex` is out of range (i.e., does not indicate a valid position within `myStr`).
`length()` returns int	Example: `i = myStr.length();` Returns the length of `myStr`.
`replace (oldChar, newChar)` returns String	Example: `str = myStr.replace('z', 'Z');` Returns a new string resulting from replacing all occurrences of `oldChar` in `myStr` with `newChar`. **`myStr` is not changed.**
`substring (beginIndex)` returns String	Example: `str = myStr.substring(6);` Returns a new string that is a substring of `myStr`. The substring begins at location `beginIndex` and extends to the end of `myStr`. An exception is thrown (i.e., an error is generated) if `beginIndex` is out of range (i.e., does not indicate a valid position within `myStr`).
`substring (beginIndex, endIndex)` returns String	Example: `str = myStr.substring(4, 8);` Similar to the preceding method except the substring extends to location `endIndex – 1` rather than to the end of `myStr`.
`toLowerCase()` returns String	Example: `str = myStr.toLowerCase();` `str` is the same as `myStr` except that all letters have been converted to lowercase. **`myStr` is not changed.**
`toUpperCase()` returns String	Example: `str = myStr.toUpperCase();` `str` is the same as `myStr` except that all letters have been converted to uppercase. **`myStr` is not changed.**
`trim()` returns String	Example: `str = myStr.trim();` `str` is the same as `myStr` except that leading and trailing spaces, if any, are absent. **`myStr` is not changed.**

Note that there are no mutator methods for strings. The reason for this is that strings are ***immutable objects***. Once a string is created, its length cannot change and one cannot modify any of its characters.

Counting the Words in a Sentence

Our next program illustrates the use of the string methods `length`, `indexOf`, and `substring` to solve a complex problem. The program accepts sentences as inputs from the keyboard. The program displays the number of words in each sentence and the average length of a word. The program assumes that words are separated by at least one blank. Punctuation marks are considered parts of words. The program halts when the user presses just the Enter key. Here is the code followed by a brief discussion:

```
// Example 7.3: Count the words and compute the average
// word length in a sentence

import java.util.Scanner;

public class SentenceStats{

   public static void main(String[] args){

      Scanner reader = new Scanner(System.in);

      // Keep taking inputs
      while (true){
         System.out.print("Enter a sentence: ");
         String input = reader.nextLine();

         // Quit when the user just presses Enter
         if (input.equals(""))
            break;

         // Initialize the counters and indexes
         int wordCount = 0;
         int sentenceLength = 0;
         int beginPosition = 0;
         int endPosition = input.indexOf(' ');

         // Continue until a blank is not seen
         while (endPosition != -1){

            // If at least one nonblank character (a word) was seen
            if (endPosition > beginPosition){
               wordCount++;
               String word = input.substring(beginPosition, endPosition);
               sentenceLength += word.length();
            }

            // Update the indexes to go to the next word
            beginPosition = endPosition + 1;
            endPosition = input.indexOf(' ', beginPosition);
         }
```

```
            // If at least one nonblank character was seen
            // at the end of the sentence, consider it a word
            if (beginPosition < input.length()){
               wordCount++;
               String word = input.substring(beginPosition, input.length());
               sentenceLength += word.length();
            }

            // Trap the case where there were no words
            if (wordCount > 0){
               System.out.println("Word count: " + wordCount);
               System.out.println("Sentence length: " + sentenceLength);
               System.out.println("Average word length: " +
                                  sentenceLength / wordCount);
            }
         }
      }
   }
```

This program contains two loops. The outer loop accepts inputs from the user and runs the nested loop to process each input. The nested loop advances two indexes through an input string. On each pass through this loop, the indexes represent the beginning and ending positions of each word. Initially, the beginning position is 0 and the ending position is the result of running `indexOf` with a blank character. At this point, there are several cases to consider:

1. `indexOf` returns –1, meaning that a blank was not found. At this point, the nested loop has moved through the entire input string and there are no more blanks to be seen, or the loop is not entered at all. If the beginning position is less than the length of the input string at this point (below the loop), the last word needs to be counted and extracted from the input string. The program uses the `substring` method with the length of the input string as the ending position in this case.
2. `indexOf` returns an ending position that is greater than the current beginning position. This means that at least one nonblank character (a word) has been seen before encountering a blank. In this case, the program continues in the nested loop. It increments the word count and extracts the word using `substring` with the current beginning and ending positions. The program then increments the sentence length by the length of the word.
3. `indexOf` returns an ending position that is equal to the current beginning position. This happens if there are any leading blanks in the sentence or there is more than one blank following a word. In this case, the program should not count a word or attempt to extract it from the input string, but just continue in the nested loop. This will have the effect of scanning over the extra blank and ignoring it.
4. In cases 2 and 3, the nested loop continues by updating the beginning and ending positions. The beginning position is incremented to 1 greater than the ending position. Then the ending position is reset to the result of running `indexOf` once more with a blank and the new beginning position. This has the effect of advancing the positions to the beginning and ending positions of the next word, if there is one.

As you can see, counting the words in a sentence using `String` methods is not a simple task. We see a much easier way to do this shortly.

Using a Scanner with a String

Until now, we have used a `Scanner` object to accept the input of integers, floating-point numbers, and lines of text from the keyboard or from text files. Interestingly, a scanner can also be used to read words from a string. When used with a file, the `Scanner` method `next()` skips any leading blanks and reads and returns a string containing the next sequence of nonblank characters. The method `next()` has the same behavior when used with a scanner that has been opened on a string. The method `hasNext()` returns `true` if there are still more words in the string to be scanned. The following code segment opens a scanner on a string and uses it to scan and display the words in it:

```
String str = "The rain in Spain falls mainly on the plain.";
Scanner reader = new Scanner(str);
while (reader.hasNext())
   System.out.println(reader.next());
```

The scanner automatically handles the tedious details, such as skipping multiple spaces between words, which made the program of Example 7.2 so complicated. A project at the end of this chapter asks you to simplify this program by using a scanner.

EXERCISE 7.8

1. Indicate the outputs of the following code segments:

a.
```
String str = "The rain in Spain falls mainly on the plain";
System.out.println(str.indexOf(' '));
```

b.
```
String str = "The rain in Spain falls mainly on the plain";
System.out.println(str.indexOf(' ', 4));
```

c.
```
String str = "The rain in Spain falls mainly on the plain";
System.out.println(str.substring(4));
```

d.
```
String str = "The rain in Spain falls mainly on the plain";
System.out.println(str.substring(4, 8));
```

e.
```
String str = "The rain in Spain falls mainly on the plain";
int begin = 0;
while (begin < str.length()){
   int end = str.indexOf(' ', begin);
   if (end == -1)
      end = str.length();
   String word = str.substring(begin, end);
   System.out.println(word);
   begin = end + 1;
}
```

EXERCISE 7.8 Continued

2. Write code segments that perform the following tasks:

a. Replace every blank space character in the string `str` with a newline character (`'\n'`).

b. Find the index of the first instance of the substring `"the"` in the string `str`.

c. Find the index of the first instance of the substring `"the"` after the midpoint of the string `str`.

d. Count the number of instances of the whole word `"the"` in the string `str`.

7.9 Graphics and GUIs: Timers and Animations

The shapes, text, and images that we have displayed in graphics applications thus far have been more or less passive objects. The user can move them around a window with a mouse, but none of them know how to move on their own. Now it is time to make these graphics objects active. In this section, we explore the use of timers to construct simple animations.

The Basic Principles of Animation

Extra Challenge

This Graphics and GUIs section gives you the opportunity to explore concepts and programming techniques required to develop modern graphics applications and graphical user interfaces. This material is not required in order to proceed with the other chapters of the book.

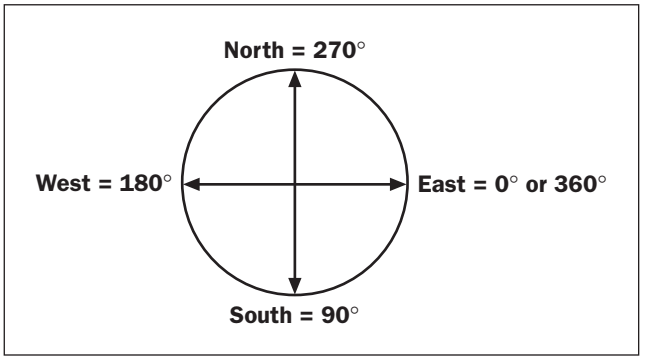
As described in Chapter 1, our perception of movement in a motion picture is based on a rapid display of successive frames. In each frame, an object is shown at a different position, but if we are shown 24 frames per second, these changes of position appear as the continuous motion of the object itself.

We already know how to change the position of a graphical object and repaint a panel, so the basic tools for displaying the same object in multiple frames are already available. The speed with which an object appears to change its position depends on the distance it travels between frames and on the number of frames displayed per unit of time. Faster objects can travel a greater distance between frames than slower ones. The realistic depiction of motion also involves many other factors, such as rates of acceleration, the resistance of friction, and some qualities of the objects themselves, such as the "bounciness" of a ball. In addition, computer video artists must worry about such phenomena as the "flicker" caused by the speed of the display medium when painting large, complex images that must appear to move rapidly. In the examples that follow, we ignore most of these details and focus on the simple display of constant motion in two dimensions.

Direction and Speed of Moving Objects

A moving object has a speed, which is the distance (in pixels) traveled in a given unit of time, and a direction. An object's direction can be fixed as the angle of its path relative to a base direction. Let's assume that the base direction is due east (to the right of the panel), at 0 degrees. Moving clockwise, south (to the bottom) is 90 degrees, west (to the left) is 180 degrees, north (to the top) is 270 degrees, and 360 degrees returns the direction to due east, as shown in Figure 7-7.

FIGURE 7-7
Representing directions in two dimensions



At any given time, we should be able to change a graphical object's speed and direction. Using good object-oriented style, we ask a graphical object, where possible, to track its own speed and direction. For example, we can add some instance variables and methods to the `Circle` class of Chapter 6 to give it this functionality. Circles have an initial speed of 0 and an initial direction of 0 degrees (due east). Table 7-15 lists these new methods.

TABLE 7-15
New methods for the circle class

NEW CIRCLE METHOD	WHAT IT DOES
`void move()`	Changes the position of the circle using its current speed and direction.
`void setDirection (int degrees)`	Sets the direction of the circle to the given direction in degrees (the default is 0 degrees for due east).
`void setSpeed(int s)`	Sets the speed of the circle to the given speed (the default is 0 pixels).
`void turn(int degrees)`	Adds the given amount of degrees to the circle's current direction. If `degrees` is positive, the direction rotates clockwise. If `degrees` is negative, the direction rotates counterclockwise.

The critical new method for movable objects is `move()`. This method moves an object a given distance in a given direction. To implement this method, we must calculate the distances to move in the x and y directions based on the object's current position, its speed, and its direction. According to basic trigonometry, the x distance is equal to the speed multiplied by the cosine of the direction angle in radians. The y distance is equal to the speed multiplied by the sine of the direction angle in radians. We use the methods `Math.radians`, `Math.cos`, and `Math.sin` to compute these values and then cast the results to integers. Here is the code for the new instance variables and methods:

```
public void setSpeed(int s){
   speed = s;
}
```

```
public void setDirection(int degrees){
   direction = degrees % 360;
}

public void turn(int degrees){
   direction = (direction + degrees) % 360;
}

public void move(){
   move((int)(speed * Math.cos(Math.toRadians(direction))),
        (int)(speed * Math.sin(Math.toRadians(direction))));
}
```

Note that the methods `setDirection` and `turn` use the `%` operator to wrap the direction around if it's greater than 360 degrees (for example, `365 % 360` is the same direction as `5`).

Moving a Circle with the Mouse

Our next example program illustrates the use of our new `Circle` methods. The program displays a filled circle at the center of the panel. When the user presses the mouse, the circle moves 50 pixels in its current direction and then turns 45 degrees. Repeated mouse presses (or "clicks") cause the circle to move in a circular pattern and return to its original position. Here is the code for the panel class:

```
// Example 7.4: Moves the circle 50 pixels and
// turns it 45 degrees in response to a mouse press

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ColorPanel extends JPanel{

   private Circle circle;

   public ColorPanel(Color backColor, int width, int height){
      setBackground(backColor);
      setPreferredSize(new Dimension(width, height));
      // Circle centered in the panel with radius 25
      circle = new Circle(width / 2, height / 2, 25, Color.red);
      circle.setFilled(true);
      // Move 50 pixels per mouse press
      circle.setSpeed(50);
      addMouseListener(new MoveListener());
   }

   public void paintComponent(Graphics g){
      super.paintComponent(g);
      circle.draw(g);
   }
```

```
   private class MoveListener extends MouseAdapter{

      public void mousePressed(MouseEvent e){
         circle.move();
         circle.turn(45);    // Turn 45 degrees
         repaint();
      }
   }
}
```

Timers

A basic algorithm for animating a graphical object can be expressed as follows:

```
Set the initial position of the shape
At regular intervals
   Move the object
   Repaint the panel
```

The first step of the algorithm is accomplished when the panel is instantiated. When the window is displayed, the object is painted in its initial position. The last two steps are accomplished by sending the standard messages to move the object and repaint the panel. For "regular intervals" we could use intervals of about 33 milliseconds, which corresponds to repainting the scene 30 times per second and giving the illusion of continuous motion provided that the changes between frames are not too great. But how can this be done automatically at regular intervals? A simple loop will not work. Because execution speeds vary from computer to computer, a simple loop might move the same object faster on one computer than on another.

Java provides a special type of object called a timer to schedule events at regular intervals. When a timer is instantiated, it is given an interval in milliseconds and a listener object similar to those discussed in Chapter 6. When the timer is sent the `start` message, its clock starts ticking. When each interval of time passes, the listener's `actionPerformed` method is triggered. In the case of our animation, this method runs the operations to move the object and repaint the panel. Because the timer uses the computer's clock to measure the intervals, they will not vary with the execution speed of the computer.

The timer for animations is an instance of the class `Timer`, which is included in the package `javax.swing`. Assuming that we have the listener class `MoveListener` at our disposal, we can declare, instantiate, and start a timer as follows:

```
javax.swing.Timer projectortimer;
timer  = new javax.swing.Timer(33, new MoveListener());
timer.start();
```

Note that we prefix the `Timer` class with its package name, so as to distinguish this `Timer` from a different class with the same name in the package `java.util`. This practice will allow us eventually to use resources from `java.util` without name conflicts. Our timer is started with an interval of 33 milliseconds. The timer ticks for 33 milliseconds and then fires an event. The `MoveListener` detects this event and runs its `actionPerformed` method. The timer automatically repeats this process until it is stopped or the program quits.

The definition of the class `MoveListener`, like the listener class for a mouse event in the preceding example, is nested in the panel class. This placement allows the listener's `actionPerformed` method to access the data within the panel. Unlike the earlier listener classes, however, `MoveListener` does not extend an adapter class, but instead implements the `ActionListener` interface. For now, we won't worry about this difference, but just assume that our class includes an `actionPerformed` method. Here is a template for this class:

```
private class MoveListener implements ActionListener{

   public void actionPerformed(ActionEvent e){
      // Code for moving objects goes here
      repaint();
   }
}
```

Like the code for other listener classes, this code requires you to import the package `javax.awt.event`.

Moving a Circle with a Timer

We are now ready to automate the movement of the circle in the previous program by using a timer. The `ColorPanel` class still has the same basic structure, but the `MoveListener` class now implements the `ActionListener` interface and packages the code to move the circle in an `actionPerformed` method. After the circle is instantiated, we create a timer with an interval of 33 milliseconds. The timer is also passed a new `MoveListener` and then started up. The rest, as they say, is automatic. Here is the code for the modified `ColorPanel` class:

```
// Example 7.5: Moves the circle 6 pixels and
// turns it 2 degrees in response to timer events

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ColorPanel extends JPanel{

   private Circle circle;
   private javax.swing.Timer timer;

   public ColorPanel(Color backColor, int width, int height){
      setBackground(backColor);
      setPreferredSize(new Dimension(width, height));
      // Circle centered in the panel with radius 25
      circle = new Circle(width / 2, height / 2, 25, Color.red);
      // Move the circle at a rate of 180 pixels per second, that is,
      // 6 pixels 30 times per second
      circle.setSpeed(6);
      // Fire timer events every 1/8 second
      timer = new javax.swing.Timer(33, new MoveListener());
      timer.start();
   }
```

```
    public void paintComponent(Graphics g){
       super.paintComponent(g);
       circle.fill(g);
    }

    private class MoveListener implements ActionListener{

       public void actionPerformed(ActionEvent e){
          circle.move();
          circle.turn(2);   // Turn 2 degrees
          repaint();
       }
    }
}
```

A Final Example: A Bouncing Circle

Let's incorporate the ideas examined thus far into a program that bounces a circle back and forth horizontally in its panel. At program startup, the circle's left side is flush with the left border of the panel. It moves continuously to the right, until its right side is flush with the panel's right border. At that point, the circle reverses direction and returns to the left border, where it reverses direction again, and so on, indefinitely.

Clearly, the action to move the circle must check to see if the circle has hit a panel boundary and then change the circle's direction if necessary. The circle's default direction is due east (0 degrees). Because the circle will start next to the left boundary of the panel, we set its initial direction to due west (180 degrees) so that it hits the left boundary first. Here is the code for the `ColorPanel` class:

```
// Example 7.6: A filled circle moves back and forth
// across the panel, appearing to bounce off its edges

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ColorPanel extends JPanel{

   private Circle circle;
   private javax.swing.Timer timer;

   public ColorPanel(Color backColor, int width, int height){
      setBackground(backColor);
      setPreferredSize(new Dimension(width, height));
      // Circle with center point (25, height / 2) and radius 25
      circle = new Circle(25, height / 2, 25, Color.red);
      // Aim due west to hit left boundary first
      circle.setDirection(180);
      // Move 180 pixels per second with 30 frames/second
      circle.setSpeed(6);
      // Move every 33 milliseconds
      timer = new javax.swing.Timer(33, new MoveListener());
      timer.start();
   }
```

```
    public void paintComponent(Graphics g){
       super.paintComponent(g);
       circle.draw(g);
    }

    private class MoveListener implements ActionListener{

       public void actionPerformed(ActionEvent e){
          int x = circle.getX(); // New method in class Circle
          int radius = circle.getRadius(); // New method in class Circle
          int width = getWidth();
          // Check for boundaries and reverse direction
          // if necessary
          if (x - radius <= 0 || x + radius >= width)
             circle.turn(180);
          circle.move();
          repaint();
       }
    }
}
```

Other Timer Methods

Once it is started, a timer fires events until it is stopped or the program quits. The `Timer` class includes methods for stopping a timer, restarting it, changing its time interval, and so forth. Some of the more commonly used methods are listed in Table 7-16.

TABLE 7-16
Some commonly used timer methods

TIMER METHOD	WHAT IT DOES
`boolean isRunning()`	Returns true if the timer is firing events or false otherwise
`void restart()`	Restarts a timer, causing it to fire the first event after its initial delay
`void setDelay(int delay)`	Sets the timer's delay to the number of milliseconds between events
`void setInitialDelay(int delay)`	Sets the timer's initial delay, which by default is its between-event delay
`void stop()`	Stops the timer, causing it to cease firing events

EXERCISE 7.9

1. Describe how to set up a timer and explain what it does.
2. Describe the factors that affect our perception of the movement of a graphical object.
3. What causes flicker? How can flicker be eliminated?

EXERCISE 7.9 Continued

4. How do the direction and speed of an object determine where it will be placed after a given unit of time?
5. Suppose you want to move a ball 300 pixels per second while avoiding flicker. What values would you recommend for the Timer's interval and the ball's speed?
6. Modify the code for the bouncing ball so that it does not begin by travelling west. What happens and why?
7. In Example 7.5 (moving a ball in a circle), try to predict the effect of each of the following and explain why. Check your prediction by running the program.
   a. Increasing the speed.
   b. Turning more degrees.
   c. Increasing the Timer's interval.

Design, Testing, and Debugging Hints

- Most errors involving selection statements and loops are not syntax errors caught at compile time. Thus, you will detect these errors only after running the program, and perhaps then only with extensive testing.
- The presence or absence of braces can seriously affect the logic of a selection statement or loop. For example, the following selection statements have a similar look but a very different logic:

```
if (x > 0){
   y = x;
   z = 1 / x;
}

if (x > 0)
   y = x;
   z = 1 / x;
```

- The first selection statement above guards against division by 0; the second statement only guards against assigning `x` to `y`. The following pair of code segments shows a similar problem with a loop:

```
while (x > 0){
   y = x;
   x = x - 1;
}

while (x > 0)
   y = x;
   x = x - 1;
```

- The first loop above terminates because the value of x decreases within the body of the loop; the second loop is infinite because the value of x decreases below the body of the loop.
- When testing programs that use `if` or `if-else` statements, be sure to use test data that force the program to exercise all of the logical branches.
- When testing a program that uses `if` statements, it helps to formulate equivalence classes, boundary conditions, and extreme conditions.
- Use an `if-else` statement rather than two `if` statements when the alternative courses of action are mutually exclusive.
- When testing a loop, be sure to use limit values as well as typical values. For example, if a loop should terminate when the control variable equals 0, run it with the values 0, –1, and 1.
- Be sure to check entry conditions and exit conditions for each loop.
- For a loop with errors, use debugging output statements to verify the values of the control variable on each pass through the loop. Check this value before the loop is initially entered, after each update, and after the loop is exited.

SUMMARY

In this chapter, you learned:

- A complex Boolean expression contains one or more Boolean expressions and the logical operators `&&` (AND), `||` (OR), and `!` (NOT).
- A truth table can determine the value of any complex Boolean expression.
- Java uses short-circuit evaluation of complex Boolean expressions. The evaluation of the operands of `||` stops at the first true value, whereas the evaluation of the operands of `&&` stops at the first false value.
- Nested `if` statements are another way of expressing complex conditions. A nested `if` statement can be translated to an equivalent `if` statement that uses logical operators.
- An extended or multiway `if` statement expresses a choice among several mutually exclusive alternatives.
- Loops can be nested in other loops.
- Equivalence classes, boundary conditions, and extreme conditions are important features used in tests of control structures involving complex conditions.
- You can verify the correctness of a loop by using assertions, loop variants, and loop invariants.

VOCABULARY *Review*

Define the following terms:

Arithmetic overflow	Extreme condition	Nested loop
Boundary condition	Input assertion	Output assertion
Combinatorial explosion	Logical operator	Quality assurance
Complete code coverage	Loop invariant	Robust
Equivalence class	Loop variant	Truth table
Extended `if` statement	Nested `if` statement	

REVIEW *Questions*

WRITTEN QUESTIONS

Write a brief answer to each of the following questions.

1. List the three logical operators.

2. Construct a truth table for the expression P OR NOT Q.

3. Suppose P is true and Q is false. What is the value of the expression P AND NOT Q?

4. Write an `if` statement that displays whether or not a given number is between a lower bound `min` and an upper bound `max`, inclusive. Use a logical operator in the condition.

5. Rewrite the `if` statement in Question 4 to use a nested `if` statement.

6. Write a nested loop that displays a 10-by-10 square of asterisks.

7. Give an example of an assertion and show how it can be checked with Java's assert statement.

8. Explain the role that variant and invariant assertions play in showing that a loop is correct.

PROJECTS

In keeping with the spirit of this chapter, each program should be robust and should validate the input data. You should try also to formulate the appropriate equivalence classes, boundary conditions, and extreme conditions and use them in testing the programs.

PROJECT 7-1

In a game of guessing numbers, one person says, "I'm thinking of a number between 1 and 100." The other person guesses "50." The first person replies, "No, the number is less." The second person then guesses "25," and so on, until she guesses correctly. Write a program that plays this game. The computer knows the number (a random number between 1 and 100) and the user is the guesser. At the end of the game, the computer displays the number of guesses required by the user to guess the number correctly.

PROJECT 7-2

Rewrite the program of Project 7-1 so that the user knows the number and the computer must guess it.

PROJECT 7-3

Write a program that expects a numeric grade as input and outputs the corresponding letter grade. The program uses the following grading scale:

NUMERIC RANGE	LETTER GRADE
96–100	A+
92–95	A
90–91	A–
86–89	B+
82–85	B
80–81	B–
76–79	C+
72–75	C
70–71	C–
66–69	D+
62–65	D
60–61	D–
0–59	F

PROJECT 7-4

Write a Java method `getLetterGrade` that is based on the grading scale of Project 7-3. This method expects the numeric grade as a parameter and returns a string representing the letter grade. The method header should have the prefix static so it can be called from main. Use this method in a program that inputs a list of grades (ending with –1) and outputs the class average, the class minimum, and the class maximum as letter grades.

PROJECT 7-5

The Euclidean algorithm can be used to find the greatest common divisor (GCD) of two positive integers (n1, n2). You can use this algorithm in the following manner:

A. Compute the remainder of dividing the larger number by the smaller number.
B. Replace the larger number with the smaller number and the smaller number with the remainder.
C. Repeat this process until the smaller number is zero:

The larger number at this point is the GCD of n1 and n2.

Write a program that lets the user enter two integers and then prints each step in the process of using the Euclidean algorithm to find their GCD.

PROJECT 7-6

Review the case study in Chapter 4 in which the Lucky Sevens gambling game program was created. Remove the code that deals with the maximum amount held. Then modify the program so that it runs the simulation 100 times and prints the average number of rolls. (*Hint*: Put the while loop inside a for statement that loops 100 times. Accumulate the total count and at the end divide by 100.)

PROJECT 7-7

Write a program to print the perimeter and area of rectangles using all combinations of heights and widths running from 1 foot to 10 feet in increments of 1 foot. Print the output in headed, formatted columns.

PROJECT 7-8

Write a program that uses a scanner to report some statistics about words in an input sentence (see Section 7.8). The outputs should be the number of words in the sentence, the average word length, and the length of the sentence.

PROJECT 7-9

Write a program that allows the user to search for a given word in a text file. The two inputs are the file's name and the target word. If the target is not found, the program outputs a message to that effect. Otherwise, the program outputs the number of times that this word occurs in the file and the position where it is first encountered (counting from position 0). The program should ignore case when it compares words.

PROJECT 7-10

Modify the example program of Section 7.8 so that the circle stops moving when the user clicks the mouse. When the user clicks the mouse again, the circle should resume moving. (*Hint:* Define a mouse listener class as shown in Chapter 6.)

PROJECT 7-11

Add another circle to the program of Project 7-10. The second circle should be placed at the right margin of the panel at program startup, exactly opposite the first circle. Both circles should reverse direction when they hit a boundary.

PROJECT 7-12

Use your knowledge of physics to make an interesting change to the program of Project 7-11. Set the initial directions of the two circles to angles other than horizontal (say, 120 degrees for one and 30 degrees for the other). When a circle hits a boundary, it should rebound at the appropriate angle. (*Hint*: The angle of reflection should equal the angle of incidence.)

CRITICAL *Thinking*

Read the sections of the ACM Code of Ethics that deal with designing and testing reliable computer systems. Prepare and present a report that explains how the ACM Code deals with this issue.