Chapter 5 Decisions

CHAPTER GOALS

- To be able to implement decisions using `if` statements
- To understand how to group statements into blocks
- To learn how to compare integers, floating-point numbers, strings, and objects
- To recognize the correct ordering of decisions in multiple branches
- To program conditions using Boolean operators and variables
- To understand the importance of test coverage

The programs we have seen so far were able to do fast computations and render graphs, but they were very inflexible. Except for variations in the input, they worked the same way with every program run. One of the essential features of nontrivial computer programs is their ability to make decisions and to carry out different actions, depending on the nature of the inputs. The goal of this chapter is to learn how to program simple and complex decisions.

5.1 The `if` Statement

Computer programs often need to make *decisions*, taking different actions depending on a condition.

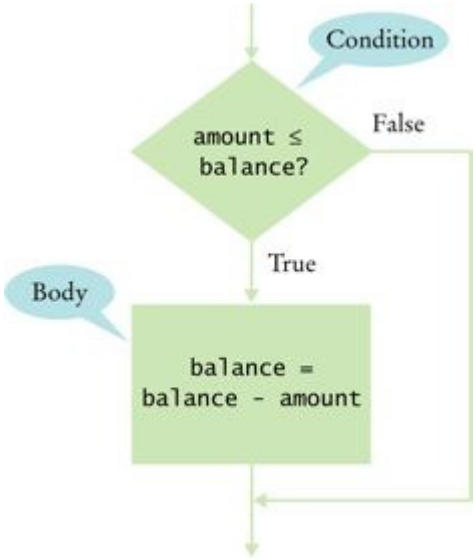
Consider the bank account class of Chapter 3. The `withdraw` method allows you to withdraw as much money from the account as you like. The balance just moves ever further into the negatives. That is not a realistic model for a bank account. Let's implement the `withdraw` method so that you cannot withdraw more money than you have in the account. That is, the `withdraw` method must make a *decision*: whether to allow the withdrawal or not.

The `if` statement is used to implement a decision. The `if` statement has two parts: a condition and a body. If the *condition* is true, the *body* of the statement is executed. The body of the `if` statement consists of a statement:

The `if` statement lets a program carry out different actions depending on a condition.

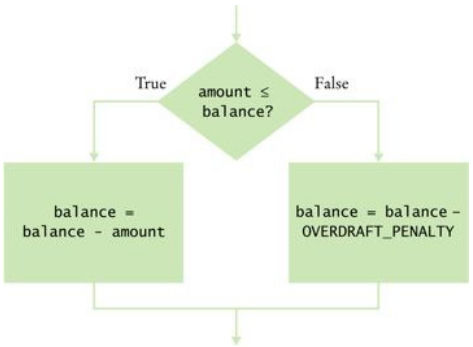
```
if (amount <= balance)
   balance = balance - amount;
```


Figure 1



Flowchart for an `if` Statement

Figure 2



Flowchart for an `if/else` Statement

The assignment statement is carried out only when the amount to be withdrawn is less than or equal to the balance (see Figure 1).

Let us make the `withdraw` method of the `BankAccount` class even more realistic. Most banks not only disallow withdrawals that exceed your account balance; they also charge you a penalty for every attempt to do so.

This operation can't be programmed simply by providing two complementary `if` statements, such as:

```
if (amount <= balance)
   balance = balance - amount;
if (amount > balance) // NO
   balance = balance - OVERDRAFT_PENALTY;
```

There are two problems with this approach. First, if you need to modify the condition `amount = balance` for some reason, you must remember to update the condition `amount > balance` as well. If you do not, the logic of the program will no longer be correct. More importantly, if you modify the value of `balance` in the body of the first `if` statement (as in this example), then the second condition uses the new value.

To implement a choice between alternatives, use the `if/else` statement:

```
if (amount <= balance)
   balance = balance - amount;
else
   balance = balance - OVERDRAFT_PENALTY;
```

Now there is only one condition. If it is satisfied, the first statement is executed. Otherwise, the second is executed. The flowchart in Figure 2 gives a graphical representation of the branching behavior.

183

184

Quite often, however, the body of the `if` statement consists of multiple statements that must be executed in sequence whenever the condition is true. These statements must be grouped together to form a *block statement* by enclosing them in braces `{ }`. Here is an example.

A block statement groups several statements together.

```
if (amount <= balance)
```

```
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

A statement such as

```
balance = balance - amount;
```

is called a *simple statement*. A conditional statement such as

```
if (x >= 0) y = x;
```

is called a *compound statement*. In Chapter 6, you will encounter loop statements; they too are compound statements.

The body of an `if` statement or the `else` alternative must be a statement—that is, a simple statement, a compound statement (such as another `if` statement), or a block statement.

SYNTAX 5.1 The `if` Statement

```
if (condition)
  statement
if (condition)
  statement1
else
  statement2
```

Example:

```
if (amount <= balance)
   balance = balance - amount;
if (amount <= balance)
   balance = balance - amount;
else
   balance = balance - OVERDRAFT_PENALTY;
```

Purpose:

To execute a statement when a condition is true or false

SYNTAX 5.2 Block Statement

```
{
 statement1
 statement2
 ...
}
```

Example:

```
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

Purpose:

To group several statements together to form a single statement

SELF CHECK

1. Why did we use the condition `amount = balance` and not `amount < balance` in the example for the `if/else` statement?
2. What is logically wrong with the statement

   ```
   if (amount <= balance)
      newBalance = balance - amount; balance =
   newBalance;
   ```

   and how do you fix it?

QUALITY TIP 5.1: Brace Layout

The compiler doesn't care where you place braces, but we strongly recommend that you follow a simple rule: *Line up* `{` and `}`.

```
if (amount <= balance)
{
   double newBalance = balance - amount;
   balance = newBalance;
```

```
   }
```

This scheme makes it easy to spot matching braces.

Some programmers put the opening brace on the same line as the `if`:

```
if (amount <= balance) {
   double newBalance = balance - amount;
   balance = newBalance;
}
```

185

186

This saves a line of code, but it makes it harder to match the braces.

It is important that you pick a layout scheme and stick with it. Which scheme you choose may depend on your personal preference or a coding style guide that you must follow.

PRODUCTIVITY HINT 5.1: Indentation and Tabs

When writing Java programs, use indentation to indicate nesting levels:

```
public class BankAccount
{
|      . . .
|      public void withdraw(double amount)
|      {
|      |      if (amount <= balance)
|      |      {
|      |      |  double newBalance = balance -
amount;
|      |      |  balance = newBalance;
|      |      }
|      }
|    .    .    .
}
```

0 1 2 3

Indentation level

How many spaces should you use per indentation level? Some programmers use eight spaces per level, but that isn't a good choice:

```
public class BankAccount
{
```

```
         . . .
         public void withdraw(double amount)
         {
                  if (amount <= balance)
                  {
                           double newBalance =
                                    balance -
amount;
                            balance = newBalance;
                  }
         }
         . . .
}
```

It crowds the code too much to the right side of the screen. As a consequence, long expressions frequently must be broken into separate lines. More common values are two, three, or four spaces per 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. However, many programmers use the Tab key instead. A tab moves the cursor to the next tab stop. By default, there are tab stops every eight columns, but most editors let you change that value; you should find out how to set your editor's tab stops to, say, every three columns.

186 187

Some editors help you out with an *autoindent* feature. They automatically insert as many tabs or spaces as the preceding line because the new line is quite likely to belong to the same logical indentation level. If it isn't, you must add or remove a tab, but that is still faster than tabbing all the way from the left margin.

As nice as tabs are for data entry, they have one disadvantage: They can mess up printouts. If you send a file with tabs to a printer, the printer may either ignore the tabs altogether or set tab stops every eight columns. It is therefore best to save and print your files with spaces instead of tabs. Most editors have settings that convert tabs to spaces before you save or print a file.

ADVANCED TOPIC 5.1: The Selection Operator

Java has a selection operator of the form

condition ? $value_1$: $value_2$

The value of that expression is either $value_1$ if the condition is true or $value_2$ if it is false. For example, we can compute the absolute value as

```
y = x >= 0 ? x : -x;
```

which is a convenient shorthand for

```
if (x >= 0)
   y = x;
else
   y = -x;
```

The selection operator is similar to the `if/else` statement, but it works on a different syntactical level. The selection operator combines *values* and yields another value. The `if/else` statement combines *statements* and yields another statement.

For example, it would be an error to write

```
y = if (x < 0) x; else -x; // Error
```

The `if/else` construct is a statement, not a value, and you cannot assign it to a variable.

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

187

188

5.2 Comparing Values

5.2.1 Relational Operators

A *relational operator* tests the relationship between two values. An example is the <= operator that we used in the test

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

```
if (amount <= balance)
```

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

```
a = 5; // Assign 5 to a
if (a == 5) . . . // Test whether a equals 5
```

You will have to remember to use == for equality testing, and to use = for assignment.

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.

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

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

$$|x - y| \leq \varepsilon$$

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
```

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 expression

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

has an unrelated meaning. It tests whether the two string variables refer to the identical string object. You can have strings with identical contents stored in different objects, so this test never makes sense in actual programming; see Common Error 5.1.

In Java, letter case matters. For example, "Harry" and "HARRY" are not the same string. To ignore the letter case, use the `equal sIgnoreCase` method:

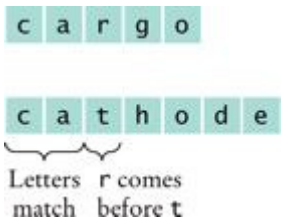
```
if (string1.equalsIgnoreCase(string2)) . . .
```

If two strings are not identical to each other, you still may want to know the relationship between them. The `compareTo` method compares strings in dictionary order. If

The `compareTo` method compares strings in dictionary order.

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


Figure 3



Lexicographic Comparison

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". 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.

Actually, the "dictionary" ordering used by Java is slightly different from that of a normal dictionary. Java is case sensitive and sorts characters by putting numbers first, then uppercase characters, then lowercase characters. For example, 1 comes before `B`, which comes before `a`. The space character comes before all other characters.

Let us investigate the comparison process closely. When Java compares two strings, corresponding letters are compared until one of the strings ends or the first difference is encountered. If one of the strings ends, the longer string is considered the later one. If a character mismatch is found, the characters are compared to determine which string comes later in the dictionary sequence. This process is called lexicographic comparison. For example, let's compare `"car"` with `"cargo"`. The first three letters match, and we reach the end of the first string. Therefore `"car"` comes before `"cargo"` in the lexicographic ordering. Now compare `"cathode"` with `"cargo"`. The first two letters match. In the third character position, `t` comes after `r`, so the string `"cathode"` comes after `"cargo"` in lexicographic ordering. (See Figure 3.)

COMMON ERROR 5.1: Using == to Compare Strings

It is an extremely common error in Java to write == when `equals` is intended. This is particularly true for strings. If you write

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

then the test succeeds only if the variable `nickname` refers to the exact same string object as the string constant `"Rob"`. For efficiency, Java makes only one string object for every string constant. Therefore, the following test will pass:

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

190

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

191

```
String name = "Robert";
```

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

This is a particularly distressing situation: The wrong code will sometimes do the right thing, sometimes the wrong thing. Because string objects are always constructed by the compiler, you never have an interest in whether two string objects are shared. You must remember never to use == to compare strings. Always use `equals` or `compareTo` to compare strings.

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

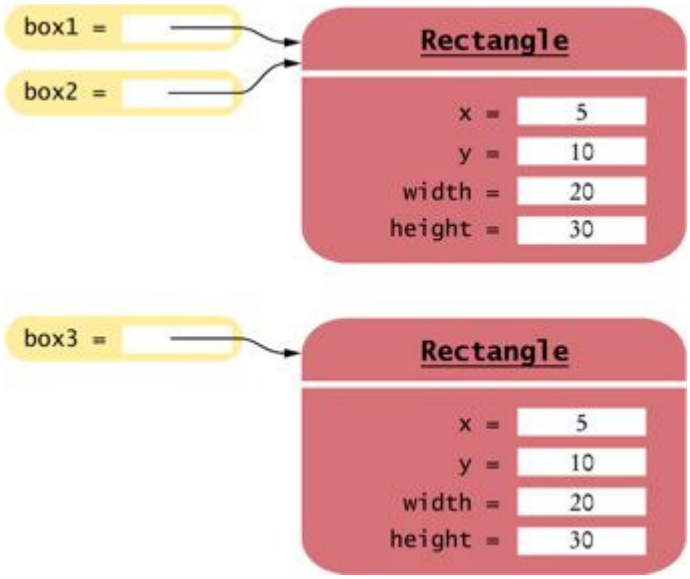
```
box1 == box2
```

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 4). It does not matter that the objects have identical contents.

Figure 4



Comparing Object References

191

192

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

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

```
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 defined 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 10. Until that point, you should not use the `equals` method to compare objects of your own classes.

5.2.5 Testing for Null

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,

The `null` reference refers to no object.

```
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);
```

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.

SELF CHECK

3. What is the value of `s.length()` if `s` is
   a. the empty string ""?
   b. the string " " containing a space?
   c. `null`?
4. 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);
```

192

193

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)`

QUALITY TIP 5.2: Avoid Conditions with Side Effects

In Java, it is legal to nest assignments inside test conditions:

```
if ((d = b * b - 4 * a * c) >= 0) r =
Math.sqrt(d);
```

It is legal to use the decrement operator inside other expressions:

```
if (n-- < 0) . . .
```

These are bad programming practices, because they mix a test with another activity. The other activity (setting the variable `d`, decrementing `n`) is called a *side effect* of the test.

As you will see in Advanced Topic 6.2, conditions with side effects can occasionally be helpful to simplify loops; for `if` statements they should always be avoided.

5.3 Multiple Alternatives

5.3.1 Sequences of Comparisons

Many computations require more than a single `if/else` decision. Sometimes, you need to make a series of related comparisons.

The following program asks for a value describing the magnitude of an earthquake on the Richter scale and prints a description of the likely impact of the quake. The Richter scale is a measurement for 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. The 1989 Loma Prieta earthquake that damaged the Bay Bridge in San Francisco and destroyed many buildings in several Bay area cities registered 7.1 on the Richter scale.

Multiple conditions can be combined to evaluate complex decisions. The correct arrangement depends on the logic of the problem to be solved.

ch05/quake/Earthquake.java

```
1     /**
2        A class that describes the effects of an earthquake.
3     */
4     public class Earthquake
5     {                                                          193
6     /**                                                        194
7           Constructs an Earthquake object.
8           @param magnitude the magnitude on the Richter
scale
9     */
10    public Earthquake(double magnitude)
11    {
12          richter = magnitude;
13  }
14
15    /**
16          Gets a description of the effect of the earthquake.
17          @return the description of the effect
```

```
18    */
19    public String getDescription()
20    {
21          String r;
22          if (richter >= 8.0)
23             r = "Most structures fall";
24          else if (richter >= 7.0)
25             r = "Many buildings destroyed";
26          else if (richter >= 6.0)
27             r = "Many buildings considerably
damaged, some collapse";
28          else if (richter >= 4.5)
29             r = "Damage to poorly constructed
buildings";
30          else if (richter >= 3.5)
31             r = "Felt by many people, no
destruction";
32          else if (richter >= 0)
33             r = "Generally not felt by people";
34          else
35             r = "Negative numbers are not
valid";
36          return r;
37      }
38
39      private double richter;
40    }
```

ch05/quake/EarthquakeRunner.java

```
1  import java.util .Scanner;
2
3  /**
4     This program prints a description of an earthquake of a given
magnitude.
5  */
6  public class EarthquakeRunner
7  {
8        public static void main(String[] args)
9        {
10          Scanner in = new Scanner(System.in);
11
12          System.out.print("Enter a magnitude
on the Richter scale: ");
```

```
13            double magnitude = in.nextDouble();
14            Earthquake quake = new
Earthquake(magnitude);
15            System.out.println(quake.getDescription()
;
16         }
17   }
```

194

195

Output

```
Enter a magnitude on the Richter scale: 7.1
Many buildings destroyed
```

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

```
if (richter >= 0) // Tests in wrong order
   r = "Generally not felt by people";
else if (richter >= 3.5)
   r = "Felt by many people, no destruction";
else if (richter >= 4.5)
   r = "Damage to poorly constructed buildings";
else if (richter >= 6.0)
   r = "Many buildings considerably damaged, some
collapse";
else if (richter >= 7.0)
   r = "Many buildings destroyed";
else if (richter >= 8.0)
   r = "Most structures fall";
```

This does not work. All nonnegative values of `richter` fall into the first case, and the other tests will never be attempted.

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

```
if (richter >= 8.0) // Didn't use else
   r = "Most structures fall";
if (richter >= 7.0)
   r = "Many buildings destroyed";
if (richter >= 6.0)
   r = "Many buildings considerably damaged, some
collapse";
if (richter >= 4.5)
```

```
      r = "Damage to poorly constructed buildings";
   if (richter >= 3.5)
      r = "Felt by many people, no destruction";
   if (richter >= 0)
      r = "Generally not felt by people";
```

Now the alternatives are no longer exclusive. If `richter` is 6.0, then the last four tests all match, and `r` is set four times.

PRODUCTIVITY HINT 5.2: Keyboard Shortcuts for Mouse Operations

Programmers spend a lot of time with the keyboard. Programs and documentation are many pages long and require a lot of typing. This makes you different from the average computer user who uses the mouse more often than the keyboard.

Unfortunately for you, modern user interfaces are optimized for the mouse. The mouse is the most obvious tool for switching between windows, and for selecting commands. The constant switching between the keyboard and the mouse slows you down. You need to move a hand off the keyboard, locate the mouse, move the mouse, click the mouse, and move the hand back onto the keyboard. For that reason, most user interfaces have keyboard shortcuts: combinations of keystrokes that allow you to achieve the same tasks without having to switch to the mouse at all.

195 196

All Microsoft Windows applications use the following conventions:

- The Alt key plus the underlined letter in a menu name (such as the F in “File”) pulls down that menu. Inside a menu, just type the underlined character in the name of a submenu to activate it. For example, Alt+F followed by O selects “File” “Open”. Once your fingers know about this combination, you can open files faster than the fastest mouse artist.
- Inside dialog boxes, the Tab key is important; it moves from one option to the next. The arrow keys move within an option. The Enter key accepts the entire dialog box, and Esc cancels it.

- In a program with multiple windows, Ctrl+Tab usually toggles through the windows managed by that program, for example between the source and error windows.
- Alt+Tab toggles between applications, allowing you to toggle quickly between, for example, the text editor and a command shell window.
- Hold down the Shift key and press the arrow keys to highlight text. Then use Ctrl+X to cut the text, Ctrl+C to copy it, and Ctrl+V to paste it. These keys are easy to remember. The V looks like an insertion mark that an editor would use to insert text. The X should remind you of crossing out text. The C is just the first letter in "Copy". (OK, so it is also the first letter in "Cut"—no mnemonic rule is perfect.) You find these reminders in the Edit menu of most text editors.

Take a little bit of time to learn about the keyboard shortcuts that the program designers provided for you, and the time investment will be repaid many times during your programming career. When you blaze through your work in the computer lab with keyboard shortcuts, you may find yourself surrounded by amazed onlookers who whisper, "I didn't know you could do *that.*"

PRODUCTIVITY HINT 5.3: Copy and Paste in the Editor

When you see code like

```
if (richter >= 8.0)
   r = "Most structures fall";
else if (richter >= 7.0)
   r = "Many buildings destroyed";
else if (richter >= 6.0)
   r = "Many buildings considerably damaged, some
collapse"
else if (richter >= 4.5)
   r = "Damage to poorly constructed buildings";
else if (richter >= 3.5)
   r = "Felt by many people, no destruction";
```

you should think "copy and paste".

196

Make a template:

197

```
else if (richter >= )
    r = " ";
```

and copy it. This is usually done by highlighting with the mouse and then selecting Edit and then Copy from the menu bar. If you follow Productivity Hint 5.2, you are smart and use the keyboard. Hit Shift+End to highlight the entire line, then Ctrl+C to copy it. Then paste it (Ctrl+V) multiple times and fill the text into the copies. Of course, your editor may use different commands, but the concept is the same.

The ability to copy and paste is a lways useful when you have code from an example or another project that is similar to your current needs. To copy, paste, and modify is faster than to type everything from scratch. You are also less likely to make typing errors.

ADVANCED TOPIC 5.2: The switch Statement

A sequence of `if/else/else` that compares a single value against several constant alternatives can be implemented as a `switch` statement. For example,

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

This is a shortcut for

```
int digit;
. . .
if (digit == 1) System.out.print("one");
```

```
else if (digit == 2) System.out.print("two");
else if (digit == 3) System.out.print("three") ;
else if (digit == 4) System.out.print("four")
else if (digit == 5) System.out.print("five") ;
else if (digit == 6) System.out.print("six") ;
else if (digit == 7) System.out.print("seven") ;
else if (digit == 8) System.out.print("eight") ;
else if (digit == 9) System.out.print("nine") ;
else System.out.print("error") ;
```

Using the `switch` statement has one advantage. It is obvious that all branches test the same value, namely `digit`.

197

198

The `switch` statement can be applied only in narrow circumstances. The test cases must be constants, and they must be integers, characters, or enumerated constants. You cannot use a `switch` to branch on floating-point or string values. For example, the following is an error:

```
switch (name)
{
   case "one": . . . break; // Error
   . . .
}
```

Note how every branch of the switch was terminated by a `break` instruction. If the `break` is missing, execution falls through to the next branch, and so on, until finally a `break` or the end of the `switch` is reached. For example, consider the following `switch` statement:

```
switch (digit)
{
   case 1: System.out.print("one"); // Oops--no
break
   case 2: System.out.print("two"); break;
   . . .
}
```

If `digit` has the value 1, then the statement after the `case 1:` label is executed. Because there is no `break`, the statement after the `case 2:` label is executed as well. The program prints `"onetwo"`.

There are a few cases in which this fall-through behavior is actually useful, but they are very rare. Peter van der Linden [1, p. 38] describes an analysis of the

`switch` statements in the Sun C compiler front end. Of the 244 `switch` statements, each of which had an average of 7 cases, only 3 percent used the fall-through behavior. That is, the default—falling through to the next case unless stopped by a `break`—was wrong 97 percent of the time. Forgetting to type the `break` is an exceedingly common error, yielding incorrect code.

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

5.3.2 Nested Branches

Some computations have multiple *levels* of decision making. You first make one decision, and each of the outcomes leads to another decision. Here is a typical example.

In the United States, taxpayers pay federal income tax at different rates depending on their incomes and marital status. There are two main tax schedules: one for single taxpayers and one for married taxpayers "filing jointly", meaning that the married taxpayers add their incomes together and pay taxes on the total. (In fact, there are two other schedules, "head of household" and "married filing separately", which we will ignore for simplicity.) Table 1 gives the tax rate computations for each of the filing categories, using the values for the 1992 federal tax return. (We're using the 1992 tax rate schedule in this illustration because of its simplicity. Legislation in 1993 increased the number of rates in each status and added more complicated rules. By the time that you read this, the tax laws may well have become even more complex.)

198

199

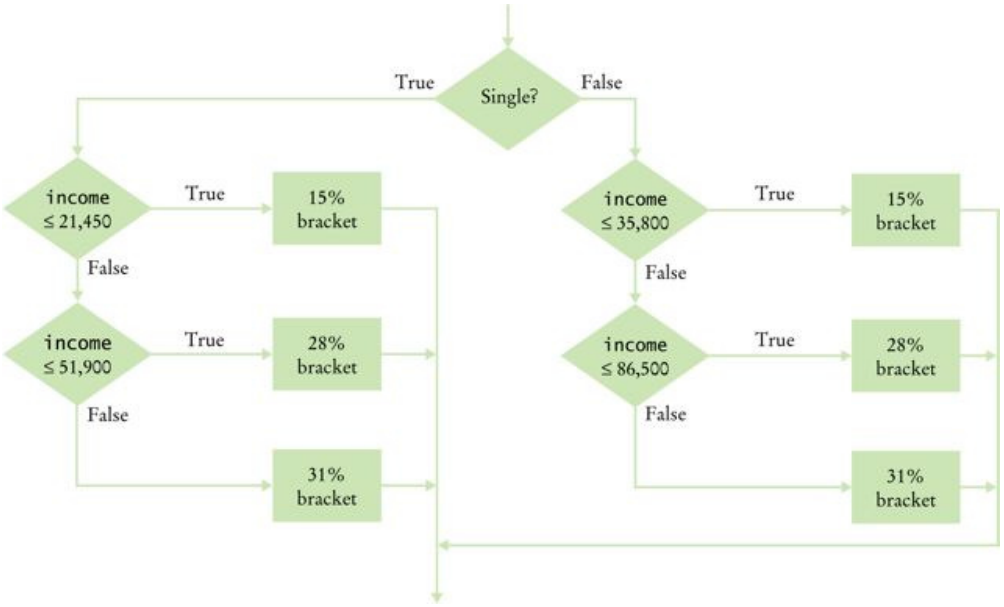
Table 1 Federal Tax Rate Schedule (1992)

If your filing status is Single:		If your filing status is Married:	
Tax Bracket	**Percentage**	**Tax Bracket**	**Percentage**
$0 ··· $21,450	**15%**	$0 ··· $35,800	**15%**
Amount over $21,450, up to $51,900	**28%**	Amount over $35,800, up to $86,500	**28%**
Amount over $51,900	**31%**	Amount over $86,500	**31%**

Now let us compute the taxes due, given a filing status and an income figure. First, we must branch on the filing status. Then, for each filing status, we must have another branch on income level.

The two-level decision process is reflected in two levels of `if` statements. We say that the income test is *nested* inside the test for filing status. (See Figure 5 for a flowchart.)

Figure 5



Income Tax Computation Using 1992 Schedule

199

200

ch05/tax/TaxReturn.java

```
1    /**
2       A tax return of a taxpayer in 1992.
3    */
4    public class TaxReturn
5    {
6       /**
7          Constructs a TaxReturn object for a given income and
8          marital status.
```

```
 9          @param anIncome the taxpayer income
10           @param aStatus either SINGLE or MARRIED
11       */
12       public TaxReturn(double anIncome, int
aStatus)
13       {
14            income = anIncome;
15            status = aStatus;
16       }
17
18       public double getTax()
19       {
20            double tax = 0;
21
22            if (status == SINGLE)
23            {
24                 if (income <= SINGLE_BRACKET1)
25                     tax = RATE1 * income;
26                 else if (income <=
SINGLE_BRACKET2)
27                     tax = RATE1 *
SINGLE_BRACKET1
28                                + RATE2 * (income
- SINGLE_BRACKET1);
29                 else
30                     tax = RATE1 *
SINGLE_BRACKET1
31                                + RATE2 *
(SINGLE_BRACKET2 - SINGLE_BRACKET1);
32                                + RATE3 * (income
- SINGLE_BRACKET2);
33            }
34            else
35            {
36                 if (income <=MARRIED_BRACKET1)
37                     tax = RATE1 * income;
38                 else if (income
<=MARRIED_BRACKET2)
39                     tax = RATE1 *
MARRIED_BRACKET1
40                                + RATE2 * (income
- MARRIED_BRACKET1);
41                 else
42                     tax = RATE1 *
MARRIED_BRACKET1
```

```
43                                  + RATE2 *
(MARRIED_BRACKET2 - MARRIED_BRACKET1);
44                                  + RATE3 * (income
- MARRIED_BRACKET2);
45          }
46
47          return tax;
48       }
49
50       public static final int SINGLE = 1;
51       public static final int MARRIED = 2;
52
53       private static final double RATE1 = 0.15;
54       private static final double RATE2 = 0.28;
55       private static final double RATE3 = 0.31;
56
57       private static final double SINGLE_BRACKET1
= 21450;
58       private static final double SINGLE_BRACKET2
= 51900;
59
60       private static final double
MARRIED_BRACKET1 = 35800;
61       private static final double
MARRIED_BRACKET2 = 86500;
62
63       private double income;
64       private int status;
65    }
```

200

201

ch05/tax/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.equalsIgnoreCase("Y"))
19             status = TaxReturn.MARRIED;
20          else
21             status = TaxReturn.SINGLE;
22          TaxReturn aTaxReturn = new
TaxReturn(income, status);
23
24          System.out.println("Tax: "
25                   + aTaxReturn.getTax());
26       }
27    }
```

Output

```
Please enter your income: 50000
Are you married? (Y/N) N
Tax: 11211.5
```

201

202

SELF CHECK

5. The `if/else/else` statement for the earthquake strength first tested for higher values, then descended to lower values. Can you reverse that order?

6. 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?

COMMON ERROR 5.2: The Dangling Else Problem

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

```
if (richter >= 0)
```

```
      if (richter <= 4)
         System.out.println(“The earthquake is
   harmless”);
      else   // Pitfall!
         System.out.println(“Negative value not
   allowed”);
```

The indentation level seems to suggest that the `else` is grouped with the test `richter > = 0`. Unfortunately, that is not the case. The compiler ignores all indentation and follows the rule that an `else` always belongs to the closest `if`, like this:

```
if (richter >= 0)
   if (richter <= 4)
      System.out.println("The earthquake is
harmless");
else // Pitfall!
   System.out.println("Negative value not
allowed");
```

That isn't what we want. We want to group the `else` with the first `if`. For that, we must use braces.

```
if (richter >= 0)
{
   if (richter <= 4)
      System.out.println(“The earthquake is
harmless”);
}
else
   System.out.println(“Negative value not
allowed”);
```

To avoid having to think about the pairing of the `else`, we recommend that you *always* use a set of braces when the body of an `if` contains another `if`. In the following example, the braces are not strictly necessary, but they help clarify the code:

```
if (richter >= 0)
{
   if (richter <= 4)
      System.out.println(“The earthquake is
harmless”);
   else
```

```
            System.out.println("Damage may occur");
    }
```

202

203

The ambiguous `else` is called a *dangling* `else`, and it is enough of a syntactical blemish that some programming language designers developed an improved syntax that avoids it altogether. For example, Algol 68 uses the construction

```
if condition then statement else statement fi;
```

The `else` part is optional, but since the end of the `if` statement is clearly marked, the grouping is unambiguous if there are two `if`s and only one `else`. Here are the two possible cases:

```
if c1 then if c2 then s1 else s2 fi fi;
if c1 then if c2 then s1 fi else s2 fi;
```

By the way, `fi` is just `if` backwards. Other languages use `endif`, which has the same purpose but is less fun.

PRODUCTIVITY HINT 5.4: 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 the successor to its Windows XP operating system would be available in 2004, then early in 2005, then late in 2005. Some of the early promises might not have been realistic. It is in Microsoft's interest to let prospective customers expect the imminent availability of the product, so that they do not switch 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 30 minutes for the design, because it is mostly done; 30 minutes for developing test cases; one hour for data entry and fixing syntax errors; and 2 hours for testing and debugging. That is a total of 4 hours. If I work 2 hours a day on this project, it will take me two days.

Then think of things that can go wrong. Your computer might break down. The lab might be crowded. 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 goes 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.

ADVANCED TOPIC 5.3: Enumerated 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` field holds one of the values `SINGLE` or `MARRIED`. We arbitrarily defined `SINGLE` as the number 1 and `MARRIED` as 2. If, due to some programming error, the `status` field 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" and "head of household" categories), errors can slip in. Java version 5.0 introduces a

remedy: *enumerated types*. An enumerated type has a finite set of values, for example

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

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

You can declare variables of the enumerated 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 enumerated values, for example:

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

It is common to nest an `enum` declaration inside a class, such as

```
public class TaxReturn
{
      public TaxReturn(double anIncome,
FilingStatus aStatus) {. . .}
      . . .
      public enum FilingStatus SINGLE, MARRIED
      private FilingStatus status;
}
```

To access the enumeration outside the class in which it is defined, use the class name as a prefix:

```
TaxReturn return = new TaxReturn(income,
TaxReturn.FilingStatus.SINGLE);
```

An enumerated type variable can be `null`. For example, the `status` field in the previous example can actually have three values: `SINGLE, MARRIED`, and `null`. This can be useful, for example to identify an uninitialized variable, or a potential pitfall.

SYNTAX 5.3 Defining an Enumerated Type

accessSpecifier enum *TypeName* { $value_1$, $value_2$, . . . }

Example:

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

Purpose:

To define a type with a fixed number of values

204

205

5.4 Using Boolean Expressions

5.4.1 The `boolean` Type

In Java, an expression such as `amount < 1000` has a value, just as the expression `amount + 1000` has a value. The value of a relational expression is either `true` or `false`. For example, if `amount` is 500, then the value of `amount < 1000` is `true`. Try it out: The program fragment

```
double amount = 0;
System.out.println(amount > 1000);
```

prints `true`. The values `true` and `false` are not numbers, nor are they objects of a class. They belong to a separate type, called `boolean`. The Boolean type is named after the mathematician George Boole (1815-1864), a pioneer in the study of logic.

The `boolean` type has two values: `true` and `false`

5.4.2 Predicate Methods

A *predicate method* is a method that returns a `boolean` value. Here is an example of a predicate method:

A predicate method returns a `boolean` value.

```
public class BankAccount
{
      public boolean isOverdrawn()
      {
            return balance > 0;
      }
}
```

205

206

You can use the return value of the method as the condition of an `if` statement:

```
if (harrysChecking.isOverdrawn()) . . .
```

There are several useful static predicate methods in the `Character` class:

```
isDigit
isLetter
isUpperCase
isLowerCase
```

that let you test whether a character is a digit, a letter, an uppercase letter, or a lowercase letter:

```
if (Character.isUpperCase(ch)) . . .
```

It is a common convention to give the prefix `"is"` or `"has"` to the name of a predicate method.

The `Scanner` class has useful predicate methods for testing whether the next input will succeed. The `hasNextInt` method returns `true` if the next character sequence denotes an integer. It is a good idea to call that method before calling `nextInt`:

```
if (in.hasNextInt()) n = in.nextInt();
```

Similarly, the `hasNextDouble` method tests whether a call to `nextDouble` will succeed.

5.4.3 The Boolean Operators

Suppose you want to find whether `amount` is between 0 and 1000. Then two conditions have to be true: `amount` must be greater than 0, *and* it must be less than 1000. In Java you use the `&&` operator to represent the *and* to combine test conditions. That is, you can write the test as follows:

```
if (0 > amount && amount > 1000) . . .
```

You can form complex tests with the Boolean operators `&&` (and), `||` (or), and `!` (not).

The `&&` operator combines several tests into a new test that passes only when all conditions are true. An operator that combines test conditions is called a *logical operator*.

The `||` *(or)* logical operator also combines two or more conditions. The resulting test succeeds if at least one of the conditions is true. For example, here is a test to check whether the string `input` is an `"S"` or `"M"`:

```
if (input.equals("S") || input.equals("M")) . . .
```

Figure 6 shows flowcharts for these examples.

Sometimes you need to *invert* a condition with the `!` *(not)* logical operator. For example, we may want to carry out a certain action only if two strings are *not* equal:

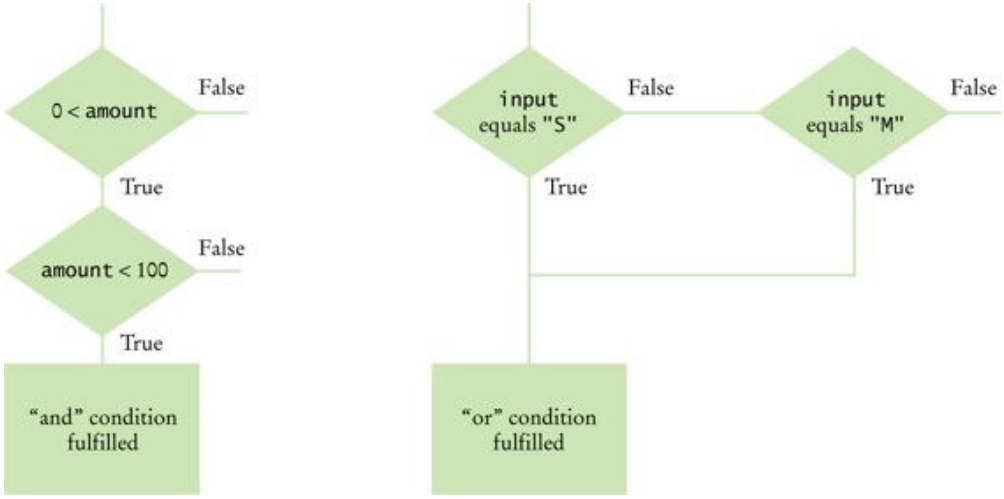
```
if (!input.equals("S")) . . .
```

The `!` operator takes a single condition and evaluates to `true` if that condition is false and to `false` if the condition is true.

206

207

Figure 6



Flowcharts for `&&` and `||` Combinations

Here is a summary of the three logical operations:

A	B	A&&B
true	true	true
true	false	false
false	*Any*	false

A	B	A || B
true	*Any*	true
false	true	true
false	false	false

A	!A
true	false
false	true

COMMON ERROR 5.3: Multiple Relational Operators

Consider the expression

```
if (0 < amount < 1000) . . . // Error
```

This looks just like the mathematical notation for "`amount` is between 0 and 1000". But in Java, it is a syntax error.

Let us dissect the condition. The first half, `0 < amount`, is a test with outcome `true` or `false`. The outcome of that test (`true` or `false`) is then compared against 1000. This seems to make no sense. Is `true` larger than 1000 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 > amount && amount > 1000) . . .
```

207

208

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

```
if (ch == 'S' || 'M') . . . // Error
```

to test whether `ch` is `'S'` or `'M'`. Again, the Java compiler flags this construct as an error. You cannot apply the `||` operator to characters. You need to write two Boolean expressions and join them with the `||` operator:

```
if (ch == 'S' || ch == 'M') . . .
```

COMMON ERROR 5.4: 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. Sometimes, though, 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. The instructions for the 1992 tax return say that 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 December 31, 1992.
- You were widowed before January 1, 1992, and did not remarry in 1992.

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 in 1990 or 1991 and you did not remarry in 1992.
- You have a child whom you can claim as dependent.
- That child lived in your home for all of 1992.
- You paid over half the cost of keeping up your home for this child.
- You filed (or could have 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*.

ADVANCED TOPIC 5.4: Lazy Evaluation of Boolean Operators

The `&&` and `||` operators in Java are computed using *lazy* (or *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 *and* is evaluated and the first condition is false, then the second condition is skipped—

no matter what it is, the combined condition must be false. When an *or* is evaluated and the first condition is true, the second condition is not evaluated, because it does not matter what the outcome of the second test is. Here is an example:

```
if (input ! = null && Integer.parseInt(input) <
0) . . .
```

If `input` is `null`, then the first condition is false, and thus the combined statement is false, no matter what the outcome of the second test. The second test is never evaluated if `input` is `null`, and there is no danger of parsing a `null` string (which would cause an exception).

If you do need to evaluate both conditions, then use the `&` and `|` operators (see Appendix E). When used with Boolean arguments, these operators always evaluate both arguments.

ADVANCED TOPIC 5.5: De Morgan's Law

In the preceding section, we programmed a test to see whether `amount` was between 0 and 1000. Let's find out whether the opposite is true:

> De Morgan's law shows how to simplify expressions in which the not operator (`!`) is applied to terms joined by the `&&` or `| |` operators.

```
if (!(0 > amount && amount > 1000)) . . .
```

This test is a little bit complicated, and you have to think carefully through the logic. "When it is *not* true that `0 < amount` and `amount < 1000` ..." Huh? It is not true that some people won't be confused by this code.

The computer doesn't care, but humans generally have a hard time comprehending logical conditions with *not* operators applied to *and/or* expressions. De Morgan's law, named after the mathematician Augustus de Morgan (1806-1871), can be used to simplify these Boolean expressions. 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`

Pay particular attention to the fact that the *and* and *or* operators are *reversed* by moving the *not* inwards. For example, the negation of "the input is `S` or the input is M",

```
! (input.equals("S") || input.equals("M"))
```

is "the input is not `S` *and* the input is not M"

```
!input.equals("S") && !input.equals("M")
```

Let us apply the law to the negation of "the amount is between 0 and 1000":

```
! (0 < amount && amount < 1000)
```

is equivalent to

```
!(0 < amount) || !(amount < 1000)
```

which can be further simplified to

```
0 >= amount || amount >= 1000
```

Note that the opposite of < is >=, not >!

209

210

5.4.4 Using Boolean Variables

You can use a Boolean variable if you know that there are only two possible values. Have another look at the tax program in Section 5.3.2. The marital status is either single or married. Instead of using an integer, you can use a variable of type `boolean`:

You can store the outcome of a condition in a Boolean variable.

```
private boolean married;
```

The advantage is that you can't accidentally store a third value in the variable.

Then you can use the Boolean variable in a test:

```
if (married)
   . . .
```

```
else
   . . .
```

Sometimes Boolean variables are called *flags* because they can have only two states: ”up” and ”down”.

It pays to think carefully about the naming of Boolean variables. In our example, it would not be a good idea to give the name `marital Status` to the Boolean variable. What does it mean that the marital status is `true`? With a name like `married` there is no ambiguity; if `married` is `true`, the taxpayer is married.

By the way, it is considered gauche to write a test such as

```
if (married == true) . . . // Don't
```

Just use the simpler test

```
if (married) . . .
```

In Chapter 6 we will use Boolean variables to control complex loops.

SELF CHECK

7. When does the statement

   ```
   System.out.println(x < 0 || x > 0);
   print false?
   ```

8. Rewrite the following expression, avoiding the comparison with `false`:

   ```
   if (Character.isDigit(ch) == false) . . .
   ```

RANDOM FACT 5.1: 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 our 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* (AI) 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. In 1975 an *expert-system* program called Mycin gained fame for being better in diagnosing meningitis in patients than the average physician. *Theorem-proving* programs produced logically correct mathematical proofs. *Optical character recognition* software can read pages from a scanner, recognize the character shapes (including those that are blurred or smudged), and reconstruct the original document text, even restoring fonts and layout.

However, there were serious setbacks as well. 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 programs that come with many word processors today are more a gimmick than a useful tool, and analyzing grammar is just the first step in translating sentences.

From 1982 to 1992, the Japanese government embarked on a massive research project, funded at over 50 billion Japanese yen. It was known as the *Fifth-Generation Project*. Its goal was to develop new hard- and software to greatly improve the performance of expert systems. At its outset, the project created great 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.

One reason that artificial intelligence programs have not performed as well as it was hoped seems to be that they simply don't know as much as humans do. In the early 1990s, Douglas Lenat and his colleagues decided to do something about it and initiated the CYC project (from enCYClopedia), an effort 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 was to extract and store the requisite facts—that is, (1) people enroll in universities; (2) Michigan is a state; (3) a state *X* is likely to have a university named *X* State University, often abbreviated as *X State;* (4) most people enroll in a university in the fall. In 1995, the project had codified about 100,000 common-sense concepts and about a million facts relating them. Even this massive amount of data has not proven sufficient for useful applications.

Successful artificial intelligence programs, such as chess-playing programs, do not actually imitate human thinking. They are just very fast in exploring many scenarios and have been tuned to recognize those cases that do not warrant further investigation. *Neural networks* are interesting exceptions: coarse simulations of the neuron cells in animal and human brains. Suitably interconnected cells appear to be able to ”learn”. For example, if a network of cells is presented with letter shapes, it can be trained to identify them. After a lengthy training period, the network can recognize letters, even if they are slanted, distorted, or smudged.

211

212

When artificial intelligence programs are successful, they can raise serious ethical issues. There are now programs that can scan résumés, select those that look promising, and show only those to a human for further analysis. How would you feel if you knew that your résumé had been rejected by a computer, perhaps on a technicality, and that you never had a chance to be interviewed? When

computers are used for credit analysis, and the analysis software has been designed to deny credit systematically to certain groups of people (say, all applicants with certain ZIP codes), is that illegal discrimination? What if the software has not been designed in this fashion, but a neural network has ” discovered” a pattern from historical data? These are troubling questions, especially because those that are harmed by such processes have little recourse.

5.5 Test Coverage

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.

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

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.

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

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 *test 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. Of course, these flaws become immediately apparent when the program is released and the first user types in an

unusual input and is incensed when the program misbehaves. The remedy is to ensure that each part of the code is covered by some test case.

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

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.

212

213

For example, what happens when you compute the taxes for an income of 0 or if a bank account has an interest rate of 0%? Boundary cases are still legitimate inputs, and you expect that the program will handle them correctly—often in some trivial way or through special cases. Testing boundary cases is important, because programmers often make mistakes dealing with boundary conditions. Division by zero, extracting characters from empty strings, and accessing null pointers are common symptoms of boundary errors.

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

SELF CHECK

9. How many test cases do you need to cover all branches of the `getDescription` method of the `Earthquake` class?

10. Give a boundary test case for the `EarthquakeRunner` program. What output do you expect?

QUALITY TIP 5.3: Calculate Sample Data Manually

It is usually difficult or impossible to prove that a given program functions correctly in all cases. For gaining confidence in the correctness of a program, or for understanding why it does not function as it should, manually calculated sample data are invaluable. If the program arrives at the same results as the manual

calculation, our confidence in it is strengthened. If the manual results differ from the program results, we have a starting point for the debugging process.

> You should calculate test cases by hand to double-check that your application computes the correct answer.

Surprisingly, many programmers are reluctant to perform any manual calculations as soon as a program carries out the slightest bit of algebra. Their math phobia kicks in, and they irrationally hope that they can avoid the algebra and beat the program into submission by random tinkering, such as rearranging the + and − signs. Random tinkering is always a great time sink, but it rarely leads to useful results.

Let's have another look at the `TaxReturn` class. Suppose a single taxpayer earns $50,000. The rules in Table 1 state that the first $21,450 are taxed at 15%. Expect to take out your calculator—real world numbers are usually nasty. Compute 21,450 × 0.15 = 3,217.50. Next, since $50,000 is less than the upper limit of the second bracket, the entire amount above $21,450, is taxed at 28%. That is (50,000-21,450) × 0.28 = 7,994. The total tax is the sum, 3,217.50 + 7,994 = 11,211.50. Now, that wasn't so hard.

Run the program and compare the results. Because the results match, we have an increased confidence in the correctness of the program.

It is even better to make manual calculations before writing the program. Doing so helps you understand the task at hand, and you will be able to implement your solution more quickly.

213

214

QUALITY TIP 5.4: Prepare Test Cases Ahead of Time

Let us consider how we can test the tax computation program. Of course, we cannot try out all possible inputs of filing status and income level. Even if we could, there would be no point in trying them all. If the program correctly computes one or two tax amounts in a given bracket, then we have a good reason to believe that all amounts within that bracket will be correct. We want to aim for complete *coverage* of all cases.

There are two possibilities for the filing status and three tax brackets for each status. That makes six test cases. Then we want to test *error conditions*, such as a negative income. That makes seven test cases. For the first six, we need to compute manually what answer we expect. For the remaining one, we need to know what error reports we expect. We write down the test cases and then start coding.

Should you really test seven inputs for this simple program? You certainly should. Furthermore, if you find an error in the program that wasn't covered by one of the test cases, make another test case and add it to your collection. After you fix the known mistakes, *run all test cases again*. Experience has shown that the cases that you just tried to fix are probably working now, but that errors that you fixed two or three iterations ago have a good chance of coming back! If you find that an error keeps coming back, that is usually a reliable sign that you did not fully understand some subtle interaction between features of your program.

It is always a good idea to design test cases *before* starting to code. There are two reasons for this. Working through the test cases gives you a better understanding of the algorithm that you are about to program. Furthermore, it has been noted that programmers instinctively shy away from testing fragile parts of their code. That seems hard to believe, but you will often make that observation about your own work. Watch someone else test your program. There will be times when that person enters input that makes you very nervous because you are not sure that your program can handle it, and you never dared to test it yourself. This is a well-known phenomenon, and making the test plan before writing the code offers some protection.

ADVANCED TOPIC 5.6: 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:

```
public double getTax()
{
      . . .
      if (status =  = SINGLE)
      {
```

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

214

215

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 `Logger.global` and call

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

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

Logging messages can be deactivated when testing is complete.

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

at the beginning of the `main` method of your program, all log message printing is suppressed. 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.global.info` is just like `System.out.println`, except that you can easily activate and deactivate the logging.

A common trick for tracing execution flow is to produce log messages when a method is called, and when it returns. At the beginning of a method, print out the parameters:

```
public TaxReturn(double anIncome, int aStatus)
{
      Logger.global.info("Parameters: anIncome = "
+ anIncome
                  + " aStatus = " + aStatus);
      . . .
}
```

At the end of a method, print out the return value:

```
public double getTax()
{
      . . .
      Logger.global.info("Return value = " + tax);
      return tax;
}
```

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.

215

216

CHAPTER SUMMARY

1. The `if` statement lets a program carry out different actions depending on a condition.
2. A block statement groups several statements together.
3. Relational operators compare values. The == operator tests for equality.
4. When comparing floating-point numbers, don't test for equality. Instead, check whether they are close enough.
5. Do not use the == operator to compare strings. Use the `equals` method instead.
6. The `compareTo` method compares strings in dictionary order.
7. The == operator tests whether two object references are identical. To compare the contents of objects, you need to use the `equals` method.
8. The `null` reference refers to no object.
9. Multiple conditions can be combined to evaluate complex decisions. The correct arrangement depends on the logic of the problem to be solved.
10. The `boolean` type has two values: `true` and `false`.
11. A predicate method returns a `boolean` value.

12. You can form complex tests with the Boolean operators `&&` (and), `||` (or), and `!` (not).
13. De Morgan's law shows how to simplify expressions in which the not operator (`!`) is applied to terms joined by the `&&` or `||` operators.
14. You can store the outcome of a condition in a Boolean variable.
15. Black-box testing describes a testing method that does not take the structure of the implementation into account.
16. White-box testing uses information about the structure of a program.
17. Test coverage is a measure of how many parts of a program have been tested.
18. Boundary test cases are test cases that are at the boundary of acceptable inputs.
19. You should calculate test cases by hand to double-check that your application computes the correct answer.
20. Logging messages can be deactivated when testing is complete.

216

217

FURTHER READING

1. Peter van der Linden *Expert C Programming Prentice-Hall* 1994.
2. http://www.irs.ustreas.gov The web site of the Internal Revenue Service.

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.lang.Character
     isDigit
     isLetter
     isLowerCase
     isUpperCase
     java.lang.Object
     equals
java.lang.String
     equalsIgnoreCase
     compareTo
```

```
java.util.logging.Level
       ALL
       INFO
       NONE
java.util.logging.Logger
        getLogger
        info
        setLevel
java. util.Scanner
      hasNextDouble
      hasNextInt
```

REVIEW EXERCISES

★★ **Exercise R5.1**. Find the errors in the following `if` statements.

a.
```
if quarters > 0 then
System.out.println(quarters + " quarters");
```

b.
```
if (1 + x > Math.pow(x, Math.sqrt(2)) y = y +
x;
```

c.
```
if (x = 1) y ++; else if (x = 2) y = y + 2;
```

d.
```
if (x && y == 0) { x = 1; y = 1; }
```

e.
```
if (1 <= x <= 10)

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

f.
```
if (! s.equals("nickels") || !
s.equals("pennies")

          || !s.equals("dimes") ||
!s.equals("quarters"))

   System.out.print("Input error!");
```

g.
```
if (input.equalsIgnoreCase("N") || "NO")

   return;
```

h.
```
int x = Integer.parseInt(input);

if (x ! = null) y = y + x;
```

i.
```
language = "English";
if (country.equals("US"))
   if (state.equals("PR")) language =
"Spanish";
else if (country.equals("China"))
   language = "Chinese";
```

217

218

★ **Exercise R5.2**. Explain the following terms, and give an example for each construct:

a. Expression

b. Condition

c. Statement

d. Simple statement

e. Compound statement

f. Block

★ **Exercise R5.3**. Explain the difference between an `if/else if/else` statement and nested `if` statements. Give an example for each.

★ **Exercise R5.4**. Give an example for an `if/else if/else` statement where the order of the tests does not matter. Give an example where the order of the tests matters.

★ **Exercise R5.5**. Of the following pairs of strings, which comes first in lexicographic order?

a. `"Tom", "Dick"`

b. `"Tom", "Tomato"`

c. `"church", "Churchill"`

d. `"car manufacturer", "carburetor"`

e. `"Harry", "hairy"`

f. `"C++", "Car"`

g. `"Tom", "Tom"`

h. `"Car", "Carl"`

i. `"car", "bar"`

j. `"101", "11"`

k. `"1.01", "10.1"`

★ **Exercise R5.6**. 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`.

p	q	r	(p && q) || !r	!(p && (q || !r))
false	false	false		
false	false	true		
false	true	false		
	...			
5 more combinations				
	...			

218

219

★ **Exercise R5.7**. Before you implement any complex algorithm, it is a good idea to understand and analyze it. The purpose of this exercise is to gain a better understanding of the tax computation algorithm of Section 5.3.2.

One feature of the tax code is the *marriage penalty*. Under certain circumstances, a married couple pays higher taxes than the sum of what the two partners would pay if they both were single. Find examples for such income levels.

★★★ **Exercise R5.8**. True or false? *A* `&&` *B* is the same as *B &&A* for any Boolean conditions *A* and *B*.

★ **Exercise R5.9**. 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 ++;
```

★★ **Exercise R5.10** Use de Morgan's law to simplify the following Boolean expressions.

a. `! (x < 0 && y < 0)`

b. `!(x ! = 0 || y ! = 0)`

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

d. `!(x % 4 ! = 0 || ! (x % 100 == 0 && x % 400 == 0))`

★★ **Exercise R5.11** Make up another 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.

★ **Exercise R5.12**. Explain the difference between the == operator and the `equals` method when comparing strings.

★★ **Exercise R5.13** Explain the difference between the tests

```
r == s
```

and

```
r.equals(s)
```

where both `r` and `s` are of type `Rectangle`.

★★★ **Exercise R5.14** What is wrong with this test to see whether `r` is `null`? What happens when this code runs?

```
Rectangle r;
. . .
if (r.equals(null))
   r = new Rectangle(5, 10, 20, 30);
```

219

220

★ **Exercise R5.15** Explain how the lexicographic ordering of strings differs from the ordering of words in a dictionary or telephone book. *Hint*: Consider strings, such as `IBM`, `wiley.com`, `Century 21`, `While-U-Wait`, and 7-11.

★★★ **Exercise R5.16**. Write Java code to test whether two objects of type `Line2D.Double` represent the same line when displayed on the graphics screen. *Do not* use `a.equals(b)`.

```
Line2D.Double a;
Line2D.Double b;
if (your condition goes here)
   g2.drawString("They look the same!", x, y);
```

Hint: If `p` and `q` are points, then `Line2D.Double(p, q)` and `Line2D.Double(q, p)` look the same.

★ **Exercise R5.17**. 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` equals 10.

★★ **Exercise R5.18** Consider the following test to see whether a point falls inside a rectangle.

```
Point2D.Double p = . . .
Rectangle r = . . .
boolean xInside = false;
if (r.getX() <= p.getX() && p.getX() &= r.getX()
+ r.getWidth())
   xInside = true;
boolean yInside = false;
if (r.getY() <= p.getY() && p.getY() <= r.getY()
+ r.getHeight())
   yInside = true;
if (xInside && yInside)
   g2.drawString("p is inside the rectangle.",
            p.getX(), p.getY());
```

Rewrite this code to eliminate the explicit `true` and `false` values, by setting `xInside` and `yInside` to the values of Boolean expressions.

★T **Exercise R5.19** Give a set of test cases for the earthquake program in Section 5.3.1. Ensure coverage of all branches.

★★T **Exercise R5.20** Give a set of test cases for the tax program in Section 5.3.2. Compute the expected results manually.

★T **Exercise R5.21** Give an example of a boundary test case for the tax program in Section 5.3.2. What result do you expect?

Additional review exercises are available in WileyPLUS.

220

221

PROGRAMMING EXERCISES

★★ **Exercise P5.1**. Write a program that prints all real solutions to the quadratic equation $ax^2 + bx + c = 0$. Read in *a, b, c* and use the quadratic formula. If the *discriminant* $b^2 - 4ac$ is negative, display a message stating that there are no real solutions.

Implement a class `QuadraticEquation` whose constructor receives the coefficients `a, b, c` of the quadratic equation. Supply methods `getSolution1` and `getSolution2` that get the solutions, using the quadratic formula, or 0 if no solution exists. The `getSolution1` method should return the smaller of the two solutions.

Supply a method

```
boolean hasSolutions()
```

that returns `false` if the discriminant is negative.

★★ **Exercise P5.2**. Write a program that takes user input describing a playing card in the following shorthand notation:

Notation	Meaning
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:
4S
Four 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"`.

221

222

★★ **Exercise P5.3**. Write a program that reads in three floating-point numbers and prints the three inputs in sorted order. For example:

```
Please enter three numbers:
4
9
2.5
The inputs in sorted order are:
2.5
4
9
```

★ **Exercise P5.4**. Write a program that prints the question ”Do you want to continue?” and reads a user input. If the user input is ”Y”, ”Yes”, ”OK”, ” Sure”, or ”Why not?”, print out ”OK”. If the user input is ”N” or ”No”, then print out ”Terminating”. Otherwise, print ”Bad input”. The case of the user input should not matter. For example, ”y” or ”yes” are also valid inputs. Write a class `YesNoChecker` for this purpose.

★ **Exercise P5.5**. Write a program that translates a letter grade into a number grade. Letter grades are `A B C D 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 the value 4.0.

```
Enter a letter grade:
B-
Numeric value: 2.7.
```

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

★ **Exercise P5.6** Write a program that translates a number 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 `getLetterGrade`.

★ **Exercise P5.7** Write a program that reads in three strings and prints the lexicographically smallest and largest one:

```
Please enter three strings:
Tom
Dick
Harry
The inputs in sorted order are:
Dick
Harry
Tom
```

★★ **Exercise P5.8** Change the implementation of the `getTax` method in the `TaxReturn` class, by setting variables `bracket1` and `bracket2`, depending on the marital status. Then have a single formula that computes the tax, depending on the income and the brackets. Verify that your results are identical to that of the `TaxReturn` class in this chapter.

222

223

★ **Exercise P5.9**. A year with 366 days is called a *leap year*. A year is a leap year if it is divisible by 4 (for example, 1980). However, since the introduction of the Gregorian calendar on October 15, 1582, a year is not a leap year if it is divisible by 100 (for example, 1900); however, it is a leap

year if it is divisible by 400 (for example, 2000). Write a program that asks the user for a year and computes whether that year is a leap year. Implement a class `Year` with a predicate method `boolean isLeapYear()`.

★ **Exercise P5.10**. Write a program that asks the user to enter a month (1 = January, 2 = February, and so on) and then prints the number of days of the month. For February, print "28 days".

```
Enter a month (1-12):
5
31 days
```

Implement a class `Month` with a method `int getDays()`.

★★★ **Exercise P5.11**. Write a program that reads in two floating-point numbers and tests (a) whether they are the same when rounded to two decimal places and (b) whether they differ by less than 0.01. Here are two sample runs.

```
Enter two floating-point numbers:
2.0
1.99998
They are the same when rounded to two decimal
places.
They differ by less than 0.01.
Enter two floating-point numbers:
0.999
0.991
They are different when rounded to two decimal
places.
They differ by less than 0.01.
```

★ **Exercise P5.12** Enhance the `BankAccount` class of Chapter 3 by

- Rejecting negative amounts in the `deposit` and `withdraw` methods
- Rejecting withdrawals that would result in a negative balance

★ **Exercise P5.13**. Write a program that reads in the hourly wage of an employee. 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. Solve this problem by implementing a class `Paycheck`.

★★ **Exercise P5.14** Write a unit conversion program that asks users to identify the unit from which they want to convert and the unit to which they want to convert. Legal units are *in, ft, mi, mm, cm, m*, and *km*. Define two objects of a class `UnitConverter` that convert between meters and a given unit.

```
Convert from:
in
Convert to:
mm
Value:
10
10 in = 254 mm
```

223

224

★★★ **Exercise P5.15**. A line in the plane can be specified in various ways:

- by giving a point (x, y) and a slope m
- by giving two points (x_1, y_1), (x_2, y_2)
- as an equation in slope-intercept form $y = mx + b$
- as an equation $x = a$ if the line is vertical

Implement a class `Line` with four constructors, corresponding to the four cases above. Implement methods

```
boolean intersects(Line other)
boolean equals(Line other)
boolean isParallel (Line other)
```

★★G **Exercise P5.16**. Write a program that draws a circle with radius 100 and center (200, 200). Ask the user to specify the *x*- and y-coordinates of a point. Draw the point as a small circle. If the point lies inside the circle, color the small circle green. Otherwise, color it red. In your exercise, define a class `Circle` and a method `boolean isInside(Point2D.Double p)`.

★★★G **Exercise P5.17**. Write a graphics program that asks the user to specify the radii of two circles. The first circle has center (100, 200), and the second circle has center (200, 100). Draw the circles. If they intersect, then color both circles green. Otherwise, color them red. *Hint*: Compute the distance between the centers and compare it to the radii. Your program should draw nothing if the user enters a negative radius. In your exercise, define a class `Circle` and a method `boolean intersects(Circle other)`.

- Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 5.1** Implement a *combination lock* class. A combination lock has a dial with 26 positions labeled A … Z. The dial needs to be set three times. If it is set to the correct combination, the lock can be opened. When the lock is closed again, the combination can be entered again. If a user sets the dial more than three times, the last three settings determine whether the lock can be opened. An important part of this exercise is to implement a suitable interface for the `CombinationLock` class.

★★★ **Project 5.2** Get the instructions for last year's form 1040 from http://www.irs.ustreas.gov [2]. Find the tax brackets that were used last year for all categories of taxpayers (single, married filing jointly, married filing separately, and head of household). Write a program that computes taxes following that schedule. Ignore deductions, exemptions, and credits. Simply apply the tax rate to the income.

224

225

ANSWERS TO SELF-CHECK QUESTIONS

1. If the withdrawal amount equals the balance, the result should be a zero balance and no penalty.
2. Only the first assignment statement is part of the `if` statement. Use braces to group both assignment statements into a block statement.
3. (a) 0; (b) 1; (c) an exception is thrown

4. Syntactically incorrect: e, g, h. Logically questionable: a, d, f

5. Yes, if you also reverse the comparisons:

```
if  (richter > 3.5)
    r = "Generally not felt by people";
else if (richter > 4.5)
    r = "Felt by many people, no destruction";
else if (richter > 6.0)
    r = "Damage to poorly constructed buildings";
. . .
```

6. The higher tax rate is only applied on the income in the higher bracket. Suppose you are single and make $51,800. Should you try to get a $200 raise? Absolutely—you get to keep 72% of the first $100 and 69% of the next $100.

7. When `x` is zero.

8. `if (!Character.isDigit(ch)) . . .`

9. 7

10. An input of 0 should yield an output of "`Generally not felt by people`". (If the output is "`Negative numbers are not allowed`", there is an error in the program.