

CHAPTER 5

DECISIONS



CHAPTER GOALS


- To implement decisions using if statements
- To compare integers, floating-point numbers, and strings
- To write statements using the Boolean data type
- To develop strategies for testing your programs
- To validate user input

CHAPTER CONTENTS

5.1 THE IF STATEMENT 180

- Syntax 5.1:* if Statement 182
- Programming Tip 5.1:* Brace Layout 184
- Programming Tip 5.2:* Always Use Braces 184
- Common Error 5.1:* A Semicolon After the if Condition 184
- Programming Tip 5.3:* Tabs 185
- Special Topic 5.1:* The Conditional Operator 185
- Programming Tip 5.4:* Avoid Duplication in Branches 186

5.2 COMPARING VALUES 186

- Syntax 5.2:* Comparisons 187
- Common Error 5.2:* Using == to Compare Strings 192
- How To 5.1:* Implementing an if Statement 193
- Worked Example 5.1:* Extracting the Middle 
- Computing & Society 5.1:* Denver's Luggage Handling System 195

5.3 MULTIPLE ALTERNATIVES 196

- Special Topic 5.2:* The switch Statement 199

5.4 NESTED BRANCHES 200

- Programming Tip 5.5:* Hand-Tracing 203

- Common Error 5.3:* The Dangling else Problem 204

- Special Topic 5.3:* Block Scope 205

- Special Topic 5.4:* Enumeration Types 206

5.5 PROBLEM SOLVING: FLOWCHARTS 207

5.6 PROBLEM SOLVING: SELECTING TEST CASES 210

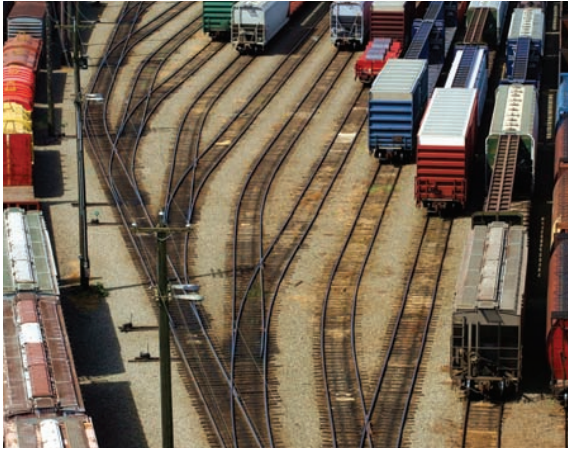
- Programming Tip 5.6:* Make a Schedule and Make Time for Unexpected Problems 212
- Special Topic 5.5:* Logging 212

5.7 BOOLEAN VARIABLES AND OPERATORS 213

- Common Error 5.4:* Combining Multiple Relational Operators 216
- Common Error 5.5:* Confusing && and || Conditions 216
- Special Topic 5.6:* Short-Circuit Evaluation of Boolean Operators 217
- Special Topic 5.7:* De Morgan's Law 217

5.8 APPLICATION: INPUT VALIDATION 218

- Computing & Society 5.2:* Artificial Intelligence 221



One of the essential features of computer programs is their ability to make decisions. Like a train that changes tracks depending on how the switches are set, a program can take different actions depending on inputs and other circumstances.

In this chapter, you will learn how to program simple and complex decisions. You will apply what you learn to the task of checking user input.

5.1 The if Statement

The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed.

The `if` statement is used to implement a decision (see Syntax 5.1). When a condition is fulfilled, one set of statements is executed. Otherwise, another set of statements is executed.

Here is an example using the `if` statement: In many countries, the number 13 is considered unlucky. Rather than offending superstitious tenants, building owners sometimes skip the thirteenth floor; floor 12 is immediately followed by floor 14. Of course, floor 13 is not usually left empty or, as some conspiracy theorists believe, filled with secret offices and research labs. It is simply called floor 14. The computer that controls the building elevators needs to compensate for this foible and adjust all floor numbers above 13.

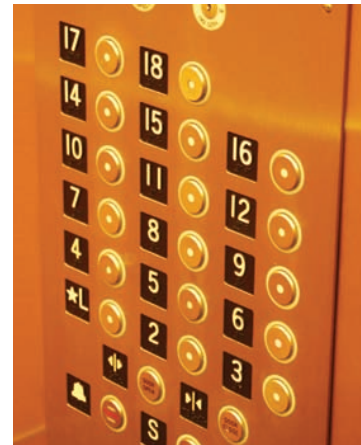
Let's simulate this process in Java. We will ask the user to type in the desired floor number and then compute the actual floor. When the input is above 13, then we need to decrement the input to obtain the actual floor. For example, if the user provides an input of 20, the program determines the actual floor to be 19. Otherwise, it simply uses the supplied floor number.

```
int actualFloor;

if (floor > 13)
{
    actualFloor = floor - 1;
}
else
{
    actualFloor = floor;
}
```

The flowchart in Figure 1 shows the branching behavior.

In our example, each branch of the `if` statement contains a single statement. You can include as many statements in each branch as you like. Sometimes, it happens that



This elevator panel “skips” the thirteenth floor. The floor is not actually missing—the computer that controls the elevator adjusts the floor numbers above 13.

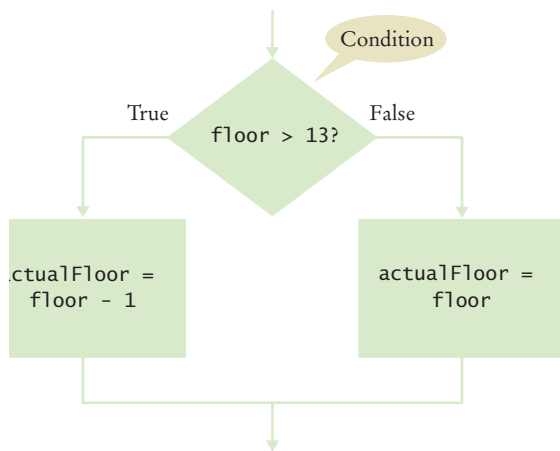


Figure 1
Flowchart for if Statement

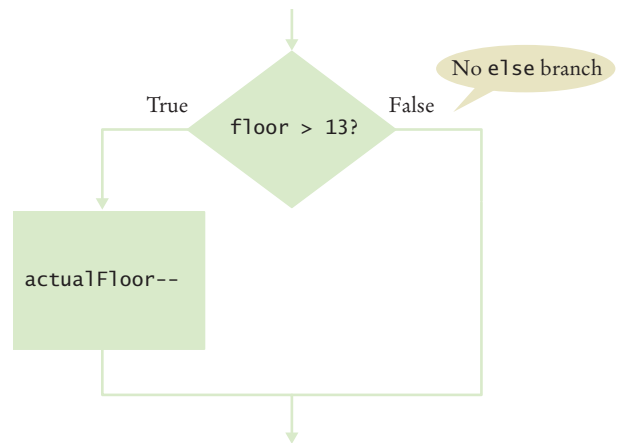


Figure 2
Flowchart for if Statement with No else Branch

there is nothing to do in the else branch of the statement. In that case, you can omit it entirely, such as in this example:

```

int actualFloor = floor;

if (floor > 13)
{
    actualFloor--;
} // No else needed
  
```

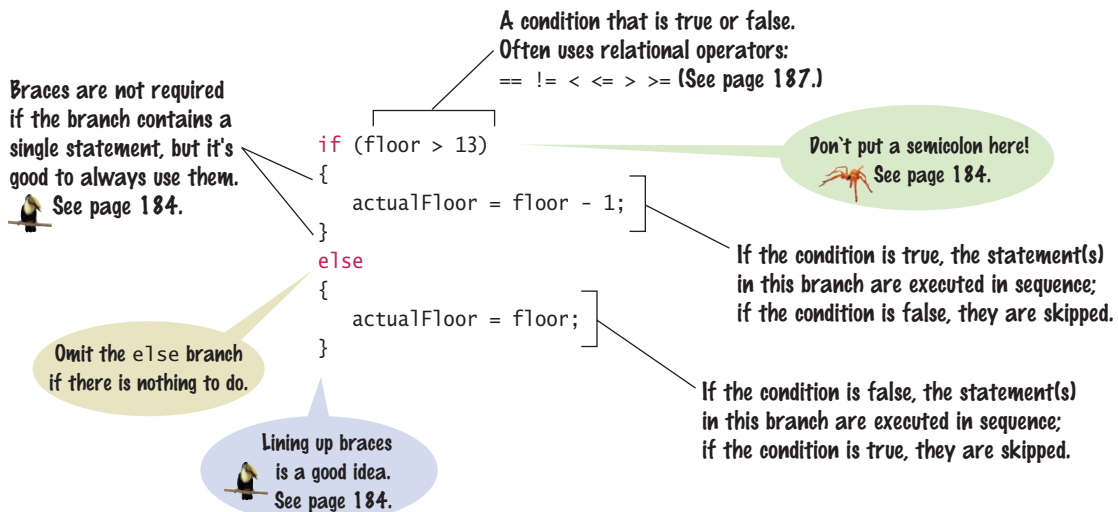
See Figure 2 for the flowchart.



An if statement is like a fork in the road. Depending upon a decision, different parts of the program are executed.

Syntax 5.1 if Statement

Syntax **if** (*condition*) **if** (*condition*) { *statements*₁ }
 {
 statements
 }



The following program puts the if statement to work. This program asks for the desired floor and then prints out the actual floor.

section_1/ElevatorSimulation.java

```

1  import java.util.Scanner;
2
3  /**
4   * This program simulates an elevator panel that skips the 13th floor.
5   */
6  public class ElevatorSimulation
7  {
8      public static void main(String[] args)
9      {
10         Scanner in = new Scanner(System.in);
11         System.out.print("Floor: ");
12         int floor = in.nextInt();
13
14         // Adjust floor if necessary
15
16         int actualFloor;
17         if (floor > 13)
18         {
19             actualFloor = floor - 1;
20         }
21         else
22         {
  
```

```

23         actualFloor = floor;
24     }
25
26     System.out.println("The elevator will travel to the actual floor "
27         + actualFloor);
28 }
29 }

```

Program Run

```

Floor: 20
The elevator will travel to the actual floor 19

```



1. In some Asian countries, the number 14 is considered unlucky. Some building owners play it safe and skip *both* the thirteenth and the fourteenth floor. How would you modify the sample program to handle such a building?

2. Consider the following if statement to compute a discounted price:

```

if (originalPrice > 100)
{
    discountedPrice = originalPrice - 20;
}
else
{
    discountedPrice = originalPrice - 10;
}

```

What is the discounted price if the original price is 95? 100? 105?

3. Compare this if statement with the one in Self Check 2:

```

if (originalPrice < 100)
{
    discountedPrice = originalPrice - 10;
}
else
{
    discountedPrice = originalPrice - 20;
}

```

Do the two statements always compute the same value? If not, when do the values differ?

4. Consider the following statements to compute a discounted price:

```

discountedPrice = originalPrice;
if (originalPrice > 100)
{
    discountedPrice = originalPrice - 10;
}

```

What is the discounted price if the original price is 95? 100? 105?

5. The variables `fuelAmount` and `fuelCapacity` hold the actual amount of fuel and the size of the fuel tank of a vehicle. If less than 10 percent is remaining in the tank, a status light should show a red color; otherwise it shows a green color. Simulate this process by printing out either "red" or "green".

Practice It Now you can try these exercises at the end of the chapter: R5.5, R5.6, E5.9.

Programming Tip 5.1

**Brace Layout**

The compiler doesn't care where you place braces. In this book, we follow the simple rule of making { and } line up.

```
if (floor > 13)
{
    floor--;
}
```

This style makes it easy to spot matching braces. Some programmers put the opening brace on the same line as the if:

```
if (floor > 13) {
    floor--;
}
```

This style makes it harder to match the braces, but it saves a line of code, allowing you to view more code on the screen without scrolling. There are passionate advocates of both styles.

It is important that you pick a layout style and stick with it consistently within a given programming project. Which style you choose may depend on your personal preference or a coding style guide that you need to follow.



Properly lining up your code makes your programs easier to read.

Programming Tip 5.2

**Always Use Braces**

When the body of an if statement consists of a single statement, you need not use braces. For example, the following is legal:

```
if (floor > 13)
    floor--;
```

However, it is a good idea to always include the braces:

```
if (floor > 13)
{
    floor--;
}
```

The braces make your code easier to read. They also make it easier for you to maintain the code because you won't have to worry about adding braces when you add statements inside an if statement.

Common Error 5.1

**A Semicolon After the if Condition**

The following code fragment has an unfortunate error:

```
if (floor > 13) ; // ERROR
{
    floor--;
}
```

There should be no semicolon after the if condition. The compiler interprets this statement as follows: If floor is greater than 13, execute the statement that is denoted by a single semicolon, that is, the do-nothing statement. The statement enclosed in braces is no longer a part of the if

statement. It is always executed. In other words, even if the value of `floor` is not above 13, it is decremented.

Programming Tip 5.3

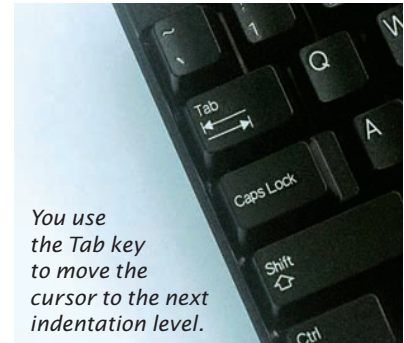


Tabs

Block-structured code has the property that nested statements are indented by one or more levels:

```
public class ElevatorSimulation
{
    public static void main(String[] args)
    {
        int floor;
        . . .
        if (floor > 13)
        {
            floor--;
        }
        . . .
    }
}
```

0 1 2 3 Indentation level



You use the Tab key to move the cursor to the next indentation level.

How do you move the cursor from the leftmost column to the appropriate indentation level? A perfectly reasonable strategy is to hit the space bar a sufficient number of times. With most editors, you can use the Tab key instead. A tab moves the cursor to the next indentation level. Some editors even have an option to fill in the tabs automatically.

While the Tab *key* is nice, some editors use *tab characters* for alignment, which is not so nice. Tab characters can lead to problems when you send your file to another person or a printer. There is no universal agreement on the width of a tab character, and some software will ignore tab characters altogether. It is therefore best to save your files with spaces instead of tabs. Most editors have a setting to automatically convert all tabs to spaces. Look at the documentation of your development environment to find out how to activate this useful setting.

Special Topic 5.1



The Conditional Operator

Java has a *conditional operator* of the form

condition ? *value*₁ : *value*₂

The value of that expression is either *value*₁ if the test passes or *value*₂ if it fails. For example, we can compute the actual floor number as

```
actualFloor = floor > 13 ? floor - 1 : floor;
```

which is equivalent to

```
if (floor > 13) { actualFloor = floor - 1; } else { actualFloor = floor; }
```

You can use the conditional operator anywhere that a value is expected, for example:

```
System.out.println("Actual floor: " + (floor > 13 ? floor - 1 : floor));
```

We don't use the conditional operator in this book, but it is a convenient construct that you will find in many Java programs.

Programming Tip 5.4

**Avoid Duplication in Branches**

Look to see whether you *duplicate code* in each branch. If so, move it out of the if statement. Here is an example of such duplication:

```
if (floor > 13)
{
    actualFloor = floor - 1;
    System.out.println("Actual floor: " + actualFloor);
}
else
{
    actualFloor = floor;
    System.out.println("Actual floor: " + actualFloor);
}
```

The output statement is exactly the same in both branches. This is not an error—the program will run correctly. However, you can simplify the program by moving the duplicated statement, like this:

```
if (floor > 13)
{
    actualFloor = floor - 1;
}
else
{
    actualFloor = floor;
}
System.out.println("Actual floor: " + actualFloor);
```

Removing duplication is particularly important when programs are maintained for a long time. When there are two sets of statements with the same effect, it can easily happen that a programmer modifies one set but not the other.

5.2 Comparing Values

Use relational operators (< <= > >= == !=) to compare numbers.

Every if statement contains a condition. In many cases, the condition involves comparing two values. In the following sections, you will learn how to implement comparisons in Java.

5.2.1 Relational Operators

Relational operators compare values. The == operator tests for equality.

A **relational operator** tests the relationship between two values. An example is the > operator that we used in the test `floor > 13`. Java has six relational operators (see Table 1).

In Java, you use a relational operator to check whether one value is greater than another.



Table 1 Relational Operators		
Java	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

As you can see, only two Java relational operators (> and <) look as you would expect from the mathematical notation. Computer keyboards do not have keys for ≥, ≤, or ≠, but the >=, <=, and != operators are easy to remember because they look similar. The == operator is initially confusing to most newcomers to Java.

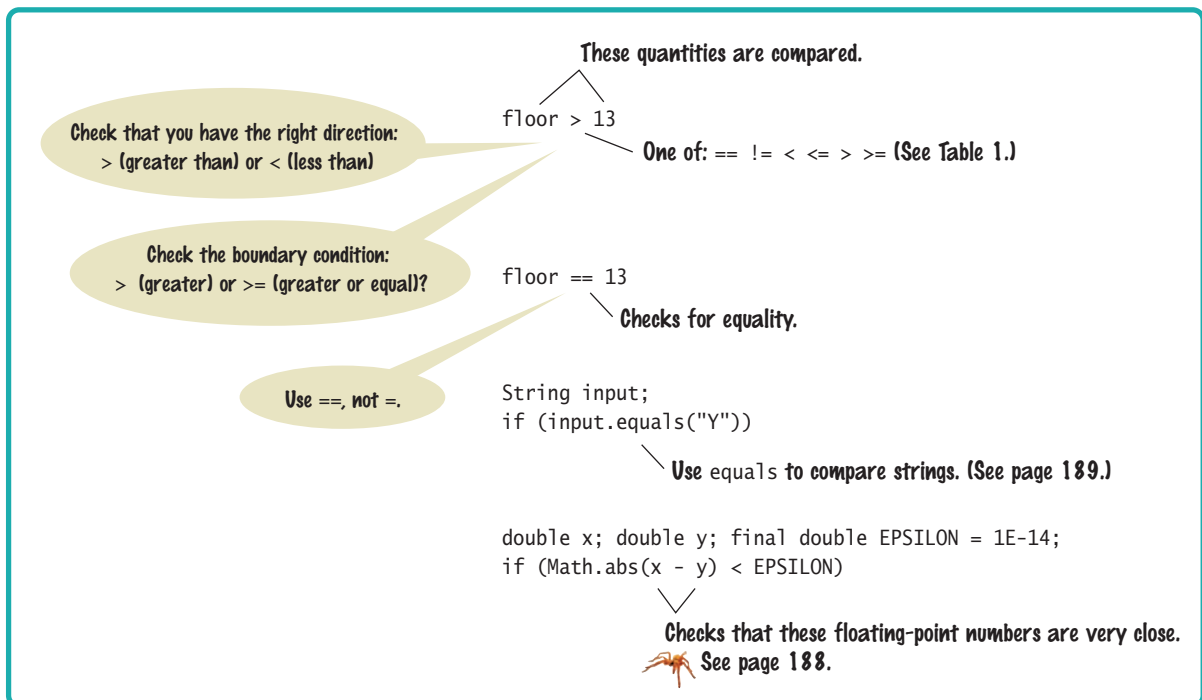
In Java, = already has a meaning, namely assignment. The == operator denotes equality testing:

```
floor = 13; // Assign 13 to floor
```

```
if (floor == 13) // Test whether floor equals 13
```

You must remember to use == inside tests and to use = outside tests.

Syntax 5.2 Comparisons



The relational operators in Table 1 have a lower precedence than the arithmetic operators. That means you can write arithmetic expressions on either side of the relational operator without using parentheses. For example, in the expression

```
floor - 1 < 13
```

both sides (`floor - 1` and `13`) of the `<` operator are evaluated, and the results are compared. Appendix B shows a table of the Java operators and their precedence.

5.2.2 Comparing Floating-Point Numbers

You have to be careful when comparing floating-point numbers in order to cope with roundoff errors. For example, the following code multiplies the square root of 2 by itself and then subtracts 2.

```
double r = Math.sqrt(2);
double d = r * r - 2;
if (d == 0)
{
    System.out.println("sqrt(2) squared minus 2 is 0");
}
else
{
    System.out.println("sqrt(2) squared minus 2 is not 0 but " + d);
}
```

Even though the laws of mathematics tell us that $(\sqrt{2})^2 - 2$ equals 0, this program fragment prints

```
sqrt(2) squared minus 2 is not 0 but 4.440892098500626E-16
```

Unfortunately, such roundoff errors are unavoidable. It plainly does not make sense in most circumstances to compare floating-point numbers exactly. Instead, test whether they are *close enough*.

To test whether a number x is close to zero, you can test whether the absolute value $|x|$ (that is, the number with its sign removed) is less than a very small threshold number. That threshold value is often called ϵ (the Greek letter epsilon). It is common to set ϵ to 10^{-14} when testing `double` numbers.

Similarly, you can test whether two numbers are approximately equal by checking whether their difference is close to 0.

$$|x - y| \leq \epsilon$$

In Java, we program the test as follows:

```
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
{
    // x is approximately equal to y
}
```

When comparing floating-point numbers, don't test for equality. Instead, check whether they are close enough.

5.2.3 Comparing Strings

To test whether two strings are equal to each other, you must use the method called `equals`:

```
if (string1.equals(string2)) . . .
```

Do not use the `==` operator to compare strings. Use the `equals` method instead.

Do not use the `==` operator to compare strings. The comparison

```
if (string1 == string2) // Not useful
```

has an unrelated meaning. It tests whether the two strings are stored in the same memory location. You can have strings with identical contents stored in different locations, so this test never makes sense in actual programming; see Common Error 5.2 on page 192.

If two strings are not identical, you still may want to know the relationship between them. The `compareTo` method compares strings in **lexicographic order**. This ordering is very similar to the way in which words are sorted in a dictionary. If

```
string1.compareTo(string2) < 0
```

then the string `string1` comes before the string `string2` in the dictionary. For example, this is the case if `string1` is "Harry", and `string2` is "Hello".

Conversely, if

```
string1.compareTo(string2) > 0
```

then `string1` comes after `string2` in dictionary order.

Finally, if

```
string1.compareTo(string2) == 0
```

then `string1` and `string2` are equal.

There are a few technical differences between the ordering in a dictionary and the lexicographic ordering in Java. In Java:

- All uppercase letters come before the lowercase letters. For example, "Z" comes before "a".
- The space character comes before all printable characters.
- Numbers come before letters.
- For the ordering of punctuation marks, see Appendix A.

When comparing two strings, you compare the first letters of each word, then the second letters, and so on, until one of the strings ends or you find the first letter pair that doesn't match.

If one of the strings ends, the longer string is considered the "larger" one. For example, compare "car" with "cart". The first three letters match, and we reach the end of the first string. Therefore "car" comes before "cart" in lexicographic ordering.

When you reach a mismatch, the string containing the "larger" character is considered "larger". For example, compare "cat" with "cart". The first two letters match. Because `t` comes after `r`, the string "cat" comes after "cart" in the lexicographic ordering.

c a r

c a r t

c a t

Letters r comes
match before t

*Lexicographic
Ordering*

*To see which of two terms comes first in the dictionary,
consider the first letter in which they differ.*



5.2.4 Comparing Objects

If you compare two object references with the `==` operator, you test whether the references refer to the same object. Here is an example:

```
Rectangle box1 = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box1;
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

The comparison

```
box1 == box2
```

is true. Both object variables refer to the same object. But the comparison

```
box1 == box3
```

is false. The two object variables refer to different objects (see Figure 3). It does not matter that the objects have identical contents.

You can use the `equals` method to test whether two rectangles have the same contents, that is, whether they have the same upper-left corner and the same width and height. For example, the test

```
box1.equals(box3)
```

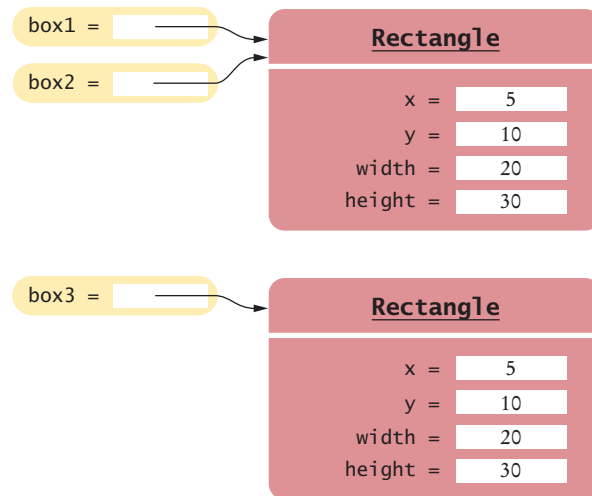
is true.

However, you must be careful when using the `equals` method. It works correctly only if the implementors of the class have supplied it. The `Rectangle` class has an `equals` method that is suitable for comparing rectangles.

For your own classes, you need to supply an appropriate `equals` method. You will learn how to do that in Chapter 9. Until that point, you should not use the `equals` method to compare objects of your own classes.

The `==` operator tests whether two object references are identical. To compare the contents of objects, you need to use the `equals` method.

Figure 3
Comparing Object References



5.2.5 Testing for null

The `null` reference refers to no object.

An object reference can have the special value `null` if it refers to no object at all. It is common to use the `null` value to indicate that a value has never been set. For example,

```
String middleInitial = null; // Not set
if ( . . . )
{
    middleInitial = middleName.substring(0, 1);
}
```

You use the `==` operator (and not `equals`) to test whether an object reference is a `null` reference:

```
if (middleInitial == null)
{
    System.out.println(firstName + " " + lastName);
}
else
{
    System.out.println(firstName + " " + middleInitial + ". " + lastName);
}
```






FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download a program that demonstrates comparisons of numbers and strings

Note that the **`null` reference** is not the same as the empty string `""`. The empty string is a valid string of length 0, whereas a `null` indicates that a string variable refers to no string at all.

Table 2 summarizes how to compare values in Java.

Table 2 Relational Operator Examples

Expression	Value	Comment
<code>3 <= 4</code>	true	3 is less than 4; <code><=</code> tests for “less than or equal”.
 <code>3 <= 4</code>	Error	The “less than or equal” operator is <code><=</code> , not <code>=<</code> . The “less than” symbol comes first.
<code>3 > 4</code>	false	<code>></code> is the opposite of <code><=</code> .
<code>4 < 4</code>	false	The left-hand side must be strictly smaller than the right-hand side.
<code>4 <= 4</code>	true	Both sides are equal; <code><=</code> tests for “less than or equal”.
<code>3 == 5 - 2</code>	true	<code>==</code> tests for equality.
<code>3 != 5 - 1</code>	true	<code>!=</code> tests for inequality. It is true that 3 is not 5 – 1.
 <code>3 = 6 / 2</code>	Error	Use <code>==</code> to test for equality.
<code>1.0 / 3.0 == 0.33333333</code>	false	Although the values are very close to one another, they are not exactly equal. See Section 5.2.2.
 <code>"10" > 5</code>	Error	You cannot compare a string to a number.
<code>"Tomato".substring(0, 3).equals("Tom")</code>	true	Always use the <code>equals</code> method to check whether two strings have the same contents.
<code>"Tomato".substring(0, 3) == ("Tom")</code>	false	Never use <code>==</code> to compare strings; it only checks whether the strings are stored in the same location. See Common Error 5.2 on page 192.



6. Which of the following conditions are true, provided a is 3 and b is 4?
 - a. $a + 1 \leq b$
 - b. $a + 1 \geq b$
 - c. $a + 1 \neq b$
7. Give the opposite of the condition
 $\text{floor} > 13$
8. What is the error in this statement?


```
if (scoreA = scoreB)
{
    System.out.println("Tie");
}
```
9. Supply a condition in this if statement to test whether the user entered a Y:


```
System.out.println("Enter Y to quit.");
String input = in.next();
if ( . . . )
{
    System.out.println("Goodbye.");
}
```
10. Give two ways of testing that a string `str` is the empty string.
11. What is the value of `s.length()` if `s` is
 - a. the empty string `""`?
 - b. the string `" "` containing a space?
 - c. `null`?
12. Which of the following comparisons are syntactically incorrect? Which of them are syntactically correct, but logically questionable?


```
String a = "1";
String b = "one";
double x = 1;
double y = 3 * (1.0 / 3);
```

 - a. `a == "1"`
 - b. `a == null`
 - c. `a.equals("")`
 - d. `a == b`
 - e. `a == x`
 - f. `x == y`
 - g. `x - y == null`
 - h. `x.equals(y)`

Practice It Now you can try these exercises at the end of the chapter: R5.4, R5.7, E5.13.

Common Error 5.2



Using `==` to Compare Strings

If you write

```
if (nickname == "Rob")
```

then the test succeeds only if the variable `nickname` refers to the exact same location as the string literal `"Rob"`.

The test will pass if a string variable was initialized with the same string literal:

```
String nickname = "Rob";
...
if (nickname == "Rob") // Test is true
```

However, if the string with the letters R o b has been assembled in some other way, then the test will fail:

```
String name = "Robert";
String nickname = name.substring(0, 3);
...
if (nickname == "Rob") // Test is false
```

In this case, the substring method produces a string in a different memory location. Even though both strings have the same contents, the comparison fails.

You must remember never to use `==` to compare strings. Always use `equals` to check whether two strings have the same contents.

HOW TO 5.1

Implementing an if Statement



This How To walks you through the process of implementing an if statement. We will illustrate the steps with the following example problem.

Problem Statement The university bookstore has a Kilobyte Day sale every October 24, giving an 8 percent discount on all computer accessory purchases if the price is less than \$128, and a 16 percent discount if the price is at least \$128. Write a program that asks the cashier for the original price and then prints the discounted price.

Step 1 Decide upon the branching condition.

In our sample problem, the obvious choice for the condition is:

original price < 128?

That is just fine, and we will use that condition in our solution.

But you could equally well come up with a correct solution if you choose the opposite condition: Is the original price at least \$128? You might choose this condition if you put yourself into the position of a shopper who wants to know when the bigger discount applies.



Sales discounts are often higher for expensive products. Use the if statement to implement such a decision.

Step 2 Give pseudocode for the work that needs to be done when the condition is true.

In this step, you list the action or actions that are taken in the “positive” branch. The details depend on your problem. You may want to print a message, compute values, or even exit the program.

In our example, we need to apply an 8 percent discount:

discounted price = 0.92 x original price

Step 3 Give pseudocode for the work (if any) that needs to be done when the condition is *not* true.

What do you want to do in the case that the condition of Step 1 is not satisfied? Sometimes, you want to do nothing at all. In that case, use an `if` statement without an `else` branch.

In our example, the condition tested whether the price was less than \$128. If that condition is *not* true, the price is at least \$128, so the higher discount of 16 percent applies to the sale:

discounted price = 0.84 x original price

Step 4 Double-check relational operators.

First, be sure that the test goes in the right *direction*. It is a common error to confuse `>` and `<`. Next, consider whether you should use the `<` operator or its close cousin, the `<=` operator.

What should happen if the original price is exactly \$128? Reading the problem carefully, we find that the lower discount applies if the original price is *less than* \$128, and the higher discount applies when it is *at least* \$128. A price of \$128 should therefore *not* fulfill our condition, and we must use `<`, not `<=`.

Step 5 Remove duplication.

Check which actions are common to both branches, and move them outside. (See Programming Tip 5.4 on page 186.)

In our example, we have two statements of the form

discounted price = ____ x original price

They only differ in the discount rate. It is best to just set the rate in the branches, and to do the computation afterwards:

```

If original price < 128
    discount rate = 0.92
Else
    discount rate = 0.84
discounted price = discount rate x original price
  
```

Step 6 Test both branches.

Formulate two test cases, one that fulfills the condition of the `if` statement, and one that does not. Ask yourself what should happen in each case. Then follow the pseudocode and act each of them out.

In our example, let us consider two scenarios for the original price: \$100 and \$200. We expect that the first price is discounted by \$8, the second by \$32.

When the original price is 100, then the condition `100 < 128` is true, and we get

discount rate = 0.92
discounted price = 0.92 x 100 = 92

When the original price is 200, then the condition `200 < 128` is false, and

discount rate = 0.84
discounted price = 0.84 x 200 = 168

In both cases, we get the expected answer.

Step 7 Assemble the `if` statement in Java.

Type the skeleton

```

if ()
{
}
else
{
}
  
```

```
}
```

and fill it in, as shown in Syntax 5.1 on page 182. Omit the `else` branch if it is not needed.

In our example, the completed statement is

```
if (originalPrice < 128)
{
    discountRate = 0.92;
}
else
{
    discountRate = 0.84;
}
discountedPrice = discountRate * originalPrice;
```

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download the complete program for calculating a discounted price.

WORKED EXAMPLE 5.1

Extracting the Middle

Learn how to extract the middle character from a string, or the two middle characters if the length of the string is even. Go to www.wiley.com/go/javaexamples and download Worked Example 5.1.

c	r	a	t	e
0	1	2	3	4



Computing & Society 5.1 Denver's Luggage Handling System

Making decisions is an essential part of any computer program. Nowhere is this more obvious than in a computer system that helps sort luggage at an airport. After scanning the luggage identification codes, the system sorts the items and routes them to different conveyor belts. Human operators then place the items onto trucks. When the city of Denver built a huge airport to replace an outdated and congested facility, the luggage system contractor went a step further. The new system was designed to replace the human operators with robotic carts. Unfortunately, the system plainly did not work. It was plagued by mechanical problems, such as luggage falling onto the tracks and jamming carts. Equally frustrating were the software glitches. Carts would uselessly accumulate at some locations when they were needed elsewhere.

The airport had been scheduled to open in 1993, but without a functioning luggage system, the opening was delayed for over a year while the contractor tried to fix the problems. The contractor never succeeded, and ultimately a manual system was installed. The delay cost the city and airlines close to a billion dollars, and the contractor, once the leading luggage systems vendor in the United States, went bankrupt.

Clearly, it is very risky to build a large system based on a technology that has never been tried on a smaller scale. As robots and the software that controls them get better over time, they will take on a larger share of luggage handling in the future. But it is likely that this will happen in an incremental fashion.



The Denver airport originally had a fully automatic system for moving luggage, replacing human operators with robotic carts. Unfortunately, the system never worked and was dismantled before the airport was opened.

5.3 Multiple Alternatives

Multiple if statements can be combined to evaluate complex decisions.

In Section 5.1, you saw how to program a two-way branch with an if statement. In many situations, there are more than two cases. In this section, you will see how to implement a decision with multiple alternatives.

For example, consider a program that displays the effect of an earthquake, as measured by the Richter scale (see Table 3).

The 1989 Loma Prieta earthquake that damaged the Bay Bridge in San Francisco and destroyed many buildings measured 7.1 on the Richter scale.



Table 3 Richter Scale

Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

The Richter scale is a measurement of the strength of an earthquake. Every step in the scale, for example from 6.0 to 7.0, signifies a tenfold increase in the strength of the quake.

In this case, there are five branches: one each for the four descriptions of damage, and one for no destruction. Figure 4 shows the flowchart for this multiple-branch statement.

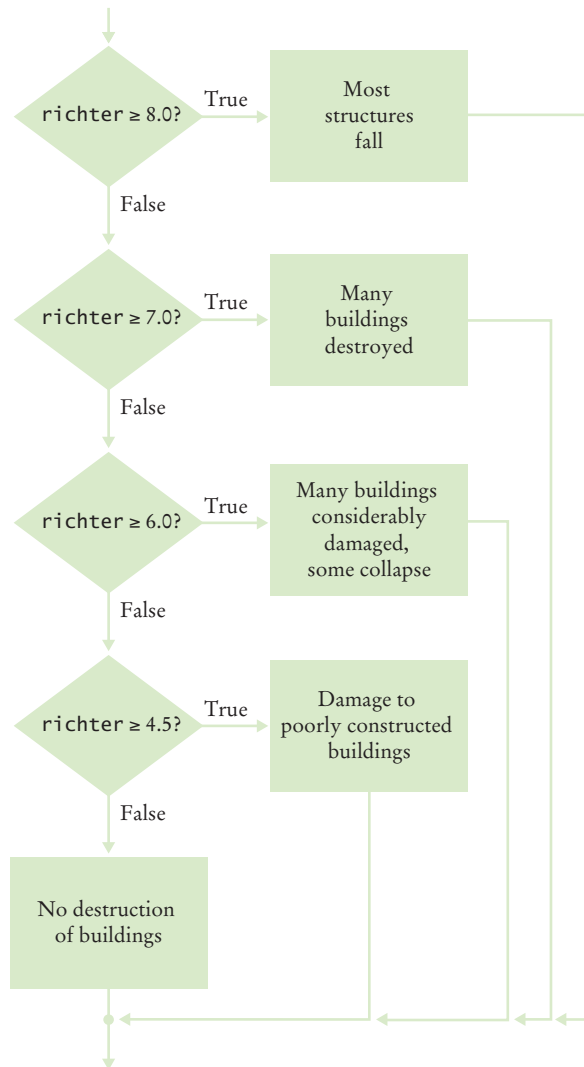
You use multiple if statements to implement multiple alternatives, like this:

```
if (richter >= 8.0)
{
    description = "Most structures fall";
}
else if (richter >= 7.0)
{
    description = "Many buildings destroyed";
}
else if (richter >= 6.0)
{
    description = "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
    description = "Damage to poorly constructed buildings";
}
else
{
    description = "No destruction of buildings";
}
```

As soon as one of the four tests succeeds, the effect is displayed, and no further tests are attempted. If none of the four cases applies, the final else clause applies, and a default message is printed.



Figure 4
Multiple Alternatives



Here you must sort the conditions and test against the largest cutoff first. Suppose we reverse the order of tests:

```

if (richter >= 4.5) // Tests in wrong order
{
    description = "Damage to poorly constructed buildings";
}
else if (richter >= 6.0)
{
    description = "Many buildings considerably damaged, some collapse";
}
else if (richter >= 7.0)
{
    description = "Many buildings destroyed";
}

```

When using multiple if statements, test general conditions after more specific conditions.

```
}
else if (richter >= 8.0)
{
    description = "Most structures fall";
}
```

This does not work. Suppose the value of `richter` is 7.1. That value is at least 4.5, matching the first case. The other tests will never be attempted.

The remedy is to test the more specific conditions first. Here, the condition `richter >= 8.0` is more specific than the condition `richter >= 7.0`, and the condition `richter >= 4.5` is more general (that is, fulfilled by more values) than either of the first two.

In this example, it is also important that we use an if/else if/else sequence, not just multiple independent if statements. Consider this sequence of independent tests.

```
if (richter >= 8.0) // Didn't use else
{
    description = "Most structures fall";
}
if (richter >= 7.0)
{
    description = "Many buildings destroyed";
}
if (richter >= 6.0)
{
    description = "Many buildings considerably damaged, some collapse";
}
if (richter >= 4.5)
{
    "Damage to poorly constructed buildings";
}
```

Now the alternatives are no longer exclusive. If `richter` is 7.1, then the last *three* tests all match. The `description` variable is set to three different strings, ending up with the wrong one.

SELF CHECK



13. In a game program, the scores of players A and B are stored in variables `scoreA` and `scoreB`. Assuming that the player with the larger score wins, write an if/else if/else sequence that prints out "A won", "B won", or "Game tied".
14. Write a conditional statement with three branches that sets `s` to 1 if `x` is positive, to -1 if `x` is negative, and to 0 if `x` is zero.
15. How could you achieve the task of Self Check 14 with only two branches?
16. Beginners sometimes write statements such as the following:

```
if (price > 100)
{
    discountedPrice = price - 20;
}
else if (price <= 100)
{
    discountedPrice = price - 10;
}
```

Explain how this code can be improved.

17. Suppose the user enters -1 into the earthquake program. What is printed?

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download the program for printing earthquake descriptions.

18. Suppose we want to have the earthquake program check whether the user entered a negative number. What branch would you add to the if statement, and where?

Practice It Now you can try these exercises at the end of the chapter: R5.22, E5.10, E5.24.

Special Topic 5.2



The switch Statement

An if/else if/else sequence that compares a *value* against several alternatives can be implemented as a switch statement. For example,

```
int digit = . . . ;
switch (digit)
{
    case 1: digitName = "one"; break;
    case 2: digitName = "two"; break;
    case 3: digitName = "three"; break;
    case 4: digitName = "four"; break;
    case 5: digitName = "five"; break;
    case 6: digitName = "six"; break;
    case 7: digitName = "seven"; break;
    case 8: digitName = "eight"; break;
    case 9: digitName = "nine"; break;
    default: digitName = ""; break;
}
```

This is a shortcut for

```
int digit = . . . ;
if (digit == 1) { digitName = "one"; }
else if (digit == 2) { digitName = "two"; }
else if (digit == 3) { digitName = "three"; }
else if (digit == 4) { digitName = "four"; }
else if (digit == 5) { digitName = "five"; }
else if (digit == 6) { digitName = "six"; }
else if (digit == 7) { digitName = "seven"; }
else if (digit == 8) { digitName = "eight"; }
else if (digit == 9) { digitName = "nine"; }
else { digitName = ""; }
```

It isn't much of a shortcut, but it has one advantage—it is obvious that all branches test the *same* value, namely digit.

The switch statement can be applied only in narrow circumstances. The values in the case clauses must be constants. They can be integers or characters. As of Java 7, strings are permitted as well. You cannot use a switch statement to branch on floating-point values.

Every branch of the switch should be terminated by a break instruction. If the break is missing, execution *falls through* to the next branch, and so on, until a break or the end of the switch is reached. In practice, this fall-through behavior is rarely useful, but it is a common cause of errors. If you accidentally forget a break statement, your program compiles but executes unwanted code. Many programmers consider the switch statement somewhat dangerous and prefer the if statement.

We leave it to you to use the switch statement for your own code or not. At any rate, you need to have a reading knowledge of switch in case you find it in other programmers' code.



The switch statement lets you choose from a fixed set of alternatives.

5.4 Nested Branches

When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.

It is often necessary to include an if statement inside another. Such an arrangement is called a *nested* set of statements.

Here is a typical example: In the United States, different tax rates are used depending on the taxpayer's marital status. There are different tax schedules for single and for married taxpayers. Married taxpayers add their income together and pay taxes on the total. Table 4 gives the tax rate computations, using a simplification of the schedules that were in effect for the 2008 tax year. A different tax rate applies to each "bracket". In this schedule, the income in the first bracket is taxed at 10 percent, and the income in the second bracket is taxed at 25 percent. The income limits for each bracket depend on the marital status.

Table 4 Federal Tax Rate Schedule

If your status is Single and if the taxable income is	the tax is	of the amount over
at most \$32,000	10%	\$0
over \$32,000	$\$3,200 + 25\%$	\$32,000
If your status is Married and if the taxable income is	the tax is	of the amount over
at most \$64,000	10%	\$0
over \$64,000	$\$6,400 + 25\%$	\$64,000

Nested decisions are required for problems that have two levels of decision making.

Now compute the taxes due, given a marital status and an income figure. The key point is that there are two *levels* of decision making. First, you must branch on the marital status. Then, for each marital status, you must have another branch on income level.

The two-level decision process is reflected in two levels of if statements in the program at the end of this section. (See Figure 5 for a flowchart.) In theory, nesting can go deeper than two levels. A three-level decision process (first by state, then by marital status, then by income level) requires three nesting levels.



Computing income taxes requires multiple levels of decisions.



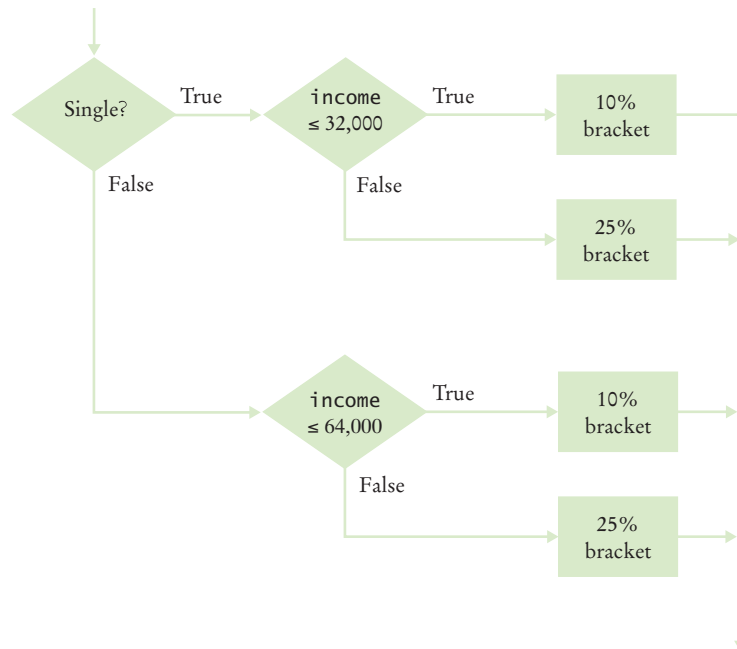


Figure 5 Income Tax Computation

section_4/TaxReturn.java

```

1  /**
2   A tax return of a taxpayer in 2008.
3   */
4  public class TaxReturn
5  {
6      public static final int SINGLE = 1;
7      public static final int MARRIED = 2;
8
9      private static final double RATE1 = 0.10;
10     private static final double RATE2 = 0.25;
11     private static final double RATE1_SINGLE_LIMIT = 32000;
12     private static final double RATE1_MARRIED_LIMIT = 64000;
13
14     private double income;
15     private int status;
16
17     /**
18      Constructs a TaxReturn object for a given income and
19      marital status.
20      @param anIncome the taxpayer income
21      @param aStatus either SINGLE or MARRIED
22      */
23     public TaxReturn(double anIncome, int aStatus)
24     {
25         income = anIncome;
26         status = aStatus;
27     }
28
29     public double getTax()
30     {

```



```

31     double tax1 = 0;
32     double tax2 = 0;
33
34     if (status == SINGLE)
35     {
36         if (income <= RATE1_SINGLE_LIMIT)
37         {
38             tax1 = RATE1 * income;
39         }
40         else
41         {
42             tax1 = RATE1 * RATE1_SINGLE_LIMIT;
43             tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
44         }
45     }
46     else
47     {
48         if (income <= RATE1_MARRIED_LIMIT)
49         {
50             tax1 = RATE1 * income;
51         }
52         else
53         {
54             tax1 = RATE1 * RATE1_MARRIED_LIMIT;
55             tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
56         }
57     }
58
59     return tax1 + tax2;
60 }
61 }

```

section_4/TaxCalculator.java

```

1  import java.util.Scanner;
2
3  /**
4   * This program calculates a simple tax return.
5   */
6  public class TaxCalculator
7  {
8      public static void main(String[] args)
9      {
10         Scanner in = new Scanner(System.in);
11
12         System.out.print("Please enter your income: ");
13         double income = in.nextDouble();
14
15         System.out.print("Are you married? (Y/N) ");
16         String input = in.next();
17         int status;
18         if (input.equals("Y"))
19         {
20             status = TaxReturn.MARRIED;
21         }
22         else
23         {
24             status = TaxReturn.SINGLE;
25         }
26         TaxReturn aTaxReturn = new TaxReturn(income, status);

```

```

27     System.out.println("Tax: "
28         + aTaxReturn.getTax());
29     }
30 }

```

Program Run

```

Please enter your income: 80000
Are you married? (Y/N) Y
Tax: 10400.0

```

SELF CHECK



19. What is the amount of tax that a single taxpayer pays on an income of \$32,000?
20. Would that amount change if the first nested if statement changed from
`if (income <= RATE1_SINGLE_LIMIT)`
to
`if (income < RATE1_SINGLE_LIMIT)`?
21. Suppose Harry and Sally each make \$40,000 per year. Would they save taxes if they married?
22. How would you modify the `TaxCalculator.java` program in order to check that the user entered a correct value for the marital status (i.e., Y or N)?
23. Some people object to higher tax rates for higher incomes, claiming that you might end up with less money after taxes when you get a raise for working hard. What is the flaw in this argument?

Practice It Now you can try these exercises at the end of the chapter: R5.9, R5.21, E5.14, E5.17.

Programming Tip 5.5



Hand-Tracing

A very useful technique for understanding whether a program works correctly is called *hand-tracing*. You simulate the program's activity on a sheet of paper. You can use this method with pseudocode or Java code.

Get an index card, a cocktail napkin, or whatever sheet of paper is within reach. Make a column for each variable. Have the program code ready. Use a marker, such as a paper clip, to mark the current statement. In your mind, execute statements one at a time. Every time the value of a variable changes, cross out the old value and write the new value below the old one.

For example, let's trace the `getTax` method with the data from the program run above.

When the `TaxReturn` object is constructed, the `income` instance variable is set to 80,000 and `status` is set to `MARRIED`. Then the `getTax` method is called. In lines 31 and 32 of `TaxReturn.java`, `tax1` and `tax2` are initialized to 0.

```

29 public double getTax()
30 {
31     double tax1 = 0;
32     double tax2 = 0;
33 }

```



Hand-tracing helps you understand whether a program works correctly.

income	status	tax1	tax2
80000	MARRIED	0	0

Because status is not SINGLE, we move to the `else` branch of the outer `if` statement (line 46).

```

34  if (status == SINGLE)
35  {
36      if (income <= RATE1_SINGLE_LIMIT)
37      {
38          tax1 = RATE1 * income;
39      }
40      else
41      {
42          tax1 = RATE1 * RATE1_SINGLE_LIMIT;
43          tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
44      }
45  }
46  else
47  {

```

Because income is not ≤ 64000 , we move to the `else` branch of the inner `if` statement (line 52).

```

48      if (income <= RATE1_MARRIED_LIMIT)
49      {
50          tax1 = RATE1 * income;
51      }
52      else
53      {
54          tax1 = RATE1 * RATE1_MARRIED_LIMIT;
55          tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
56      }

```

The values of `tax1` and `tax2` are updated.

```

53  {
54      tax1 = RATE1 * RATE1_MARRIED_LIMIT;
55      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
56  }
57  }

```

income	status	tax1	tax2
80000	MARRIED	Ø	Ø
		6400	4000

Their sum is returned and the method ends.

```

58
59  return tax1 + tax2;
60 }

```

income	status	tax1	tax2	return value
80000	MARRIED	Ø	Ø	
		6400	4000	10400

Because the program trace shows the expected return value (\$10,400), it successfully demonstrates that this test case works correctly.

Common Error 5.3



The Dangling `else` Problem

When an `if` statement is nested inside another `if` statement, the following error may occur.

```

double shippingCharge = 5.00; // $5 inside continental U.S.
if (country.equals("USA"))
    if (state.equals("HI"))
        shippingCharge = 10.00; // Hawaii is more expensive
    else // Pitfall!
        shippingCharge = 20.00; // As are foreign shipments

```

The indentation level seems to suggest that the `else` is grouped with the test `country.equals("USA")`. Unfortunately, that is not the case. The compiler ignores all indentation and matches the `else` with the preceding `if`. That is, the code is actually

```

double shippingCharge = 5.00; // $5 inside continental U.S.
if (country.equals("USA"))
    if (state.equals("HI"))
        shippingCharge = 10.00; // Hawaii is more expensive
    else // Pitfall!
        shippingCharge = 20.00; // As are foreign shipments

```

That isn't what you want. You want to group the `else` with the first `if`.

The ambiguous `else` is called a *dangling else*. You can avoid this pitfall if you always use braces, as recommended in Programming Tip 5.2 on page 184:

```
double shippingCharge = 5.00; // $5 inside continental U.S.
if (country.equals("USA"))
{
    if (state.equals("HI"))
    {
        shippingCharge = 10.00; // Hawaii is more expensive
    }
}
else
{
    shippingCharge = 20.00; // As are foreign shipments
}
```

Special Topic 5.3



Block Scope

A *block* is a sequence of statements that is enclosed in braces. For example, consider this statement:

```
if (status == TAXABLE)
{
    double tax = price * TAX_RATE;
    price = price + tax;
}
```

The highlighted part is a block. You can declare a variable in a block, such as the `tax` variable in this example. Such a variable is only visible inside the block.

```
{
    double tax = price * TAX_RATE; // Variable declared inside a block
    price = price + tax;
}
// You can no longer access the tax variable here
```

In fact, the variable is only created after the program enters the block, and it is removed as soon as the program exits the block. Such a variable is said to have *block scope*. In general, the *scope* of a variable is the part of the program in which the variable can be accessed. A variable with block scope is visible only inside a block.

It is considered good design to minimize the scope of a variable. This reduces the possibility of accidental modification and name conflicts. For example, as long as the `tax` variable is not



In the same way that there can be a street named "Main Street" in different cities, a Java program can have multiple variables with the same name.

needed outside the block, it is a good idea to declare it inside the block. However, if you need the variable outside the block, you must define it outside. For example,

```
double tax = 0;
if (status == TAXABLE)
{
    tax = price * TAX_RATE;
}
price = price + tax;
```

Here, the `tax` variable is used outside the block of the `if` statement, and you must declare it outside.

In Java, the scope of a local variable can never contain the declaration of another local variable with the same name. For example, the following is an error:

```
double tax = 0;
if (status == TAXABLE)
{
    double tax = price * TAX_RATE;
    // Error: Cannot declare another variable with the same name
    price = price + tax;
}
```

However, you can have local variables with identical names if their scopes do not overlap, such as

```
if (Math.random() > 0.5)
{
    Rectangle r = new Rectangle(5, 10, 20, 30);
    . . .
} // Scope of r ends here
else
{
    int r = 5;
    // OK—it is legal to declare another r here
    . . .
}
```

These variables are independent from each other. You can have local variables with the same name, as long as their scopes don't overlap.

Special Topic 5.4



Enumeration Types

In many programs, you use variables that can hold one of a finite number of values. For example, in the tax return class, the `status` instance variable holds one of the values `SINGLE` or `MARRIED`. We arbitrarily declared `SINGLE` as the number 1 and `MARRIED` as 2. If, due to some programming error, the `status` variable is set to another integer value (such as -1, 0, or 3), then the programming logic may produce invalid results.

In a simple program, this is not really a problem. But as programs grow over time, and more cases are added (such as the “married filing separately” status), errors can slip in. Java version 5.0 introduces a remedy: **enumeration types**. An enumeration type has a finite set of values, for example

```
public enum FilingStatus { SINGLE, MARRIED, MARRIED_FILING_SEPARATELY }
```

You can have any number of values, but you must include them all in the `enum` declaration.

You can declare variables of the enumeration type:

```
FilingStatus status = FilingStatus.SINGLE;
```


If you try to assign a value that isn't a `FilingStatus`, such as 2 or "S", then the compiler reports an error.

Use the `==` operator to compare enumeration values, for example:

```
if (status == FilingStatus.SINGLE) . . .
```

Place the enum declaration inside the class that implements your program, such as

```
public class TaxReturn
{
    public enum FilingStatus { SINGLE, MARRIED, MARRIED_FILING_SEPARATELY }
    . . .
}
```

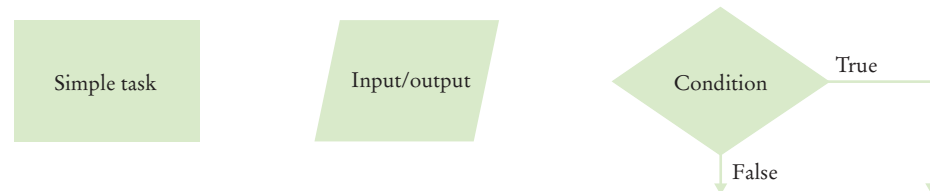
5.5 Problem Solving: Flowcharts

Flow charts are made up of elements for tasks, input/output, and decisions.

You have seen examples of flowcharts earlier in this chapter. A flowchart shows the structure of decisions and tasks that are required to solve a problem. When you have to solve a complex problem, it can help to draw a flowchart to visualize the flow of control.

The basic flowchart elements are shown in Figure 6.

Figure 6
Flowchart Elements



Each branch of a decision can contain tasks and further decisions.

The basic idea is simple enough. Link tasks and input/output boxes in the sequence in which they should be executed. Whenever you need to make a decision, draw a diamond with two outcomes (see Figure 7).

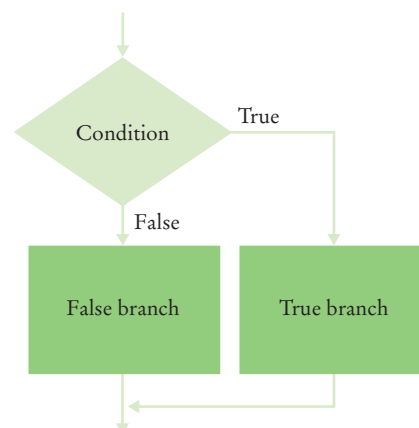
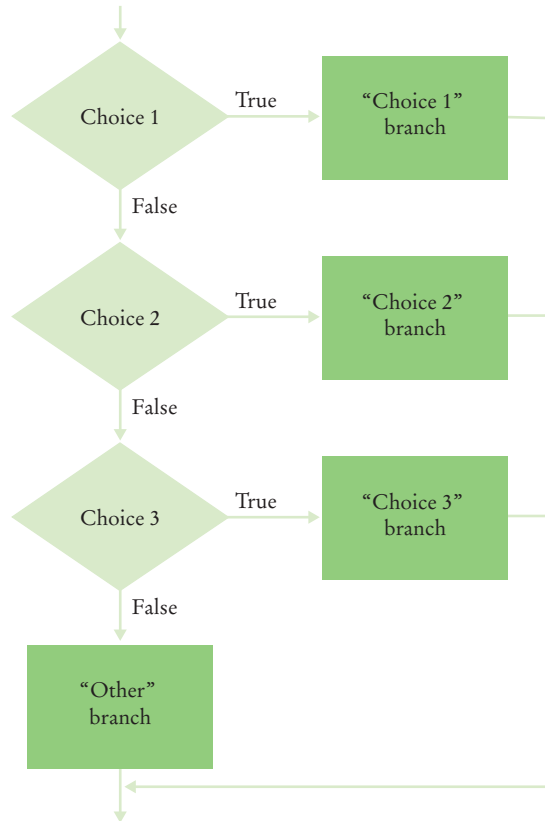


Figure 7
Flowchart with Two Outcomes

Figure 8
Flowchart with Multiple Choices

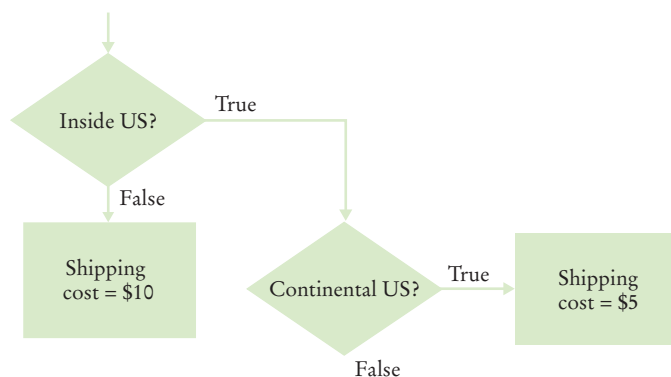


Each branch can contain a sequence of tasks and even additional decisions. If there are multiple choices for a value, lay them out as in Figure 8.

There is one issue that you need to be aware of when drawing flowcharts. Unconstrained branching and merging can lead to “spaghetti code”, a messy network of possible pathways through a program.

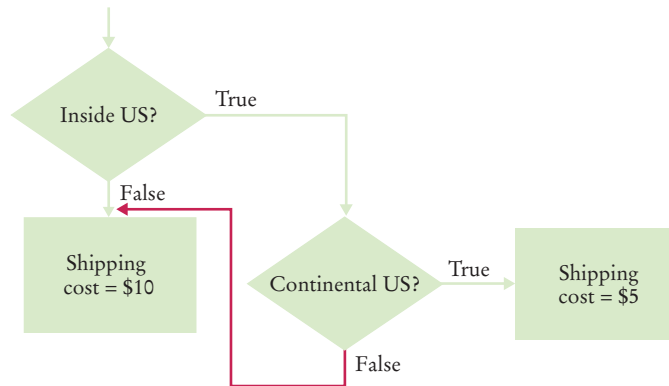
There is a simple rule for avoiding spaghetti code: Never point an arrow *inside* another branch.

To understand the rule, consider this example: Shipping costs are \$5 inside the United States, except that to Hawaii and Alaska they are \$10. International shipping costs are also \$10. You might start out with a flowchart like the following:

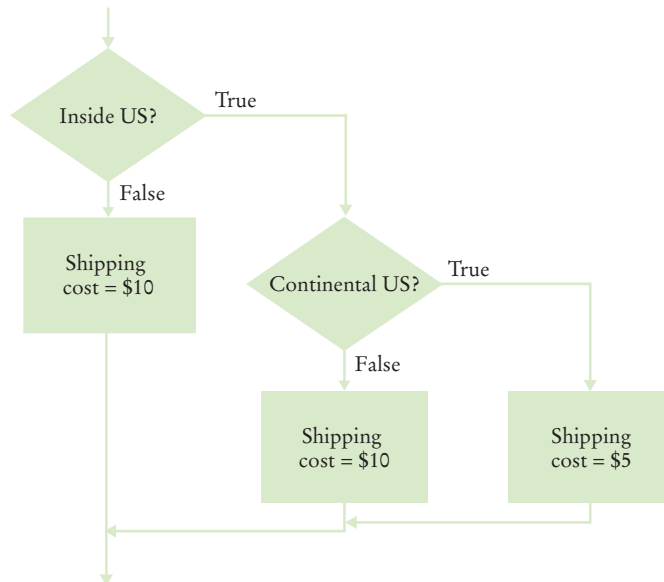


Never point an arrow inside another branch.

Now you may be tempted to reuse the “shipping cost = \$10” task:



Don't do that! The red arrow points inside a different branch. Instead, add another task that sets the shipping cost to \$10, like this:



FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download a program that computes shipping costs.

Not only do you avoid spaghetti code, but it is also a better design. In the future it may well happen that the cost for international shipments is different from that to Alaska and Hawaii.

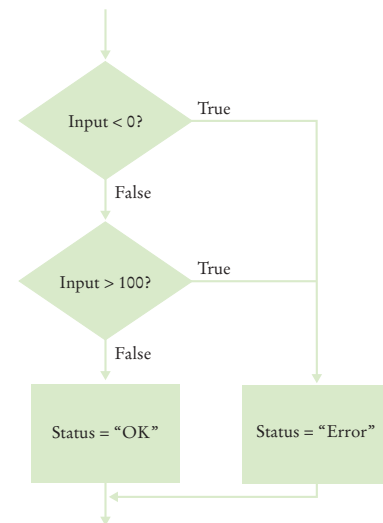
Flowcharts can be very useful for getting an intuitive understanding of the flow of an algorithm. However, they get large rather quickly when you add more details. At that point, it makes sense to switch from flowcharts to pseudocode.



Spaghetti code has so many pathways that it becomes impossible to understand.



24. Draw a flowchart for a program that reads a value temp and prints “Frozen” if it is less than zero.
25. What is wrong with the flowchart at right?
26. How do you fix the flowchart of Self Check 25?
27. Draw a flowchart for a program that reads a value x. If it is less than zero, print “Error”. Otherwise, print its square root.
28. Draw a flowchart for a program that reads a value temp. If it is less than zero, print “Ice”. If it is greater than 100, print “Steam”. Otherwise, print “Liquid”.



Practice It Now you can try these exercises at the end of the chapter: R5.12, R5.13, R5.14.

5.6 Problem Solving: Selecting Test Cases

Black-box testing describes a testing method that does not take the structure of the implementation into account.

White-box testing uses information about the structure of a program.

Code coverage is a measure of how many parts of a program have been tested.

Testing the functionality of a program without consideration of its internal structure is called **black-box testing**. This is an important part of testing, because, after all, the users of a program do not know its internal structure. If a program works perfectly on all inputs, then it surely does its job.

However, it is impossible to ensure absolutely that a program will work correctly on all inputs just by supplying a finite number of test cases. As the famous computer scientist Edsger Dijkstra pointed out, testing can show only the presence of bugs—not their absence. To gain more confidence in the correctness of a program, it is useful to consider its internal structure. Testing strategies that look inside a program are called **white-box testing**. Performing unit tests of each method is a part of white-box testing.

You want to make sure that each part of your program is exercised at least once by one of your test cases. This is called **code coverage**. If some code is never executed by any of your test cases, you have no way of knowing whether that code would perform correctly if it ever were executed by user input. That means that you need to look at every if/else branch to see that each of them is reached by some test case. Many conditional branches are in the code only to take care of strange and abnormal inputs, but they still do something. It is a common phenomenon that they end up doing something incorrectly, but those faults are never discovered during testing, because nobody supplied the strange and abnormal inputs. The remedy is to ensure that each part of the code is covered by some test case.

For example, in testing the `getTax` method of the `TaxReturn` class, you want to make sure that every if statement is entered for at least one test case. You should test both single and married taxpayers, with incomes in each of the three tax brackets.

When you select test cases, you should make it a habit to include **boundary test cases**: legal values that lie at the boundary of the set of acceptable inputs.

Boundary test cases are test cases that are at the boundary of acceptable inputs.

Here is a plan for obtaining a comprehensive set of test cases for the tax program:

- There are two possibilities for the marital status and two tax brackets for each status, yielding four test cases.
- Test a handful of *boundary* conditions, such as an income that is at the boundary between two brackets, and a zero income.
- If you are responsible for error checking (which is discussed in Section 5.8), also test an invalid input, such as a negative income.

Make a list of the test cases and the expected outputs:

Test Case	Married	Expected Output	Comment
30,000	N	3,000	10% bracket
72,000	N	13,200	3,200 + 25% of 40,000
50,000	Y	5,000	10% bracket
104,000	Y	16,400	6,400 + 25% of 40,000
32,000	N	3,200	boundary case
0		0	boundary case

It is a good idea to design test cases before implementing a program.

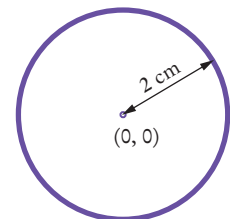
When you develop a set of test cases, it is helpful to have a flowchart of your program (see Section 5.5). Check off each branch that has a test case. Include test cases for the boundary cases of each decision. For example, if a decision checks whether an input is less than 100, test with an input of 100.

It is always a good idea to design test cases *before* starting to code. Working through the test cases gives you a better understanding of the algorithm that you are about to implement.

SELF CHECK



- Using Figure 1 on page 181 as a guide, follow the process described in this section to design a set of test cases for the `ElevatorSimulation.java` program in Section 5.1.
- What is a boundary test case for the algorithm in How To 5.1 on page 193? What is the expected output?
- Using Figure 4 on page 197 as a guide, follow the process described in Section 5.6 to design a set of test cases for the `Earthquake.java` program in Section 5.3.
- Suppose you are designing a part of a program for a medical robot that has a sensor returning an x - and y -location (measured in cm). You need to check whether the sensor location is inside the circle, outside the circle, or on the boundary (specifically, having a distance of less than 1 mm from the boundary). Assume the circle has center $(0, 0)$ and a radius of 2 cm. Give a set of test cases.



Practice It Now you can try these exercises at the end of the chapter: R5.15, R5.16.

Programming Tip 5.6

**Make a Schedule and Make Time for Unexpected Problems**

Commercial software is notorious for being delivered later than promised. For example, Microsoft originally promised that its Windows Vista operating system would be available late in 2003, then in 2005, then in March 2006; it finally was released in January 2007. Some of the early promises might not have been realistic. It was in Microsoft's interest to let prospective customers expect the imminent availability of the product. Had customers known the actual delivery date, they might have switched to a different product in the meantime. Undeniably, though, Microsoft had not anticipated the full complexity of the tasks it had set itself to solve.

Microsoft can delay the delivery of its product, but it is likely that you cannot. As a student or a programmer, you are expected to manage your time wisely and to finish your assignments on time. You can probably do simple programming exercises the night before the due date, but an assignment that looks twice as hard may well take four times as long, because more things can go wrong. You should therefore make a schedule whenever you start a programming project.

First, estimate realistically how much time it will take you to:

- Design the program logic.
- Develop test cases.
- Type the program in and fix syntax errors.
- Test and debug the program.

For example, for the income tax program I might estimate an hour for the design; 30 minutes for developing test cases; an hour for data entry and fixing syntax errors; and an hour for testing and debugging. That is a total of 3.5 hours. If I work two hours a day on this project, it will take me almost two days.

Then think of things that can go wrong. Your computer might break down. You might be stumped by a problem with the computer system. (That is a particularly important concern for beginners. It is *very* common to lose a day over a trivial problem just because it takes time to track down a person who knows the magic command to overcome it.) As a rule of thumb, *double* the time of your estimate. That is, you should start four days, not two days, before the due date. If nothing went wrong, great; you have the program done two days early. When the inevitable problem occurs, you have a cushion of time that protects you from embarrassment and failure.



Make a schedule for your programming work and build in time for problems.

Special Topic 5.5

**Logging**

Sometimes you run a program and you are not sure where it spends its time. To get a printout of the program flow, you can insert **trace messages** into the program, such as this one:

```
if (status == SINGLE)
{
    System.out.println("status is SINGLE");
    . . .
}
```

However, there is a problem with using `System.out.println` for trace messages. When you are done testing the program, you need to remove all print statements that produce trace messages. If you find another error, however, you need to stick the print statements back in.

To overcome this problem, you should use the `Logger` class, which allows you to turn off the trace messages without removing them from the program.

Instead of printing directly to `System.out`, use the global logger object that is returned by the call `Logger.getGlobal()`. (Prior to Java 7, you obtained the global logger as `Logger.getLogger("global")`.) Then call the `info` method:

```
Logger.getGlobal().info("status is SINGLE");
```

By default, the message is printed. But if you call

```
Logger.getGlobal().setLevel(Level.OFF);
```

at the beginning of the main method of your program, all log message printing is suppressed. Set the level to `Level.INFO` to turn logging of info messages on again. Thus, you can turn off the log messages when your program works fine, and you can turn them back on if you find another error. In other words, using `Logger.getGlobal().info` is just like `System.out.println`, except that you can easily activate and deactivate the logging.

The `Logger` class has many other options for industrial-strength logging. Check out the API documentation if you want to have more control over logging.

Logging messages can be deactivated when testing is complete.

5.7 Boolean Variables and Operators

The Boolean type `boolean` has two values, `false` and `true`.



A Boolean variable is also called a *flag* because it can be either up (*true*) or down (*false*).

Sometimes, you need to evaluate a logical condition in one part of a program and use it elsewhere. To store a condition that can be true or false, you use a *Boolean variable*. Boolean variables are named after the mathematician George Boole (1815–1864), a pioneer in the study of logic.

In Java, the `boolean` data type has exactly two values, denoted `false` and `true`. These values are not strings or integers; they are special values, just for Boolean variables. Here is a declaration of a Boolean variable:

```
boolean failed = true;
```

You can use the value later in your program to make a decision:

```
if (failed) // Only executed if failed has been set to true
{
    . . .
}
```

When you make complex decisions, you often need to combine Boolean values. An operator that combines Boolean conditions is called a **Boolean operator**. In Java, the `&&` operator (called *and*) yields true only when both conditions are true. The `||` operator (called *or*) yields the result true if at least one of the conditions is true.

A	B	A && B	A	B	A B	A	!A
true	true	true	true	true	true	true	false
true	false	false	true	false	true	false	true
false	true	false	false	true	true		
false	false	false	false	false	false		

Figure 9 Boolean Truth Tables

At this geyser in Iceland, you can see ice, liquid water, and steam.



Java has two Boolean operators that combine conditions: `&&` (*and*) and `||` (*or*).

Suppose you write a program that processes temperature values, and you want to test whether a given temperature corresponds to liquid water. (At sea level, water freezes at 0 degrees Celsius and boils at 100 degrees.) Water is liquid if the temperature is greater than zero *and* less than 100:

```
if (temp > 0 && temp < 100) { System.out.println("Liquid"); }
```

The condition of the test has two parts, joined by the `&&` operator. Each part is a Boolean value that can be true or false. The combined expression is true if both individual expressions are true. If either one of the expressions is false, then the result is also false (see Figure 9).

The Boolean operators `&&` and `||` have a lower precedence than the relational operators. For that reason, you can write relational expressions on either side of the Boolean operators without using parentheses. For example, in the expression

```
temp > 0 && temp < 100
```

the expressions `temp > 0` and `temp < 100` are evaluated first. Then the `&&` operator combines the results. Appendix B shows a table of the Java operators and their precedence.

Conversely, let's test whether water is *not* liquid at a given temperature. That is the case when the temperature is at most 0 *or* at least 100.

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download a program comparing numbers using Boolean expressions.

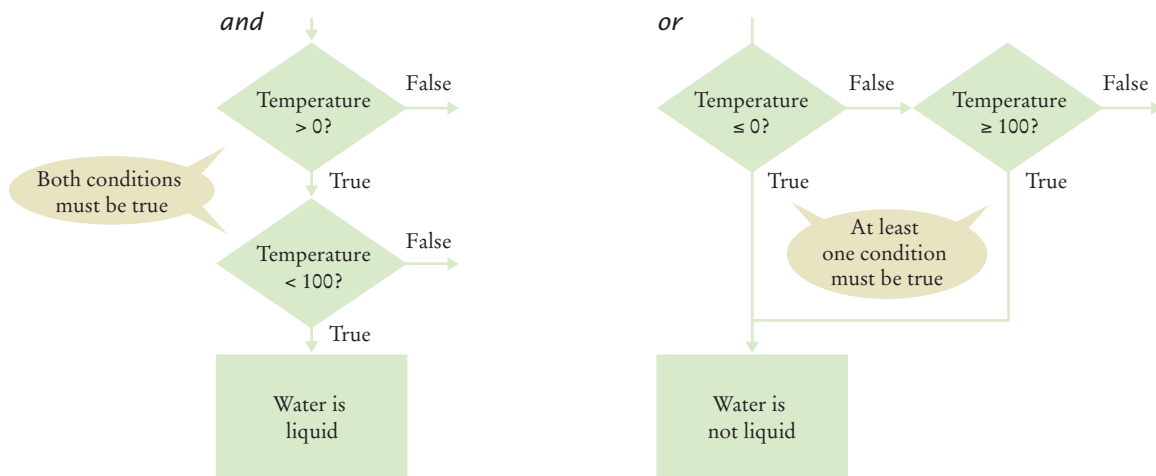




Figure 10 Flowcharts for *and* and *or* Combinations

Table 5 Boolean Operator Examples

Expression	Value	Comment
<code>0 < 200 && 200 < 100</code>	false	Only the first condition is true.
<code>0 < 200 200 < 100</code>	true	The first condition is true.
<code>0 < 200 100 < 200</code>	true	The <code> </code> is not a test for “either-or”. If both conditions are true, the result is true.
<code>0 < x && x < 100 x == -1</code>	<code>(0 < x && x < 100) x == -1</code>	The <code>&&</code> operator has a higher precedence than the <code> </code> operator (see Appendix B).
 <code>0 < x < 100</code>	Error	Error: This expression does not test whether <code>x</code> is between 0 and 100. The expression <code>0 < x</code> is a Boolean value. You cannot compare a Boolean value with the integer 100.
 <code>x && y > 0</code>	Error	Error: This expression does not test whether <code>x</code> and <code>y</code> are positive. The left-hand side of <code>&&</code> is an integer, <code>x</code> , and the right-hand side, <code>y > 0</code> , is a Boolean value. You cannot use <code>&&</code> with an integer argument.
<code>!(0 < 200)</code>	false	<code>0 < 200</code> is true, therefore its negation is false.
<code>frozen == true</code>	frozen	There is no need to compare a Boolean variable with true.
<code>frozen == false</code>	!frozen	It is clearer to use <code>!</code> than to compare with false.

Use the `||` (*or*) operator to combine the expressions:

```
if (temp <= 0 || temp >= 100) { System.out.println("Not liquid"); }
```

Figure 10 shows flowcharts for these examples.

Sometimes you need to *invert* a condition with the *not* Boolean operator. The `!` operator takes a single condition and evaluates to true if that condition is false and to false if the condition is true. In this example, output occurs if the value of the Boolean variable `frozen` is false: .

```
if (!frozen) { System.out.println("Not frozen"); }
```

Table 5 illustrates additional examples of evaluating Boolean operators.



33. Suppose `x` and `y` are two integers. How do you test whether both of them are zero?
34. How do you test whether at least one of them is zero?
35. How do you test whether *exactly one of them* is zero?
36. What is the value of `!!frozen`?
37. What is the advantage of using the type `boolean` rather than strings `"false"/"true"` or integers `0/1`?

Practice It Now you can try these exercises at the end of the chapter: R5.29, E5.22, E5.23.

To invert a condition, use the `!` (*not*) operator.

Common Error 5.4

**Combining Multiple Relational Operators**

Consider the expression

```
if (0 <= temp <= 100) // Error
```

This looks just like the mathematical test $0 \leq \text{temp} \leq 100$. But in Java, it is a compile-time error.

Let us dissect the condition. The first half, $0 \leq \text{temp}$, is a test with an outcome true or false. The outcome of that test (true or false) is then compared against 100. This seems to make no sense. Is true larger than 100 or not? Can one compare truth values and numbers? In Java, you cannot. The Java compiler rejects this statement.

Instead, use `&&` to combine two separate tests:

```
if (0 <= temp && temp <= 100) . . .
```

Another common error, along the same lines, is to write

```
if (input == 1 || 2) . . . // Error
```

to test whether input is 1 or 2. Again, the Java compiler flags this construct as an error. You cannot apply the `||` operator to numbers. You need to write two Boolean expressions and join them with the `||` operator:

```
if (input == 1 || input == 2) . . .
```

Common Error 5.5

**Confusing `&&` and `||` Conditions**

It is a surprisingly common error to confuse *and* and *or* conditions. A value lies between 0 and 100 if it is at least 0 *and* at most 100. It lies outside that range if it is less than 0 *or* greater than 100. There is no golden rule; you just have to think carefully.

Often the *and* or *or* is clearly stated, and then it isn't too hard to implement it. But sometimes the wording isn't as explicit. It is quite common that the individual conditions are nicely set apart in a bulleted list, but with little indication of how they should be combined.

Consider these instructions for filing a tax return. You can claim single filing status if any one of the following is true:

- You were never married.
- You were legally separated or divorced on the last day of the tax year.
- You were widowed, and did not remarry.

Because the test passes if *any one* of the conditions is true, you must combine the conditions with *or*.

Elsewhere, the same instructions state that you may use the more advantageous status of married filing jointly if all five of the following conditions are true:

- Your spouse died less than two years ago and you did not remarry.
- You have a child whom you can claim as dependent.
- That child lived in your home for all of the tax year.
- You paid over half the cost of keeping up your home for this child.
- You filed a joint return with your spouse the year he or she died.

Because *all* of the conditions must be true for the test to pass, you must combine them with an *and*.

Special Topic 5.6

**Short-Circuit Evaluation of Boolean Operators**

The `&&` and `||` operators are computed using short-circuit evaluation. In other words, logical expressions are evaluated from left to right, and evaluation stops as soon as the truth value is determined. When an `&&` is evaluated and the first condition is false, the second condition is not evaluated, because it does not matter what the outcome of the second test is.

For example, consider the expression

```
quantity > 0 && price / quantity < 10
```

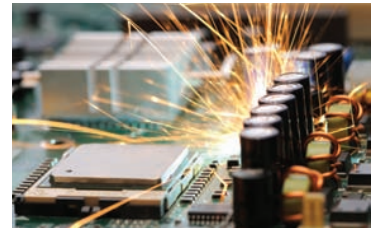
Suppose the value of `quantity` is zero. Then the test `quantity > 0` fails, and the second test is not attempted. That is just as well, because it is illegal to divide by zero.

Similarly, when the first condition of an `||` expression is true, then the remainder is not evaluated because the result must be true.

This process is called **short-circuit evaluation**.

The `&&` and `||` operators are computed using *short-circuit evaluation*: As soon as the truth value is determined, no further conditions are evaluated.

In a short circuit, electricity travels along the path of least resistance. Similarly, short-circuit evaluation takes the fastest path for computing the result of a Boolean expression.



Special Topic 5.7

**De Morgan's Law**

Humans generally have a hard time comprehending logical conditions with *not* operators applied to *and/or* expressions. **De Morgan's Law**, named after the logician Augustus De Morgan (1806–1871), can be used to simplify these Boolean expressions.

Suppose we want to charge a higher shipping rate if we don't ship within the continental United States:

```
if (!(country.equals("USA") && !state.equals("AK") && !state.equals("HI")))
{
    shippingCharge = 20.00;
}
```

This test is a little bit complicated, and you have to think carefully through the logic. When it is *not* true that the country is USA *and* the state is not Alaska *and* the state is not Hawaii, then charge \$20.00. Huh? It is not true that some people won't be confused by this code.

The computer doesn't care, but it takes human programmers to write and maintain the code. Therefore, it is useful to know how to simplify such a condition.

De Morgan's Law has two forms: one for the negation of an *and* expression and one for the negation of an *or* expression:

```
!(A && B)    is the same as    !A || !B
!(A || B)    is the same as    !A && !B
```

De Morgan's Law tells you how to negate `&&` and `||` conditions.

Pay particular attention to the fact that the *and* and *or* operators are *reversed* by moving the *not* inward. For example, the negation of “the state is Alaska *or* it is Hawaii”,

```
!(state.equals("AK") || state.equals("HI"))
```

is “the state is not Alaska *and* it is not Hawaii”:

```
!state.equals("AK") && !state.equals("HI")
```

Now apply the law to our shipping charge computation:

```
!(country.equals("USA")
  && !state.equals("AK")
  && !state.equals("HI"))
```

is equivalent to

```
!country.equals("USA")
|| !!state.equals("AK")
|| !!state.equals("HI"))
```

Because two ! cancel each other out, the result is the simpler test

```
!country.equals("USA")
|| state.equals("AK")
|| state.equals("HI")
```

In other words, higher shipping charges apply when the destination is outside the United States or to Alaska or Hawaii.

To simplify conditions with negations of *and* or *or* expressions, it is usually a good idea to apply De Morgan's Law to move the negations to the innermost level.

5.8 Application: Input Validation



Like a quality control worker, you want to make sure that user input is correct before processing it.

An important application for the `if` statement is *input validation*. Whenever your program accepts user input, you need to make sure that the user-supplied values are valid before you use them in your computations.

Consider our elevator simulation program. Assume that the elevator panel has buttons labeled 1 through 20 (but not 13). The following are illegal inputs:

- The number 13
- Zero or a negative number
- A number larger than 20
- An input that is not a sequence of digits, such as five

In each of these cases, we want to give an error message and exit the program.

It is simple to guard against an input of 13:

```
if (floor == 13)
{
    System.out.println("Error: There is no thirteenth floor.");
}
```

Here is how you ensure that the user doesn't enter a number outside the valid range:

```
if (floor <= 0 || floor > 20)
{
    System.out.println("Error: The floor must be between 1 and 20.");
}
```

However, dealing with an input that is not a valid integer is a more serious problem. When the statement

```
floor = in.nextInt();
```

is executed, and the user types in an input that is not an integer (such as five), then the integer variable `floor` is not set. Instead, a run-time exception occurs and the program is terminated. To avoid this problem, you should first call the `hasNextInt` method

Call the `hasNextInt` or `hasNextDouble` method to ensure that the next input is a number.

which checks whether the next input is an integer. If that method returns true, you can safely call `nextInt`. Otherwise, print an error message and exit the program:

```
if (in.hasNextInt())
{
    int floor = in.nextInt();
    Process the input value.
}
else
{
    System.out.println("Error: Not an integer.");
}
```

Here is the complete elevator simulation program with input validation:

section_8/ElevatorSimulation2.java

```
1  import java.util.Scanner;
2
3  /**
4   This program simulates an elevator panel that skips the 13th floor, checking for
5   input errors.
6   */
7  public class ElevatorSimulation2
8  {
9      public static void main(String[] args)
10     {
11         Scanner in = new Scanner(System.in);
12         System.out.print("Floor: ");
13         if (in.hasNextInt())
14         {
15             // Now we know that the user entered an integer
16
17             int floor = in.nextInt();
18
19             if (floor == 13)
20             {
21                 System.out.println("Error: There is no thirteenth floor.");
22             }
23             else if (floor <= 0 || floor > 20)
24             {
25                 System.out.println("Error: The floor must be between 1 and 20.");
26             }
27             else
28             {
29                 // Now we know that the input is valid
30
31                 int actualFloor = floor;
32                 if (floor > 13)
33                 {
34                     actualFloor = floor - 1;
35                 }
36
37                 System.out.println("The elevator will travel to the actual floor "
38                     + actualFloor);
39             }
40         }
41         else
42         {
43             System.out.println("Error: Not an integer.");
44         }
45     }
46 }
```

```

44     }
45 }
46 }

```

Program Run

```

Floor: 13
Error: There is no thirteenth floor.

```

SELF CHECK

38. In the `ElevatorSimulation2` program, what is the output when the input is
- 100?
 - 1?
 - 20?
 - thirteen?
39. Your task is to rewrite lines 19–26 of the `ElevatorSimulation2` program so that there is a single `if` statement with a complex condition. What is the condition?
- ```

if (. . .)
{
 System.out.println("Error: Invalid floor number");
}

```
40. In the Sherlock Holmes story “The Adventure of the Sussex Vampire”, the inimitable detective uttered these words: “Matilda Briggs was not the name of a young woman, Watson, ... It was a ship which is associated with the giant rat of Sumatra, a story for which the world is not yet prepared.” Over a hundred years later, researchers found giant rats in Western New Guinea, another part of Indonesia.

Suppose you are charged with writing a program that processes rat weights. It contains the statements

```

System.out.print("Enter weight in kg: ");
double weight = in.nextDouble();

```

What input checks should you supply?



*When processing inputs, you want to reject values that are too large. But how large is too large? These giant rats, found in Western New Guinea, are about five times the size of a city rat.*

41. Run the following test program and supply inputs 2 and three at the prompts. What happens? Why?

```

import java.util.Scanner

public class Test
{
 public static void main(String[] args)
 {
 Scanner in = new Scanner(System.in);
 System.out.print("Enter an integer: ");
 int m = in.nextInt();
 System.out.print("Enter another integer: ");
 int n = in.nextInt();
 System.out.println(m + " " + n);
 }
}

```

**Practice It** Now you can try these exercises at the end of the chapter: R5.3, R5.32, E5.12.





## Computing & Society 5.2 Artificial Intelligence

When one uses a sophisticated computer program such as a tax preparation package, one is bound to attribute some intelligence to the computer. The computer asks sensible questions and makes computations that we find a mental challenge. After all, if doing one's taxes were easy, we wouldn't need a computer to do it for us.

As programmers, however, we know that all this apparent intelligence is an illusion. Human programmers have carefully "coached" the software in all possible scenarios, and it simply replays the actions and decisions that were programmed into it.

Would it be possible to write computer programs that are genuinely intelligent in some sense? From the earliest days of computing, there was a sense that the human brain might be nothing but an immense computer, and that it might well be feasible to program computers to imitate some processes of human thought. Serious research into *artificial intelligence* began in the mid-1950s, and the first twenty years brought some impressive successes. Programs that play chess—surely an activity that appears to require remarkable intellectual powers—have become so good that they now routinely beat all but the best human players. As far back as 1975, an *expert-system* program called Mycin gained fame for being better in diagnosing meningitis in patients than the average physician.

However, there were serious setbacks as well. From 1982 to 1992, the Japanese government embarked on a massive research project, funded at over 40 billion Japanese yen. It was known as the *Fifth-Generation Project*. Its goal was to develop new hardware and software to greatly improve the performance of expert system software. At its outset, the project created fear in other countries that the Japanese computer industry was about to become the undisputed leader in the field. However, the end results were disappointing and did little to bring

artificial intelligence applications to market.

From the very outset, one of the stated goals of the AI community was to produce software that could translate text from one language to another, for example from English to Russian. That undertaking proved to be enormously complicated. Human language appears to be much more subtle and interwoven with the human experience than had originally been thought. Even the grammar-checking tools that come with word-processing programs today are more of a gimmick than a useful tool, and analyzing grammar is just the first step in translating sentences.

The CYC (from encyclopedia) project, started by Douglas Lenat in 1984, tries to codify the implicit assumptions that underlie human speech and writing. The team members started out analyzing news articles and asked themselves what unmentioned facts are necessary to actually understand the sentences. For example, consider the sentence, "Last fall she enrolled in Michigan State". The reader automatically realizes that "fall" is not related to falling down in this context, but refers to the season. While there is a state of Michigan, here Michigan State denotes the university. A priori, a computer program has none of this

knowledge. The goal of the CYC project is to extract and store the requisite facts—that is, (1) people enroll in universities; (2) Michigan is a state; (3) many states have universities named X State University, often abbreviated as X State; (4) most people enroll in a university in the fall. By 1995, the project had codified about 100,000 common-sense concepts and about a million facts of knowledge relating them. Even this massive amount of data has not proven sufficient for useful applications.

In recent years, artificial intelligence technology has seen substantial advances. One of the most astounding examples is the outcome of a series of "grand challenges" for autonomous vehicles posed by the Defense Advanced Research Projects Agency (DARPA). Competitors were invited to submit a computer-controlled vehicle that had to complete an obstacle course without a human driver or remote control. The first event, in 2004, was a disappointment, with none of the entrants finishing the route. In 2005, five vehicles completed a grueling 212 km course in the Mojave desert. Stanford's Stanley came in first, with an average speed of 30 km/h. In 2007, DARPA moved the competition to an "urban" environment, an abandoned air force base. Vehicles

had to be able to interact with each other, following California traffic laws. As Stanford's Sebastian Thrun explained: "In the last Grand Challenge, it didn't really matter whether an obstacle was a rock or a bush, because either way you'd just drive around it. The current challenge is to move from just sensing the environment to understanding it."



Winner of the 2007 DARPA Urban Challenge

## CHAPTER SUMMARY

**Use the if statement to implement a decision.**

- The if statement allows a program to carry out different actions depending on the nature of the data to be processed.

**Implement comparisons of numbers and objects.**

- Use relational operators (< <= > >= == !=) to compare numbers.
- Relational operators compare values. The == operator tests for equality.
- When comparing floating-point numbers, don't test for equality. Instead, check whether they are close enough.
- Do not use the == operator to compare strings. Use the equals method instead.
- The compareTo method compares strings in lexicographic order.
- The == operator tests whether two object references are identical. To compare the contents of objects, you need to use the equals method.
- The null reference refers to no object.

**Implement complex decisions that require multiple if statements.**

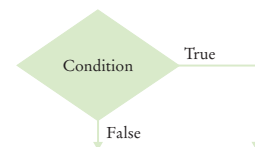
- Multiple if statements can be combined to evaluate complex decisions.
- When using multiple if statements, test general conditions after more specific conditions.

**Implement decisions whose branches require further decisions.**

- When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.
- Nested decisions are required for problems that have two levels of decision making.

**Draw flowcharts for visualizing the control flow of a program.**

- Flow charts are made up of elements for tasks, input/output, and decisions.
- Each branch of a decision can contain tasks and further decisions.
- Never point an arrow inside another branch.



**Design test cases for your programs.**

- Black-box testing describes a testing method that does not take the structure of the implementation into account.
- White-box testing uses information about the structure of a program.
- Code coverage is a measure of how many parts of a program have been tested.
- Boundary test cases are test cases that are at the boundary of acceptable inputs.
- It is a good idea to design test cases before implementing a program.
- Logging messages can be deactivated when testing is complete.

**Use the Boolean data type to store and combine conditions that can be true or false.**

- The Boolean type `boolean` has two values, `false` and `true`.
- Java has two Boolean operators that combine conditions: `&&` (*and*) and `||` (*or*).
- To invert a condition, use the `!` (*not*) operator.
- The `&&` and `||` operators are computed using **short-circuit evaluation**: As soon as the truth value is determined, no further conditions are evaluated.
- De Morgan's Law tells you how to negate `&&` and `||` conditions.

**Apply if statements to detect whether user input is valid.**

- Call the `hasNextInt` or `hasNextDouble` method to ensure that the next input is a number.

**STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER**

```
java.awt.Rectangle
equals
java.lang.String
equals
compareTo
java.util.Scanner
hasNextDouble
hasNextInt
```

```
java.util.logging.Level
INFO
OFF
java.util.logging.Logger
getGlobal
info
setLevel
```

**REVIEW QUESTIONS****■ R5.1** What is the value of each variable after the if statement?

- `int n = 1; int k = 2; int r = n;`  
if (`k < n`) { `r = k;` }
- `int n = 1; int k = 2; int r;`  
if (`n < k`) { `r = k;` }  
else { `r = k + n;` }
- `int n = 1; int k = 2; int r = k;`  
if (`r < k`) { `n = r;` }  
else { `k = n;` }
- `int n = 1; int k = 2; int r = 3;`  
if (`r < n + k`) { `r = 2 * n;` }  
else { `k = 2 * r;` }

- **R5.2** Explain the difference between

```
s = 0;
if (x > 0) { s++; }
if (y > 0) { s++; }
```

and

```
s = 0;
if (x > 0) { s++; }
else if (y > 0) { s++; }
```

- **R5.3** Find the errors in the following if statements.

- a. `if x > 0 then System.out.print(x);`
- b. `if (1 + x > Math.pow(x, Math.sqrt(2))) { y = y + x; }`
- c. `if (x = 1) { y++; }`
- d. `x = in.nextInt();`  
`if (in.hasNextInt())`  
`{`  
`sum = sum + x;`  
`}`  
`else`  
`{`  
`System.out.println("Bad input for x");`  
`}`
- e. `String letterGrade = "F";`  
`if (grade >= 90) { letterGrade = "A"; }`  
`if (grade >= 80) { letterGrade = "B"; }`  
`if (grade >= 70) { letterGrade = "C"; }`  
`if (grade >= 60) { letterGrade = "D"; }`

- **R5.4** What do these code fragments print?

- a. `int n = 1;`  
`int m = -1;`  
`if (n < -m) { System.out.print(n); }`  
`else { System.out.print(m); }`
- b. `int n = 1;`  
`int m = -1;`  
`if (-n >= m) { System.out.print(n); }`  
`else { System.out.print(m); }`
- c. `double x = 0;`  
`double y = 1;`  
`if (Math.abs(x - y) < 1) { System.out.print(x); }`  
`else { System.out.print(y); }`
- d. `double x = Math.sqrt(2);`  
`double y = 2;`  
`if (x * x == y) { System.out.print(x); }`  
`else { System.out.print(y); }`

- **R5.5** Suppose `x` and `y` are variables of type `double`. Write a code fragment that sets `y` to `x` if `x` is positive and to 0 otherwise.
- **R5.6** Suppose `x` and `y` are variables of type `double`. Write a code fragment that sets `y` to the absolute value of `x` without calling the `Math.abs` function. Use an `if` statement.
- **R5.7** Explain why it is more difficult to compare floating-point numbers than integers. Write Java code to test whether an integer `n` equals 10 and whether a floating-point number `x` is approximately equal to 10.

- **R5.8** It is easy to confuse the = and == operators. Write a test program containing the statement

```
if (floor = 13)
```

What error message do you get? Write another test program with the statement

```
count == 0;
```

What does your compiler do when you compile the program?

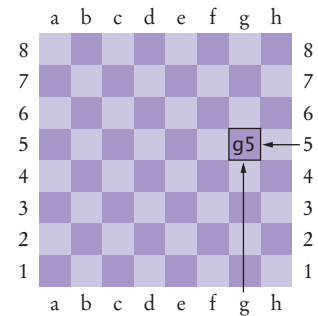
- **R5.9** Each square on a chess board can be described by a letter and number, such as g5 in the example at right.

The following pseudocode describes an algorithm that determines whether a square with a given letter and number is dark (black) or light (white).

```

If the letter is an a, c, e, or g
 If the number is odd
 color = "black"
 Else
 color = "white"
Else
 If the number is even
 color = "black"
 Else
 color = "white"

```



Using the procedure in Programming Tip 5.5, trace this pseudocode with input g5.

- **Testing R5.10** Give a set of four test cases for the algorithm of Exercise R5.9 that covers all branches.

- **R5.11** In a scheduling program, we want to check whether two appointments overlap. For simplicity, appointments start at a full hour, and we use military time (with hours 0–24). The following pseudocode describes an algorithm that determines whether the appointment with start time **start1** and end time **end1** overlaps with the appointment with start time **start2** and end time **end2**.

```

If start1 > start2
 s = start1
Else
 s = start2
If end1 < end2
 e = end1
Else
 e = end2
If s < e
 The appointments overlap.
Else
 The appointments don't overlap.

```

Trace this algorithm with an appointment from 10–12 and one from 11–13, then with an appointment from 10–11 and one from 12–13.

- **R5.12** Draw a flow chart for the algorithm in Exercise R5.11.
- **R5.13** Draw a flow chart for the algorithm in Exercise E5.13.

- **R5.14** Draw a flow chart for the algorithm in Exercise E5.14.
- ■ **Testing R5.15** Develop a set of test cases for the algorithm in Exercise R5.11.
- ■ **Testing R5.16** Develop a set of test cases for the algorithm in Exercise E5.14.
- ■ **R5.17** Write pseudocode for a program that prompts the user for a month and day and prints out whether it is one of the following four holidays:
  - New Year's Day (January 1)
  - Independence Day (July 4)
  - Veterans Day (November 11)
  - Christmas Day (December 25)
- ■ **R5.18** Write pseudocode for a program that assigns letter grades for a quiz, according to the following table:
 

| Score  | Grade |
|--------|-------|
| 90-100 | A     |
| 80-89  | B     |
| 70-79  | C     |
| 60-69  | D     |
| < 60   | F     |
- ■ **R5.19** Explain how the lexicographic ordering of strings in Java differs from the ordering of words in a dictionary or telephone book. *Hint:* Consider strings such as IBM, wiley.com, Century 21, and While-U-Wait.
- ■ **R5.20** Of the following pairs of strings, which comes first in lexicographic order?
  - a. "Tom", "Jerry"
  - b. "Tom", "Tomato"
  - c. "church", "Churchill"
  - d. "car manufacturer", "carburetor"
  - e. "Harry", "hairy"
  - f. "Java", " Car"
  - g. "Tom", "Tom"
  - h. "Car", "Car1"
  - i. "car", "bar"
- **R5.21** Explain the difference between an if/else if/else sequence and nested if statements. Give an example of each.
- ■ **R5.22** Give an example of an if/else if/else sequence where the order of the tests does not matter. Give an example where the order of the tests matters.
- **R5.23** Rewrite the condition in Section 5.3 to use < operators instead of >= operators. What is the impact on the order of the comparisons?
- ■ **Testing R5.24** Give a set of test cases for the tax program in Exercise P5.2. Manually compute the expected results.
- **R5.25** Make up a Java code example that shows the dangling else problem using the following statement: A student with a GPA of at least 1.5, but less than 2, is on probation. With less than 1.5, the student is failing.

- ■ ■ **R5.26** Complete the following truth table by finding the truth values of the Boolean expressions for all combinations of the Boolean inputs *p*, *q*, and *r*.

| <i>p</i>            | <i>q</i> | <i>r</i> | $(p \ \&\& \ q) \    \ !r$ | $!(p \ \&\& \ (q \    \ !r))$ |
|---------------------|----------|----------|----------------------------|-------------------------------|
| false               | false    | false    |                            |                               |
| false               | false    | true     |                            |                               |
| false               | true     | false    |                            |                               |
| ...                 |          |          |                            |                               |
| 5 more combinations |          |          |                            |                               |
| ...                 |          |          |                            |                               |

- ■ ■ **R5.27** True or false?  $A \ \&\& \ B$  is the same as  $B \ \&\& \ A$  for any Boolean conditions *A* and *B*.
- **R5.28** The “advanced search” feature of many search engines allows you to use Boolean operators for complex queries, such as “(cats OR dogs) AND NOT pets”. Contrast these search operators with the Boolean operators in Java.
- ■ **R5.29** Suppose the value of *b* is false and the value of *x* is 0. What is the value of each of the following expressions?
- |                                |                                |
|--------------------------------|--------------------------------|
| <b>a.</b> $b \ \&\& \ x == 0$  | <b>e.</b> $b \ \&\& \ x != 0$  |
| <b>b.</b> $b \    \ x == 0$    | <b>f.</b> $b \    \ x != 0$    |
| <b>c.</b> $!b \ \&\& \ x == 0$ | <b>g.</b> $!b \ \&\& \ x != 0$ |
| <b>d.</b> $!b \    \ x == 0$   | <b>h.</b> $!b \    \ x != 0$   |
- ■ **R5.30** Simplify the following expressions. Here, *b* is a variable of type boolean.
- |                               |
|-------------------------------|
| <b>a.</b> $b == \text{true}$  |
| <b>b.</b> $b == \text{false}$ |
| <b>c.</b> $b != \text{true}$  |
| <b>d.</b> $b != \text{false}$ |
- ■ ■ **R5.31** Simplify the following statements. Here, *b* is a variable of type boolean and *n* is a variable of type int.
- |                                                                                 |
|---------------------------------------------------------------------------------|
| <b>a.</b> <code>if (n == 0) { b = true; } else { b = false; }</code>            |
| (Hint: What is the value of $n == 0$ ?)                                         |
| <b>b.</b> <code>if (n == 0) { b = false; } else { b = true; }</code>            |
| <b>c.</b> <code>b = false; if (n &gt; 1) { if (n &lt; 2) { b = true; } }</code> |
| <b>d.</b> <code>if (n &lt; 1) { b = true; } else { b = n &gt; 2; }</code>       |
- **R5.32** What is wrong with the following program?
- ```

System.out.print("Enter the number of quarters: ");
int quarters = in.nextInt();
if (in.hasNextInt())
{
    total = total + quarters * 0.25;
    System.out.println("Total: " + total);
}

```



```

    }
    else
    {
        System.out.println("Input error.");
    }
}

```

PRACTICE EXERCISES

- **E5.1** Write a program that reads an integer and prints whether it is negative, zero, or positive.
- **E5.2** Write a program that reads a floating-point number and prints “zero” if the number is zero. Otherwise, print “positive” or “negative”. Add “small” if the absolute value of the number is less than 1, or “large” if it exceeds 1,000,000.
- **E5.3** Write a program that reads an integer and prints how many digits the number has, by checking whether the number is ≥ 10 , ≥ 100 , and so on. (Assume that all integers are less than ten billion.) If the number is negative, first multiply it with -1 .
- **E5.4** Write a program that reads three numbers and prints “all the same” if they are all the same, “all different” if they are all different, and “neither” otherwise.
- **E5.5** Write a program that reads three numbers and prints “increasing” if they are in increasing order, “decreasing” if they are in decreasing order, and “neither” otherwise. Here, “increasing” means “strictly increasing”, with each value larger than its predecessor. The sequence 3 4 4 would not be considered increasing.
- **E5.6** Repeat Exercise E5.5, but before reading the numbers, ask the user whether increasing/decreasing should be “strict” or “lenient”. In lenient mode, the sequence 3 4 4 is increasing and the sequence 4 4 4 is both increasing and decreasing.
- **E5.7** Write a program that reads in three integers and prints “in order” if they are sorted in ascending *or* descending order, or “not in order” otherwise. For example,

```

1 2 5   in order
1 5 2   not in order
5 2 1   in order
1 2 2   in order

```

- **E5.8** Write a program that reads four integers and prints “two pairs” if the input consists of two matching pairs (in some order) and “not two pairs” otherwise. For example,

```

1 2 2 1   two pairs
1 2 2 3   not two pairs
2 2 2 2   two pairs

```

- **Business E5.9** Write a program that reads in the name and salary of an employee. Here the salary will denote an *hourly* wage, such as \$9.25. Then ask how many hours the employee worked in the past week. Be sure to accept fractional hours. Compute the pay. Any overtime work (over 40 hours per week) is paid at 150 percent of the regular wage. Print a paycheck for the employee. In your solution, implement a class `Paycheck`.
- **E5.10** Write a program that reads a temperature value and the letter C for Celsius or F for Fahrenheit. Print whether water is liquid, solid, or gaseous at the given temperature at sea level.

- **E5.11** The boiling point of water drops by about one degree centigrade for every 300 meters (or 1,000 feet) of altitude. Improve the program of Exercise E5.10 to allow the user to supply the altitude in meters or feet.
- **E5.12** Add error handling to Exercise E5.11. If the user does not enter a number when expected, or provides an invalid unit for the altitude, print an error message and end the program.
- **E5.13** When two points in time are compared, each given as hours (in military time, ranging from 0 and 23) and minutes, the following pseudocode determines which comes first.

```

If hour1 < hour2
    time1 comes first.
Else if hour1 and hour2 are the same
    If minute1 < minute2
        time1 comes first.
    Else if minute1 and minute2 are the same
        time1 and time2 are the same.
    Else
        time2 comes first.
Else
    time2 comes first.

```

Write a program that prompts the user for two points in time and prints the time that comes first, then the other time. In your program, supply a class `Time` and a method

```
public int compareTo(Time other)
```

that returns `-1` if the time comes before the other, `0` if both are the same, and `1` otherwise.

- **E5.14** The following algorithm yields the season (Spring, Summer, Fall, or Winter) for a given month and day.

```

If month is 1, 2, or 3, season = "Winter"
Else if month is 4, 5, or 6, season = "Spring"
Else if month is 7, 8, or 9, season = "Summer"
Else if month is 10, 11, or 12, season = "Fall"
If month is divisible by 3 and day >= 21
    If season is "Winter", season = "Spring"
    Else if season is "Spring", season = "Summer"
    Else if season is "Summer", season = "Fall"
    Else season = "Winter"

```



Write a program that prompts the user for a month and day and then prints the season, as determined by this algorithm. Use a class `Date` with a method `getSeason`.

- **E5.15** Write a program that translates a letter grade into a number grade. Letter grades are A, B, C, D, and F, possibly followed by `+` or `-`. Their numeric values are 4, 3, 2, 1, and 0. There is no `F+` or `F-`. A `+` increases the numeric value by 0.3, a `-` decreases it by 0.3. However, an `A+` has value 4.0.

```

Enter a letter grade: B-
The numeric value is 2.7.

```

Use a class `Grade` with a method `getNumericGrade`.

- ■ **E5.16** Write a program that translates a number between 0 and 4 into the closest letter grade. For example, the number 2.8 (which might have been the average of several grades) would be converted to B−. Break ties in favor of the better grade; for example 2.85 should be a B.

Use a class `Grade` with a method `getNumericGrade`.

- ■ **E5.17** The original U.S. income tax of 1913 was quite simple. The tax was

- 1 percent on the first \$50,000.
- 2 percent on the amount over \$50,000 up to \$75,000.
- 3 percent on the amount over \$75,000 up to \$100,000.
- 4 percent on the amount over \$100,000 up to \$250,000.
- 5 percent on the amount over \$250,000 up to \$500,000.
- 6 percent on the amount over \$500,000.

There was no separate schedule for single or married taxpayers. Write a program that computes the income tax according to this schedule.

- ■ **E5.18** Write a program that takes user input describing a playing card in the following shorthand notation:

A	Ace
2 ... 10	Card values
J	Jack
Q	Queen
K	King
D	Diamonds
H	Hearts
S	Spades
C	Clubs

Your program should print the full description of the card. For example,

```
Enter the card notation: QS
Queen of Spades
```

Implement a class `Card` whose constructor takes the card notation string and whose `getDescription` method returns a description of the card. If the notation string is not in the correct format, the `getDescription` method should return the string "Unknown".

- ■ **E5.19** Write a program that reads in three floating-point numbers and prints the largest of the three inputs. For example:

```
Please enter three numbers: 4 9 2.5
The largest number is 9.
```

- ■ **E5.20** Write a program that reads in three strings and sorts them lexicographically.

```
Enter three strings: Charlie Able Baker
Able
Baker
Charlie
```

- ■ **E5.21** Write a program that reads in two floating-point numbers and tests whether they are the same up to two decimal places. Here are two sample runs.

Enter two floating-point numbers: 2.0 1.99998
 They are the same up to two decimal places.
 Enter two floating-point numbers: 2.0 1.98999
 They are different.

- **E5.22** Write a program that prompts the user to provide a single character from the alphabet. Print Vowel or Consonant, depending on the user input. If the user input is not a letter (between a and z or A and Z), or is a string of length > 1, print an error message.

- ■ **E5.23** Write a program that asks the user to enter a month (1 for January, 2 for February, etc.) and then prints the number of days in the month. For February, print “28 days”.

Enter a month: 5
 30 days

Use a class Month with a method

```
public int getLength()
```

Do not use a separate if/else branch for each month. Use Boolean operators.

- **Business E5.24** A supermarket awards coupons depending on how much a customer spends on groceries. For example, if you spend \$50, you will get a coupon worth eight percent of that amount. The following table shows the percent used to calculate the coupon awarded for different amounts spent. Write a program that calculates and prints the value of the coupon a person can receive based on groceries purchased.

Here is a sample run:

Please enter the cost of your groceries: 14
 You win a discount coupon of \$ 1.12. (8% of your purchase)

Money Spent	Coupon Percentage
Less than \$10	No coupon
From \$10 to \$60	8%
More than \$60 to \$150	10%
More than \$150 to \$210	12%
More than \$210	14%

PROGRAMMING PROJECTS

- ■ **P5.1** Write a program that prompts for the day and month of the user’s birthday and then prints a horoscope. Make up fortunes for programmers, like this:



Please enter your birthday (month and day): 6 16
 Gemini are experts at figuring out the behavior of complicated programs.
 You feel where bugs are coming from and then stay one step ahead. Tonight,
 your style wins approval from a tough critic.

Each fortune should contain the name of the astrological sign. (You will find the names and date ranges of the signs at a distressingly large number of sites on the Internet.) Use a class Date with a method getFortune.

- ■ **P5.2** Write a program that computes taxes for the following schedule.

If your status is Single and if the taxable income is over	but not over	the tax is	of the amount over
\$0	\$8,000	10%	\$0
\$8,000	\$32,000	\$800 + 15%	\$8,000
\$32,000		\$4,400 + 25%	\$32,000
If your status is Married and if the taxable income is over	but not over	the tax is	of the amount over
\$0	\$16,000	10%	\$0
\$16,000	\$64,000	\$1,600 + 15%	\$16,000
\$64,000		\$8,800 + 25%	\$64,000

- ■ ■ **P5.3** The `TaxReturn.java` program uses a simplified version of the 2008 U.S. income tax schedule. Look up the tax brackets and rates for the current year, for both single and married filers, and implement a program that computes the actual income tax.

- ■ ■ **P5.4** *Unit conversion.* Write a unit conversion program that asks the users from which unit they want to convert (fl. oz, gal, oz, lb, in, ft, mi) and to which unit they want to convert (ml, l, g, kg, mm, cm, m, km). Reject incompatible conversions (such as gal \rightarrow km). Ask for the value to be converted, then display the result:

```
Convert from? gal
Convert to? ml
Value? 2.5
2.5 gal = 9462.5 ml
```

- ■ ■ **P5.5** A year with 366 days is called a leap year. Leap years are necessary to keep the calendar synchronized with the sun because the earth revolves around the sun once every 365.25 days. Actually, that figure is not entirely precise, and for all dates after 1582 the *Gregorian correction* applies. Usually years that are divisible by 4 are leap years, for example 1996. However, years that are divisible by 100 (for example, 1900) are not leap years, but years that are divisible by 400 are leap years (for example, 2000). Write a program that asks the user for a year and computes whether that year is a leap year. Provide a class `Year` with a method `isLeapYear`. Use a single `if` statement and Boolean operators.

- ■ ■ **P5.6** *Roman numbers.* Write a program that converts a positive integer into the Roman number system. The Roman number system has digits

```
I      1
V      5
X     10
L     50
C    100
D    500
M   1,000
```

Numbers are formed according to the following rules:

- a. Only numbers up to 3,999 are represented.
- b. As in the decimal system, the thousands, hundreds, tens, and ones are expressed separately.
- c. The numbers 1 to 9 are expressed as

I	1	VI	6
II	2	VII	7
III	3	VIII	8
IV	4	IX	9
V	5		



As you can see, an I preceding a V or X is subtracted from the value, and you can never have more than three I's in a row.

- d. Tens and hundreds are done the same way, except that the letters X, L, C and C, D, M are used instead of I, V, X, respectively.

Your program should take an input, such as 1978, and convert it to Roman numerals, MCMLXXVIII.

- ■ ■ **P5.7** French country names are feminine when they end with the letter e, masculine otherwise, except for the following which are masculine even though they end with e:

- le Belize
- le Cambodge
- le Mexique
- le Mozambique
- le Zaïre
- le Zimbabwe

Write a program that reads the French name of a country and adds the article: le for masculine or la for feminine, such as le Canada or la Belgique.

However, if the country name starts with a vowel, use l'; for example, l'Afghanistan.

For the following plural country names, use les:

- les Etats-Unis
- les Pays-Bas

- ■ ■ **Business P5.8** Write a program to simulate a bank transaction. There are two bank accounts: checking and savings. First, ask for the initial balances of the bank accounts; reject negative balances. Then ask for the transactions; options are deposit, withdrawal, and transfer. Then ask for the account; options are checking and savings. Reject transactions that overdraw an account. At the end, print the balances of both accounts.

- ■ **Business P5.9** When you use an automated teller machine (ATM) with your bank card, you need to use a personal identification number (PIN) to access your account. If a user fails more than three times when entering the PIN, the machine will block the card. Assume that the user's PIN is "1234" and write a program that asks the user for the PIN no more than three times, and does the following:

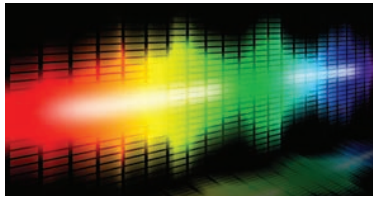


- If the user enters the right number, print a message saying, “Your PIN is correct”, and end the program.
- If the user enters a wrong number, print a message saying, “Your PIN is incorrect” and, if you have asked for the PIN less than three times, ask for it again.
- If the user enters a wrong number three times, print a message saying “Your bank card is blocked” and end the program.

■ **Business P5.10** Calculating the tip when you go to a restaurant is not difficult, but your restaurant wants to suggest a tip according to the service diners receive. Write a program that calculates a tip according to the diner’s satisfaction as follows:

- Ask for the diners’ satisfaction level using these ratings: 1 = Totally satisfied, 2 = Satisfied, 3 = Dissatisfied.
- If the diner is totally satisfied, calculate a 20 percent tip.
- If the diner is satisfied, calculate a 15 percent tip.
- If the diner is dissatisfied, calculate a 10 percent tip.
- Report the satisfaction level and tip in dollars and cents.

■ **Science P5.11** Write a program that prompts the user for a wavelength value and prints a description of the corresponding part of the electromagnetic spectrum, as given in the following table.



Electromagnetic Spectrum		
Type	Wavelength (m)	Frequency (Hz)
Radio Waves	$> 10^{-1}$	$< 3 \times 10^9$
Microwaves	10^{-3} to 10^{-1}	3×10^9 to 3×10^{11}
Infrared	7×10^{-7} to 10^{-3}	3×10^{11} to 4×10^{14}
Visible light	4×10^{-7} to 7×10^{-7}	4×10^{14} to 7.5×10^{14}
Ultraviolet	10^{-8} to 4×10^{-7}	7.5×10^{14} to 3×10^{16}
X-rays	10^{-11} to 10^{-8}	3×10^{16} to 3×10^{19}
Gamma rays	$< 10^{-11}$	$> 3 \times 10^{19}$

■ **Science P5.12** Repeat Exercise P5.11, modifying the program so that it prompts for the frequency instead.

■ ■ **Science P5.13** Repeat Exercise P5.11, modifying the program so that it first asks the user whether the input will be a wavelength or a frequency.

■ ■ ■ **Science P5.14** A minivan has two sliding doors. Each door can be opened by either a dashboard switch, its inside handle, or its outside handle. However, the inside handles do not work if a child lock switch is activated. In order for the sliding doors to open, the gear shift must be in park, *and* the master unlock switch must be activated. (This book’s author is the long-suffering owner of just such a vehicle.)



Your task is to simulate a portion of the control software for the vehicle. The input is a sequence of values for the switches and the gear shift, in the following order:

- Dashboard switches for left and right sliding door, child lock, and master unlock (0 for off or 1 for activated)
- Inside and outside handles on the left and right sliding doors (0 or 1)
- The gear shift setting (one of P N D 1 2 3 R).

A typical input would be 0 0 0 1 0 1 0 0 P.

Print “left door opens” and/or “right door opens” as appropriate. If neither door opens, print “both doors stay closed”.

■ **Science P5.15** Sound level L in units of decibel (dB) is determined by

$$L = 20 \log_{10}(p/p_0)$$

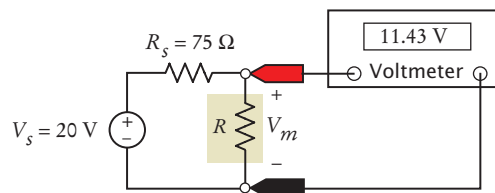
where p is the sound pressure of the sound (in Pascals, abbreviated Pa), and p_0 is a reference sound pressure equal to 20×10^{-6} Pa (where L is 0 dB). The following table gives descriptions for certain sound levels.



Threshold of pain	130 dB
Possible hearing damage	120 dB
Jack hammer at 1 m	100 dB
Traffic on a busy roadway at 10 m	90 dB
Normal conversation	60 dB
Calm library	30 dB
Light leaf rustling	0 dB

Write a program that reads a value and a unit, either dB or Pa, and then prints the closest description from the list above.

■ ■ **Science P5.16** The electric circuit shown below is designed to measure the temperature of the gas in a chamber.



The resistor R represents a temperature sensor enclosed in the chamber. The resistance R , in Ω , is related to the temperature T , in $^{\circ}\text{C}$, by the equation

$$R = R_0 + kT$$

In this device, assume $R_0 = 100 \Omega$ and $k = 0.5$. The voltmeter displays the value of the voltage, V_m , across the sensor. This voltage V_m indicates the temperature, T , of the gas according to the equation

$$T = \frac{R}{k} - \frac{R_0}{k} = \frac{R_s}{k} \frac{V_m}{V_s - V_m} - \frac{R_0}{k}$$

Suppose the voltmeter voltage is constrained to the range $V_{\min} = 12 \text{ volts} \leq V_m \leq V_{\max} = 18 \text{ volts}$. Write a program that accepts a value of V_m and checks that it's between 12 and 18. The program should return the gas temperature in degrees Celsius when V_m is between 12 and 18 and an error message when it isn't.

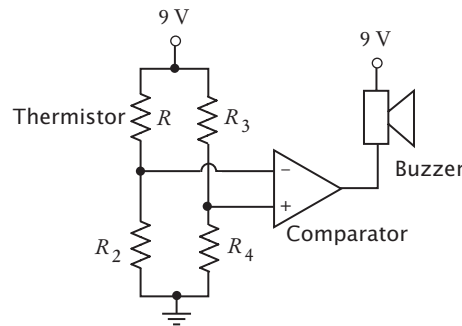
■ ■ ■ Science P5.17



Crop damage due to frost is one of the many risks confronting farmers. The figure below shows a simple alarm circuit designed to warn of frost. The alarm circuit uses a device called a thermistor to sound a buzzer when the temperature drops below freezing. Thermistors are semiconductor devices that exhibit a temperature dependent resistance described by the equation

$$R = R_0 e^{\beta \left(\frac{1}{T} - \frac{1}{T_0} \right)}$$

where R is the resistance, in Ω , at the temperature T , in $^{\circ}\text{K}$, and R_0 is the resistance, in Ω , at the temperature T_0 , in $^{\circ}\text{K}$. β is a constant that depends on the material used to make the thermistor.



The circuit is designed so that the alarm will sound when

$$\frac{R_2}{R + R_2} < \frac{R_4}{R_3 + R_4}$$

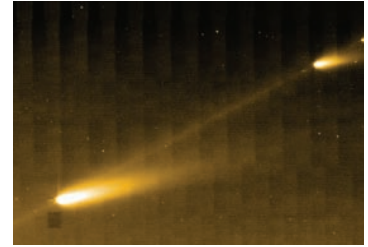
The thermistor used in the alarm circuit has $R_0 = 33,192 \Omega$ at $T_0 = 40^{\circ}\text{C}$, and $\beta = 3,310^{\circ}\text{K}$. (Notice that β has units of $^{\circ}\text{K}$. The temperature in $^{\circ}\text{K}$ is obtained by adding 273° to the temperature in $^{\circ}\text{C}$.) The resistors R_2 , R_3 , and R_4 have a resistance of $156.3 \text{ k}\Omega = 156,300 \Omega$.

Write a Java program that prompts the user for a temperature in $^{\circ}\text{F}$ and prints a message indicating whether or not the alarm will sound at that temperature.

■ Science P5.18 A mass $m = 2$ kilograms is attached to the end of a rope of length $r = 3$ meters. The mass is whirled around at high speed. The rope can withstand a maximum tension of $T = 60$ Newtons. Write a program that accepts a rotation speed v and determines whether such a speed will cause the rope to break. *Hint:* $T = mv^2/r$.

■ Science P5.19 A mass m is attached to the end of a rope of length $r = 3$ meters. The rope can only be whirled around at speeds of 1, 10, 20, or 40 meters per second. The rope can withstand a maximum tension of $T = 60$ Newtons. Write a program where the user enters the value of the mass m , and the program determines the greatest speed at which it can be whirled without breaking the rope. *Hint:* $T = mv^2/r$.

- ■ **Science P5.20** The average person can jump off the ground with a velocity of 7 mph without fear of leaving the planet. However, if an astronaut jumps with this velocity while standing on Halley's Comet, will the astronaut ever come back down? Create a program that allows the user to input a launch velocity (in mph) from the surface of Halley's Comet and determine whether a jumper will return to the surface. If not, the program should calculate how much more massive the comet must be in order to return the jumper to the surface.



Hint: Escape velocity is $v_{\text{escape}} = \sqrt{2 \frac{GM}{R}}$, where $G = 6.67 \times 10^{-11} \text{ N m}^2/\text{kg}^2$ is the gravitational constant, $M = 1.3 \times 10^{22} \text{ kg}$ is the mass of Halley's comet, and $R = 1.153 \times 10^6 \text{ m}$ is its radius.

ANSWERS TO SELF-CHECK QUESTIONS

1. Change the if statement to


```
if (floor > 14)
{
    actualFloor = floor - 2;
}
```
2. 85. 90. 85.
3. The only difference is if originalPrice is 100. The statement in Self Check 2 sets discounted-Price to 90; this one sets it to 80.
4. 95. 100. 95.
5.


```
if (fuelAmount < 0.10 * fuelCapacity)
{
    System.out.println("red");
}
else
{
    System.out.println("green");
}
```
6. (a) and (b) are both true, (c) is false.
7. floor <= 13
8. The values should be compared with ==, not =.
9. input.equals("Y")
10. str.equals("") or str.length() == 0
11. (a) 0; (b) 1; (c) An exception occurs.
12. Syntactically incorrect: e, g, h. Logically questionable: a, d, f.
13.


```
if (scoreA > scoreB)
{
    System.out.println("A won");
}
else if (scoreA < scoreB)
{
    System.out.println("B won");
}
else
{
    System.out.println("Game tied");
}
```
14.

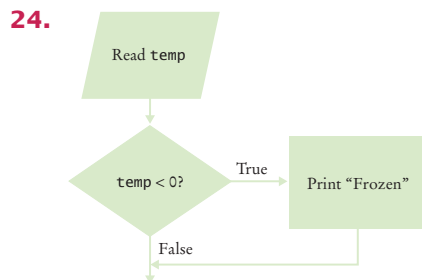

```
if (x > 0) { s = 1; }
else if (x < 0) { s = -1; }
else { s = 0; }
```
15. You could first set s to one of the three values:


```
s = 0;
if (x > 0) { s = 1; }
else if (x < 0) { s = -1; }
```
16. The if (price <= 100) can be omitted (leaving just else), making it clear that the else branch is the sole alternative.
17. No destruction of buildings.
18. Add a branch before the final else:


```
else if (richter < 0)
{
    System.out.println("Error: Negative input");
}
```
19. 3200.

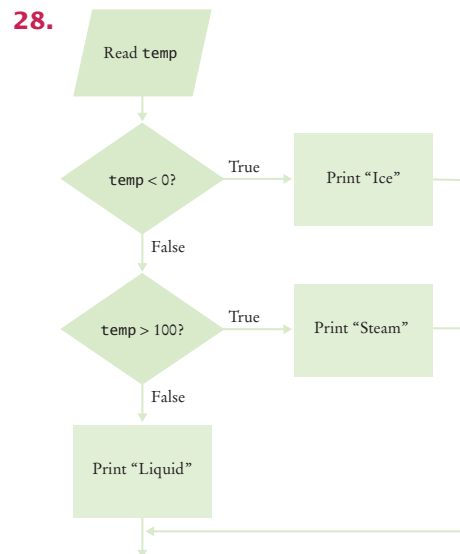
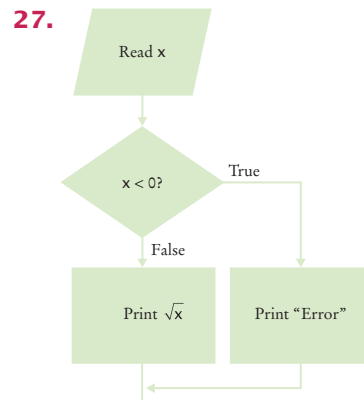
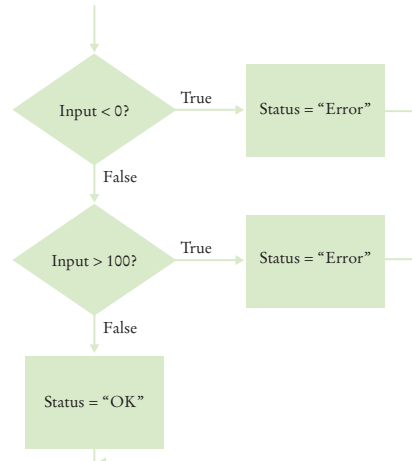
20. No. Then the computation is $0.10 \times 32000 + 0.25 \times (32000 - 32000)$.
21. No. Their individual tax is \$5,200 each, and if they married, they would pay \$10,400. Actually, taxpayers in higher tax brackets (which our program does not model) may pay higher taxes when they marry, a phenomenon known as the *marriage penalty*.
22. Change `else` in line 22 to
`else if (maritalStatus.equals("N"))`
 and add another branch after line 25:

```
else
{
    System.out.println(
        "Error: Please answer Y or N.");
}
```
23. The higher tax rate is only applied on the income in the higher bracket. Suppose you are single and make \$31,900. Should you try to get a \$200 raise? Absolutely: you get to keep 90 percent of the first \$100 and 75 percent of the next \$100.



25. The “True” arrow from the first decision points into the “True” branch of the second decision, creating spaghetti code.

26. Here is one solution. In Section 5.7, you will see how you can combine the conditions for a more elegant solution.



29.

Test Case	Expected Output	Comment
12	12	Below 13th floor
14	13	Above 13th floor
13	?	The specification is not clear— See Section 5.8 for a version of this program with error handling

- 30.** A boundary test case is a price of \$128. A 16 percent discount should apply because the problem statement states that the larger discount applies if the price is *at least* \$128. Thus, the expected output is \$107.52.

31.

Test Case	Expected Output	Comment
9	Most structures fall	
7.5	Many buildings destroyed	
6.5	Many buildings ...	
5	Damage to poorly...	
3	No destruction...	
8.0	Most structures fall	Boundary case. In this program, boundary cases are not as significant because the behavior of an earthquake changes gradually.
-1		The specification is not clear—see Self Check 18 for a version of this program with error handling.

32.

Test Case	Expected Output	Comment
(0.5, 0.5)	inside	
(4, 2)	outside	
(0, 2)	on the boundary	Exactly on the boundary
(1.414, 1.414)	on the boundary	Close to the boundary
(0, 1.9)	inside	Not less than 1 mm from the boundary
(0, 2.1)	outside	Not less than 1 mm from the boundary

33. `x == 0 && y == 0`

34. `x == 0 || y == 0`

35. `(x == 0 && y != 0) || (y == 0 && x != 0)`

36. The same as the value of frozen.

37. You are guaranteed that there are no other values. With strings or integers, you would need to check that no values such as "maybe" or -1 enter your calculations.

38. (a) Error: The floor must be between 1 and 20.

(b) Error: The floor must be between 1 and 20.

(c) 19 (d) Error: Not an integer.

39. `floor == 13 || floor <= 0 || floor > 20`

40. Check for `in.hasNextDouble()`, to make sure a researcher didn't supply an input such as oh my. Check for `weight <= 0`, because any rat must surely have a positive weight. We don't know how giant a rat could be, but the New Guinea rats weighed no more than 2 kg. A regular house rat (*rattus rattus*) weighs up to 0.2 kg, so we'll say that any weight > 10 kg was surely an input error, perhaps confusing grams and kilograms. Thus, the checks are

```
if (in.hasNextDouble())
{
    double weight = in.nextDouble();
    if (weight < 0)
    {
        System.out.println(
            "Error: Weight cannot be negative.");
    }
    else if (weight > 10)
    {
        System.out.println(
            "Error: Weight > 10 kg.");
    }
    else
    {
        Process valid weight.
    }
}
else
{
    System.out.print("Error: Not a number");
}
```

41. The second input fails, and the program terminates without printing anything.