CHAPTER

5

INPUT AND SELECTION

In this chapter, you will:

- Accept keyboard input
- Use the JOptionPane class for GUI input and output
- Draw flowcharts
- Make decisions with the `if` and `if...else` structures
- Use compound statements in an `if` or `if...else` structure
- Nest `if` and `if...else` statements
- Use AND and OR operators
- Use the `switch` statement
- Use the conditional and NOT operators
- Understand precedence

Lynn Greenbrier asks, "Why are you frowning?"

"It's fun writing programs," you tell her, "but I don't think my programs can do much yet. When I use programs written by other people, I can respond to questions and make choices. In addition, other people's programs keep running for a while—the programs I write finish as soon as they start."

"You're disappointed because the programs you've written so far simply carry out a sequence of steps," Lynn says. "You need to make your programs interactive by accepting user input. To be able to do this, you need to learn about decision-making structures."

PREVIEWING THE CHOOSEMANAGER PROGRAM USING THE EVENT CLASS

To preview the ChooseManager program using the Event class:

1. In your text editor, open the **Chap5Event.java** file from the Chapter.05 folder on your Student Disk and examine the code. This file contains a class definition for a class that stores information about events that Event Handlers Incorporated will handle. You will create a similar class file in this chapter.
2. At the command line, compile the **Chap5Event.java** file using the command **`javac Chap5Event.java`**.
3. Open the **Chap5ChooseManager.java** file from the Chapter.05 folder on your Student Disk and examine the code. This file contains a program that will demonstrate prompting the user for input and creating objects based on the input.
4. At the command line, compile the **Chap5ChooseManager.java** file using the command **`javac Chap5ChooseManager.java`**.
5. Execute the program by typing the command **`java Chap5ChooseManager`**. At the prompt to enter C, P, or N, ignore the directions, and enter an invalid letter. Do this as many times as you like—the program will continue to prompt you until you enter a valid letter. Then enter **C**, **P**, or **N** to see the name of the manager and the minimum charge assigned to your event. You will create a similar program in this chapter.

ACCEPTING KEYBOARD INPUT

In Chapters 1 through 4 of this book, you wrote programs that created objects, performed mathematical calculations, and produced output. A shortcoming of these programs is that when you write the program you must know the values with which you want to work. It is far more useful to provide values to your program at **run time**, that is, while the program is executing. A program that accepts values at run time is **interactive** because it exchanges communications, or interacts, with the user. Providing values during the execution of a program requires input; the simplest form of input to use is keyboard entry from the program's user.

You already have used the System class and its out object and println() method to produce output. The **in object** is similar; it has access to a method named read() that retrieves data from the keyboard. Figure 5-1 shows a program that accepts simple user input.

The DemoInput class shown in Figure 5-1 has just one method—a main() method. At the end of the line containing the main() method header is the phrase `throws Exception`. The main() methods you have written that use `System.out.println();` have not required this phrase, but programs you write using `System.in.read();` do. An **exception** is an error situation. Errors are the "exception to the rule." Unfortunately, when a program user provides input, all sorts of error situations can arise. For example, the keyboard might be disconnected or the user might enter the wrong type of data. As you become a better Java programmer, you learn to handle these exceptional situations by writing code to take appropriate action, such as issuing detailed messages that explain the problem to the user. For now, however, you can let the compiler handle the problem by **throwing the exception**, or passing the error to the operating system. The code `throws Exception` after the main() method header accomplishes this; a program that reads keyboard input will not compile without this phrase.

```
public class DemoInput
{
  public static void main(String[] args) throws Exception
  {
    char userInput;
    System.out.println("Please enter a character ");
    userInput = (char)System.in.read();
    System.out.println("You entered " + userInput);
  }
}
```

Figure 5-1 DemoInput program

Tip

You write `Exception` with an uppercase E because it is a class name. Classes, by convention, begin with uppercase letters.

In Figure 5-1, a character named userInput is declared inside the main() method of the DemoInput program. The string "Please enter a character " prints on the screen. A message requesting user input commonly is called a **prompt** because it prompts or coaches the user to enter an appropriate response.

Tip

You are not required to supply a prompt every time there is user input, but you almost always will want to do so. Unless you supply a prompt, your user will see a blank screen and won't know how to proceed.

The statement `userInput = (char)System.in.read();` in the DemoInput program accomplishes three separate tasks:

- The method call `System.in.read();` gets the input from the keyboard. The read() method accepts a byte and returns an integer.
- The cast `(char)` converts the returned integer into a character.
- The assignment `userInput =` assigns the converted character to the variable userInput.

At first, it might not make sense that `System.in.read();` returns an integer value. However, Java's creators chose to have `System.in.read();` behave this way for the following reasons:

- To the computer, all values are integers because computers store input (and everything else) as a series of 0s and 1s. The character A, for example, is stored in Unicode as 0000 0000 0100 0001, which can also be expressed as '\u0041' or decimal 65. You learned (in Chapter 2) that the ASCII code is an 8-bit code. Unicode is a 16-bit code that represents a much larger character set that includes special and international characters.
- The System.in.read() method must return a value to indicate that no input is available. For example, when you use System.in.read() to read records from a disk file, at some point the compiler reaches the end of the file and no more input is available. Java's creators decided that the System.in.read() method should return the value -1 when the compiler reaches the end of a file. To accomplish this, the read() method must have a return type of int.

The final statement in the DemoInput program shown in Figure 5-1, `System.out.println("You entered " + userInput);`, **echoes**, or repeats, the userInput character. When you write interactive programs, it is a good idea to echo the input so the user can visually confirm that the data is correct.

When you run the DemoInput program, the prompt appears on the screen. The program will not proceed any further until you type a character and press [Enter]. The read() method accepts precisely one byte of input. Therefore, you cannot enter a floating-point number or a string of characters.

Next you will write a simple program that accepts three bytes of user input and echoes them.

To write a program that accepts and echoes user input:

1. Start your text editor, and then open a new text file.
2. Type the class header for a UsersInitials class, **`public class UsersInitials`**, press **[Enter]**, type the opening curly brace for the class, and then press **[Enter]** again.
3. Type the header for the main() method, **`public static void main(String[] args) throws Exception`**, press **[Enter]**, type the opening curly brace for the main() method, and then press **[Enter]** again.
4. Type the following declarations for three character variables: **`char firstInit, middleInit, lastInit;`**.
5. On new lines, prompt the user for three initials by typing the following:

```
System.out.println("Please enter your three initials.");
System.out.println
   ("Do not use periods or spaces between initials.");
System.out.println("Press Enter when you're done.");
```

The instruction "Do not use periods or spaces between initials." is important because you will write the program to accept only three characters from the keyboard. If a user enters A.B.C., then six characters were entered—three letters and three periods. The first letter would become firstInit, the first period would become secondInit, and the second letter would become thirdInit. There would be no room to store the second period, the third letter, or the last period.

Tip

It is customary to type a println() statement all on one line. The println() statement sometimes appears on two lines in this book due to printing limitations.

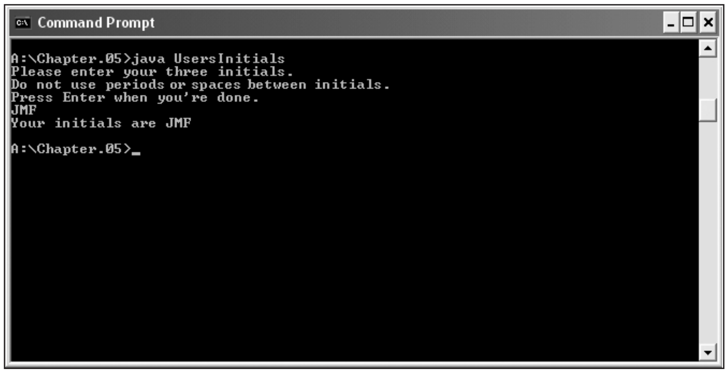
6. On new lines, type the following code to read each of the three initials into the appropriate variables:

```
firstInit = (char)System.in.read();
middleInit = (char)System.in.read();
lastInit = (char)System.in.read();
```

7. On a new line, type the following code to write the statements that will echo the three initials to the screen:

```
System.out.println("Your initials are " + firstInit +
   middleInit + lastInit);
```

8. On new lines, type the two closing curly braces that respectively close the main() method and the UsersInitials class.
9. Save the file as **UsersInitials.java** in the Chapter.05 folder on your Student Disk, and then compile and run the program. When you are prompted for three initials, enter any three characters and confirm that they are echoed to the screen correctly. Your output should look like Figure 5-2.



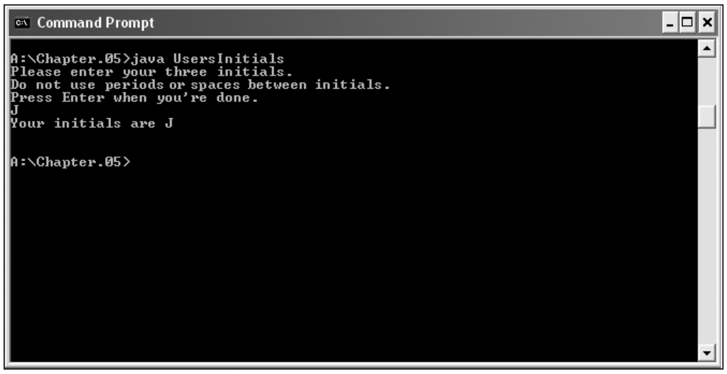
```
A:\Chapter.05>java UsersInitials
Please enter your three initials.
Do not use periods or spaces between initials.
Press Enter when you're done.
JMF
Your initials are JMF

A:\Chapter.05>_
```

Figure 5-2 Output of the UsersInitials program

The UsersInitials program works correctly as long as the user follows directions by entering three initials and pressing [Enter] only once after typing all three initials. However, just as if the user types periods between initials, a problem also occurs if the user presses [Enter] after typing each initial, as you will see next.

To demonstrate that the user should not press [Enter] after typing each initial:

1. Run the UsersInitials program again at the command prompt. When you see the prompt to enter your initials, type an initial and then press **[Enter]**. The program will terminate before you can type the second initial. The output will display only one initial, as shown in Figure 5-3.



```
A:\Chapter.05>java UsersInitials
Please enter your three initials.
Do not use periods or spaces between initials.
Press Enter when you're done.
J
Your initials are J

A:\Chapter.05>
```

Figure 5-3 Output of the UsersInitials program when the user presses [Enter] after the first initial

The problem occurs because when you use read() to accept a character from the keyboard, every key you press—including [Enter]—is accepted, one at a time. When you type your first initial, it is correctly stored in the firstInit variable. When you press [Enter] after entering the first initial, the value for [Enter] is stored in two bytes—the middleInit and the lastInit variables. When all three variables display on the screen, you see the first initial and the insertion point on a new line below the initial. The insertion point advances a line because the middleInit and lastInit together hold the [Enter] value.

Tip

The values for the two bytes occupied by [Enter] are '\u000D' and '\u000A', or decimal 13 and 10.

You can deal with this input problem by being very specific in your instructions to the user and insisting that the user type all three initials before pressing [Enter]. Alternately, you can ask the user for one initial at a time, and take care of [Enter] yourself. You can absorb the extra [Enter] key after each initial by reading it in with two read() method calls, and then not storing the bytes anywhere, as you will see next.

Tip

Depending on the version of JDK you are using, you might require only one extra read() statement to absorb [Enter]. For now, use one or two read() statements so all programs work correctly with your compiler. Later in this chapter, you will learn to use dialog boxes to eliminate the [Enter] problem.

To eliminate the [Enter] problem in the UsersInitials program:

1. In the UsersInitials.java text file, change the class name to **UsersInitials2**. Delete the following three lines of code that prompt the user for initials:
```
System.out.println("Please enter your three initials.");
System.out.println
   ("Do not use periods or spaces between initials.");
System.out.println("Press Enter when you're done.");
```
2. Replace the deleted lines with the following single statement:
   `System.out.print("Enter your first initial and press Enter. ");`.
3. Position the insertion point at the end of the read() statement that reads the firstInit variable, press **[Enter]** to start a new line, and then type the following statements to read in the two [Enter] bytes without storing them: **`System.in.read(); System.in.read();`**.
4. Press **[Enter]**, and then type the following prompt for the second initial on the new line:
   `System.out.print("Enter your second initial and press Enter. ");`.
5. Position the insertion point at the end of the statement that reads the middleInit variable, press **[Enter]** to start a new line, and then type the following statements to read [Enter] pressed after the second initial, and to prompt for the third initial:
```
System.in.read();
System.in.read();
System.out.print
   ("Enter your third initial and press Enter. ");
```

Tip

You might choose to place a final `System.in.read(); System.in.read();` statement after the statement that reads the third initial, to discard its [Enter]. Because the program doesn't accept any more input after reading the third initial, these extra read() statements will not affect program execution. However, if you add extra read() statements to absorb the last [Enter], the [Enter] following the third initial already will be discarded if you add additional input steps to this program later.

6. Save the program as **UsersInitials2.java**, compile, and run the program. Respond to each prompt by typing your initial and then pressing **[Enter]**. Your output should display your three initials correctly.

Using the JOptionPane Class for GUI Input and Output

Often referred to as **Swing components**, the classes found in the javax.swing package define GUI elements and provide alternatives to the System.in.read() and System.out.println() methods found in the java.lang package. The Swing classes are part of a more general set of GUI programming capabilities that are collectively referred to as the **Java Foundation Classes**, or **JFC** for short. JFC includes Swing component classes and selected classes from the java.awt package.

Tip

Swing will be introduced in greater detail starting in Chapter 9. In this chapter, only a few components that deal with input and output are demonstrated.

To access the Swing components used in this chapter, it is necessary to import the javax.swing package using `import javax.swing.*;`. Recall that you learned that the asterisk(*) is used as a wildcard symbol to represent all the classes in a package.

The Swing component **JOptionPane** can be used to create standard dialog boxes. These dialog boxes are small windows that ask a question, warn a user, or provide brief important user messages. As such, they provide a GUI interface to communicate with the user, as opposed to the nonwindowed standard input and output methods presented so far. These dialog boxes also provide methods that automatically handle input and output.

Three standard dialog boxes of the JOptionPane class are:

- InputDialog—prompts the user for text input
- MessageDialog—displays a user message
- ConfirmDialog—asks the user a question, with buttons for Yes, No, and Cancel responses

Input Dialog Boxes

An **input dialog box** asks a question and uses a text field for entering a response. You can create an input dialog box using the **showInputDialog() method**. There are two components or arguments with this method, the parent component and the string component. The string component is composed of a string or icon to be displayed in the dialog box. When no parent component is used, the keyword `null` is substituted.

The input dialog method returns a string that represents a user's response. In Chapter 2 you learned that a String is an object that can hold more than one character. In the example, `String firstName = "Audrey";`, the statement stores the name Audrey as a string in a variable named firstName. The ShowInputDialog() method returns a string. For example, Figure 5-4 shows an input dialog box created with the statement `String response = JOptionPane.showInputDialog(null, "Enter your first name");`. If the user types "Audrey", then clicks the OK button or presses [Enter] on the keyboard, the response string will contain "Audrey".

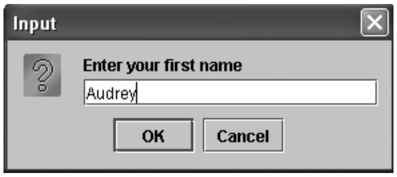


Figure 5-4 Input dialog box containing user input

Tip: When the keyword `null` is used as the first argument in the showInputDialog() method, it can be omitted entirely. Thus, `String response = JOptionPane.showInputDialog("Enter your first name");` is also correct.

An overloaded version of the showInputDialog() method contains more options; it allows the programmer flexibility in controlling the appearance of the input dialog box. The **showInputDialog() method with four arguments** can be used to display a title in the dialog box title bar and a message that describes the type of dialog box. The first two arguments (or components) are the same as in the shorter method, and the last two arguments are as follows:

- The title to be displayed in the title bar
- A class variable describing the type of dialog box—ERROR_MESSAGE, INFORMATION_MESSAGE, PLAIN_MESSAGE, QUESTION_MESSAGE, or WARNING_MESSAGE

When the statement `JOptionPane.showInputDialog(null,"What is your area code?:", "Enter area code", JOptionPane.QUESTION_MESSAGE);` is executed, it displays the input dialog box shown in Figure 5-5. Note that the title bar displays "Enter area code," and the dialog box shows a question mark icon.

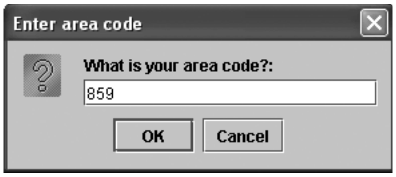
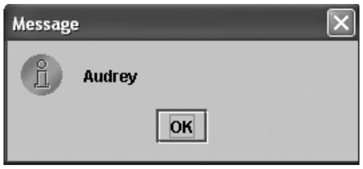


Figure 5-5 Input dialog box with four arguments

Message Dialog Boxes

The **message dialog box** uses a simple window to display information. A message dialog box is created with the **showMessageDialog() method**. The arguments are the same as with the input dialog box—the parent component (if any) and the string component composed of a string or icon to display in the box. Unlike the input dialog box, the message dialog box does not return any user response. For example, if a string variable named response holds "Audrey", then the statement `JOptionPane.showMessageDialog(null,response);` displays "Audrey" as shown in Figure 5-6.



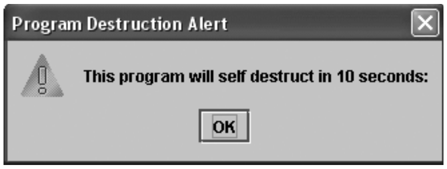
Audrey

Figure 5-6 The resulting message dialog box

You can also create a message dialog box with more options. The **showMessageDialog() method with four arguments** is used to display a title in the dialog box title bar and a message that best describes its message type. The first two arguments are the same as the shorter method, and the last two arguments are as follows:

- The title to be displayed in the title bar
- A class variable describing the type of dialog box—ERROR_MESSAGE, INFORMATION_MESSAGE, PLAIN_MESSAGE, QUESTION_MESSAGE, and WARNING_MESSAGE

When the statement `JOptionPane.showMessageDialog(null,"This program will self destruct in 10 seconds:","Program Destruction Alert", JOptionPane.WARNING_MESSAGE);` is executed, it displays a message dialog box as shown in Figure 5-7. Note that the title bar displays Program Destruction Alert, and the dialog box shows a warning sign with an exclamation mark.



This program will self destruct in 10 seconds:

Figure 5-7 Message dialog box with four arguments

Confirm Dialog Boxes

An easy way to create a confirm dialog box which displays the options Yes, No, and Cancel, is with the **showConfirmDialog() method**. Like the previous dialog boxes, the two arguments are the same—the parent component (if any) and the string component composed of a string or icon to display in the box. When you use `null` as the first argument to showConfirmDialog(), the dialog box is centered on the screen. When you use the showConfirmDialog() method, it returns one of three possible integer values, each a class variable of JOptionPane:YES_OPTION, NO_OPTION, and CANCEL_OPTION. To set an integer named option equal to the class variable NO_OPTION, you write the statement `int option = JOptionPane.NO_OPTION;`.

The statement `JOptionPane.showConfirmDialog(null, "Are you ready to quit");` displays the dialog box shown in Figure 5-8. The program will not terminate, however, without a statement to stop the program. To stop the program, you can call the exit() method from the System class with an argument of 0, for example, `System.exit(0);`. When you include this statement at the end of your dialog box programs they terminate correctly.

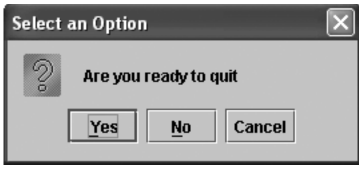
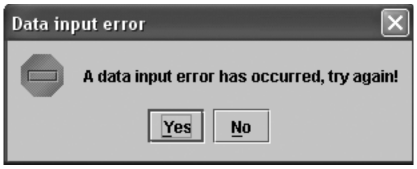


Figure 5-8 The resulting confirm dialog box

You can also create a confirm dialog box with five components. The first two arguments are the same as for the previous input dialog box and message dialog box examples. The last three components represent:

- The title to be displayed in the title bar
- An integer that indicates which option button will be shown (It should be equal to the class variables YES_NO_CANCEL_OPTION or YES_NO_OPTION.)
- An integer that describes the kind of dialog box, using the class variables ERROR_MESSAGE, INFORMATION_MESSAGE, PLAIN_MESSAGE, QUESTION MESSAGE, or WARNING_MESSAGE

When the statement `JOptionPane.showConfirmDialog(null,"A data input error has occurred, try again!", "Data input error", JOptionPane.YES_NO_OPTION, JOptionPane.ERROR_MESSAGE);` is executed, it displays an input dialog box as shown in Figure 5-9. Note that the title bar displays Data Input Error, the YES and NO buttons appear, and the dialog box shows the error message, "A data input error has occurred, try again!" It also displays the ERROR_MESSAGE type with an octagonal icon.



A data input error has occurred, try again!

Figure 5-9 Confirm dialog box with five arguments

Next you will use the dialog box methods to create a class comparable to the UserInitials class presented earlier in this chapter.

To write a program that accepts and displays user input:

1. Start your text editor, open a new file, type the import statement, **`import javax.swing.*;`**, and then press **[Enter]**.
2. Type the class header for a DialogInitials class, **`public class DialogInitials`**, press **[Enter]**, type the opening curly brace for the class, and then press **[Enter]** again.
3. Type the header for the main() method, **`public static void main(String[] args)`**, press **[Enter]**, type the opening curly brace for the main() method, and then press **[Enter]** again.
4. Type the statement to create an input dialog box and store user input in a String variable named response; **`String response = JOptionPane.showInputDialog(null, "Please enter three initials:");`**, and then press **[Enter]**.
5. Type the statement to create a message dialog box to display the user's input stored in the String response; **`JOptionPane.showMessageDialog(null, response);`**, and then press **[Enter]**.
6. Type the statement to create a confirm dialog box that asks the user to confirm that the initials entered are correct: **`JOptionPane.showConfirmDialog(null, "Are the initials correct?");`**.
7. Add the statement to close the program: **`System.exit(0);`**. Press **[Enter]**, type the closing curly braces for the main() method, press **[Enter]** a second time, and then type the closing curly brace for the class.
8. Save the file as **DialogInitials** in the Chapter.05 folder on your Student Disk, and then compile and run the program. When you are prompted for three initials, enter any three letters and confirm that they are displayed correctly. Finally, click the **YES** button when the message "Are the initials correct?" appears. The sequence of input, output, and confirmation of input are shown in Figures 5-10 through 5-12.

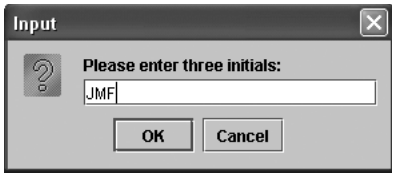


Figure 5-10 Input dialog box with initials entered

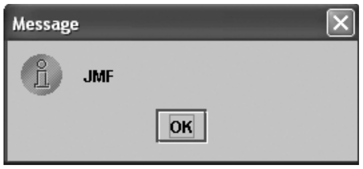


Figure 5-11 Message dialog box

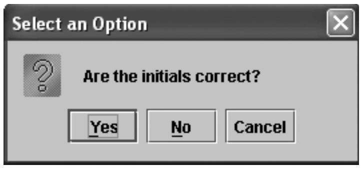


Figure 5-12 Confirm dialog box

Tip

Examples of the capabilities of dialog boxes introduced in this chapter will be expanded and used as programming structures for decision making in this and future chapters.

Drawing Flowcharts

This section is not a thorough discussion of flowcharting. Instead, it is a brief introduction, so you can use flowcharts as a visual aid in the next sections of this chapter.

When computer programmers write programs, they seldom simply sit down at a keyboard and begin typing. Programmers must plan the complex portions of programs using paper and pencil tools. Programmers often use **pseudocode**, or lists of tasks that must be accomplished, to help them plan a program's logic. Using pseudocode requires that you write down the steps needed to accomplish a given task. You write pseudocode in English; you concentrate on the logic required, and not the syntax used in any programming language. As a matter of fact, a task you pseudocode does not have to be computer related. If you have ever written a list of things you must accomplish during a day (for example; 1. Wash car, 2. Study for test, 3. Buy birthday gift for Mom), then you have written pseudocode. A **flowchart** is similar to pseudocode, but you write the steps in diagram form, as a series of shapes connected by arrows.

Tip

You learned the difference between a program's logic and its syntax in the Running a Program section of Chapter 1.

Some programmers use a variety of shapes to represent different tasks in their flowcharts, but you can draw simple flowcharts that express very complex situations using just rectangles and diamonds. You use a rectangle to represent any unconditional step, and a diamond to represent any decision. For example, Figure 5-13 shows a flowchart of a day's tasks.

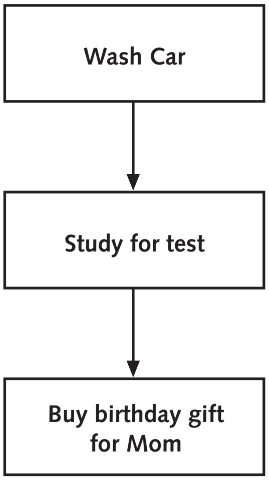


Figure 5-13 Flowchart of a day's tasks

Sometimes your days don't consist of a series of unconditional tasks—some tasks may or may not occur based on decisions you make. Using diamond shapes, flowchart creators draw paths to alternate courses of action emanating from the sides of the diamonds. Figure 5-14 shows a flowchart of a day's tasks in which some tasks are based on decisions.

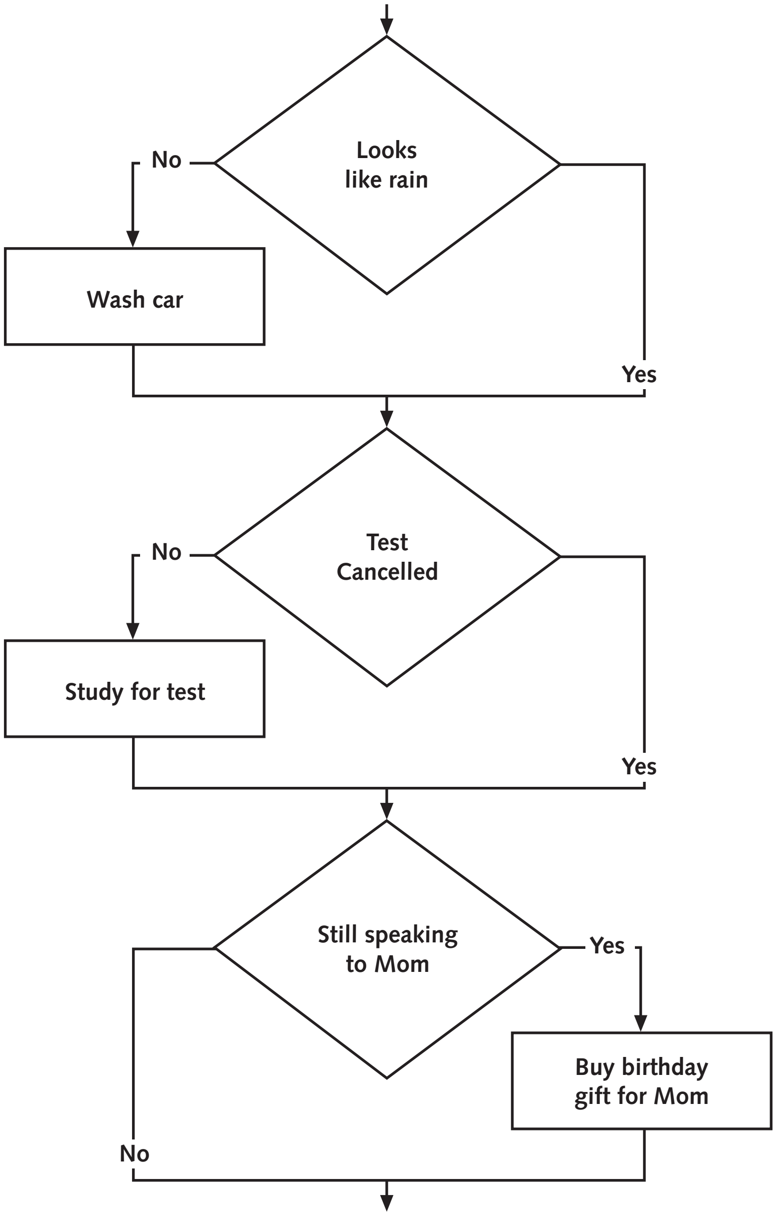


Figure 5-14 Flowchart of a day's tasks with decisions

Making Decisions with the `if` and `if...else` Structures

You can already write a program that produces different output based on input; for example, a user who types JFK into the UsersInitials program receives different output than a user who types FDR. Additionally, after you learn to write programs that can accept input, you gain a powerful new capability—you can alter the events that occur within a program based on user input. Now you can make decisions.

Making a **decision** involves choosing between alternate courses of action based on some value within a program. For example, the program that produces your paycheck can make decisions about the proper amount to withhold for taxes, the program that guides a missile can alter its course, and a program that monitors your blood pressure during surgery can determine when to sound an alarm. Making decisions is what makes computer programs seem "smart."

The value upon which a decision is made is always a Boolean value, which in turn is always one of two values—`true` or `false`. Figure 5-15 shows the logic of the decision structure. Recall from Figure 5-14 that each decision structure was couched in a yes and no answer, rather than a Boolean `true` or `false` value.

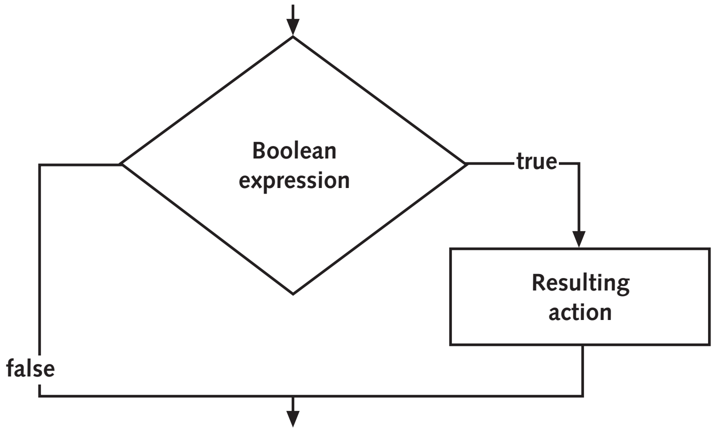


Figure 5-15 Decision structure

One statement you can use to make a decision is the **`if` statement**. For example, you can store a value in an integer variable named someVariable, and then print the value of someVariable when the user wants to see it. As you can see in Figure 5-16, you can prompt the user to enter Y or N (for "Yes" or "No") and store the response in a character variable.

```
char userResponse;
int someVariable = 512;
System.out.println
  ("Do you want to see the value of someVariable?");
System.out.println("Enter Y for yes or N for no");
char userResponse = (char)System.in.read();
```

Figure 5-16 Storing a user's response

The following is the `if` statement that makes the decision to print. Note that the double equals sign (==) is used to determine equality.

```
if(userResponse == 'Y')
  System.out.println
   ("The value of someVariable is " + someVariable);
```

Tip

Remember that you reference character values using single quotation marks.

If the userResponse variable holds the value 'Y,' then the Boolean value of the expression `userResponse == 'Y'` is `true`, and the subsequent println() statement will execute. If the value of the expression `userResponse == 'Y'` is `false`, then the println() statement will not execute. The `userResponse == 'Y'` expression will be `false` if userResponse holds anything other than 'Y,' including 'y,' 'N,' 'n,' 'A,' or any other value.

The Boolean expression `(userResponse == 'Y')` must appear within parentheses.

You are not required to leave a space between the keyword `if` and the opening parentheses, but if you do, the statement is easier to read and is less likely to be confused with a method call. Also, there is no semicolon at the end of the first line of the `if` statement `if(userResponse == 'Y')` because the statement does not end there. The statement ends after the println() call, so that is where you type the semicolon. You also could type the same statement on one line and execute it in the same manner. However, the two-line format is more conventional and easier to read, so you will usually type `if` and the Boolean expression on one line, press [Enter], and then indent a few spaces before coding the action that will occur if the Boolean expression evaluates as `true`. Be careful—when you use the two-line format, do not type a semicolon at the end of the first line, as in the following example:

```
if(userResponse == 'Y');
// Notice the incorrect semicolon here
 System.out.println("The value of someVariable is " +
 someVariable);
```

When this `if` expression is evaluated, the statement ends if it evaluates as `true`. Whether the expression evaluates as `true` or `false`, execution continues with the next independent statement that prints someVariable. In this case, because of the incorrect semicolon, the `if` statement accomplishes nothing.

Another very common programming error occurs when a programmer uses a single equals sign rather than the double equals sign when attempting to determine equivalency. The expression `userResponse = 'Y'` does not compare userResponse to 'Y.' Instead, it attempts to assign the value 'Y' to the userResponse variable. When the expression is part of an `if` statement, this assignment is illegal. The confusion arises in part because the single equals sign is used within Boolean expressions in `if` statements in many other programming languages, such as COBOL, Pascal, and BASIC. Adding to the confusion, Java programmers use the word *equals* when speaking of equivalencies. For example, you might say, "If userResponse *equals* 'Y'…" rather than "If userResponse is *equivalent* to 'Y'…"

An alternative to using a Boolean expression, such as `userResponse == 'Y'`, is to store the Boolean expression's value in a Boolean variable. For example, if userSaidYes is a Boolean variable, then `userSaidYes = (userResponse == 'Y');` compares userResponse to 'Y' and stores `true` or `false` in userSaidYes. Then you can write the `if` as `if(userSaidYes)....` This adds an extra step to the program, but makes the `if` statement more similar to an English statement.

The `if...else` Structure

Consider the following statement:

```
if(userResponse == 'Y')
System.out.println("The value of someVariable is "
 + someVariable);
```

Such a statement is sometimes called a **single-alternative `if`** because you only perform an action based on one alternative, which is the case when userResponse is 'Y.' Often you require two options for the next course of action, or a **dual-alternative `if`**. For example, if the user does not respond 'Y' to a prompt, you might want to print a message that at least acknowledges that the response was received. The **`if...else` statement** provides the mechanism to perform one action when a Boolean expression evaluates as `true`, and performs a different action when a Boolean expression evaluates as `false`. Figure 5-17 shows the logic for the `if...else` structure. Figure 5-18 shows an example of the `if...else` structure coded in Java. In Figure 5-18, the value of someVariable is printed when userResponse is equivalent to 'Y.' When userResponse is any other value, the program prints the message "Too bad."

Tip

You can code an `if` without an `else`, but it is illegal to code an `else` without an `if`.

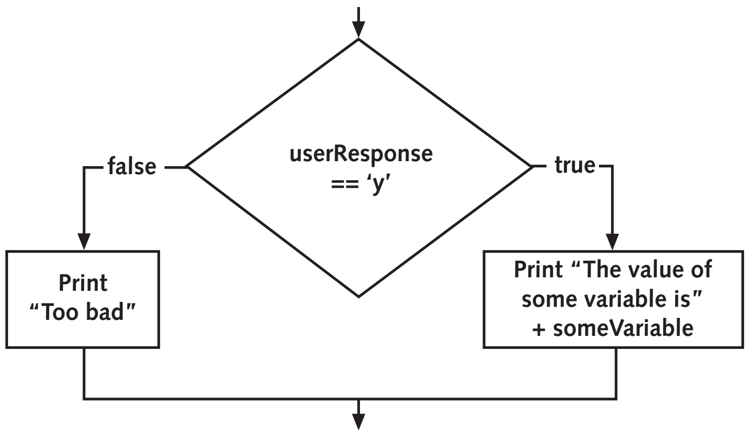


Figure 5-17 The `if...else` structure

```
if(userResponse == 'Y')
  System.out.println("The value of someVariable is"
    + someVariable);
else
  System.out.println("Too bad.");
```

Figure 5-18 A dual-alternative `if` structure

Tip: The indentation shown in the `if...else` example shown in Figure 5-18 is not required, but is standard usage. You vertically align the keyword `if` with the keyword `else`, and then indent the action statements that depend on the evaluation.

When you execute the code shown in Figure 5-18, only one of the println() statements will execute; the one that executes depends upon the evaluation of `(userResponse == 'Y')`. Each println() statement is a complete statement, so each statement ends with a semicolon.

Next you will start writing a program for Event Handlers Incorporated that determines which of several employees will be assigned to manage a client's scheduled event. To begin, you will prompt the user to answer a question about the event type, and then the program will display the name of the manager who handles such events. There are two event types: corporate events, handled by Dustin Britt; and private events, handled by Carmen Lindsey.

To write a program that chooses between two managers:

1. Open a new text file, and then enter the code to choose a manager:

```
public class ChooseManager
{
 public static void main(String[] args) throws Exception
  {
```

2. On a new line, declare a variable that will hold the type of event by typing **`char eventType;`**.
3. On a new line, type the three-line prompt that explains what is expected of the user:

```
System.out.println
 ("Enter type of event you are scheduling");
System.out.println("C for a corporate event");
System.out.println("P for a private event");
```

4. On a new line, type the statement that reads in the type of event:

```
eventType = (char)System.in.read();.
```

5. On a new line, type the print() statement that explains the output:

```
System.out.print("The manager for this event will be ");.
```

6. Code the `if...else` structure to determine which of two managers will be assigned to the event.

```
if(eventType == 'C')
 System.out.println("Dustin Britt");
else
 System.out.println("Carmen Lindsey");
```

7. Type the two closing curly braces to end the main() method and the ChooseManager class.
8. Save the program as **ChooseManager.java** in the Chapter.05 folder on your Student Disk, then compile and run the program. Confirm that the program selects the correct manager when you choose C for a corporate event or P for a private event.

USING COMPOUND STATEMENTS IN AN `if` OR `if...else` STRUCTURE

Often there is more than one action to take following the evaluation of a Boolean expression within an `if` statement. For example, you might want to print several separate lines of output or perform several mathematical calculations. To execute more than one statement that depends on the evaluation of a Boolean expression, you use a pair of curly braces to place the dependent statements within a block. For example, the program segment shown in Figure 5-19 determines whether an employee has worked more than 40 hours in a single week, and if so, the program computes regular and overtime salary, and then prints the results.

Tip

When you create a block, you do not have to place multiple statements within it. It is perfectly legal to block a single statement.

```
if(hoursWorked > 40)
{
  regularPay = 40 * rate;
  overTimePay = (hoursWorked - 40) * 1.5 * rate;
    // Time and a half for hours over 40
  System.out.println("Regular pay is " + regularPay);
  System.out.println("Overtime pay is " + overTimePay);
} // The if structure ends here
```

Figure 5-19 An `if` statement with multiple dependent statements

If you compare Figures 5-19 and 5-20, you will see that in Figure 5-19, the regularPay calculation, the overTimePay calculation, and the println() statement are executed only when `hoursWorked > 40` is `true`. In Figure 5-20, the curly braces are omitted. Within the program in Figure 5-20, when `hoursWorked > 40` is `true`, regularPay is calculated and the `if` expression ends. The next three statements that compute overTimePay and print the results always execute every time the program runs, no matter what value is stored in hours. These last three statements are not dependent on the `if` statement; they are independent, stand-alone statements. The indentation might be deceiving; it looks as though four statements depend on the `if` statement, but indentation does not cause statements following an `if` statement to be dependent. Rather, curly braces are required if the four statements must be treated as a block.

```
if(hoursWorked > 40)
  regularPay = 40 * rate;  // The if structure ends here
  overTimePay = (hoursWorked - 40) * 1.5 * rate;
  System.out.println("Regular pay is " + regularPay);
  System.out.println("Overtime pay is " + overTimePay);
```

Figure 5-20 `if` statement with a single dependent statement

The code shown in Figure 5-20 might not compile if regularPay is not assigned a value—the compiler will recognize that you are attempting to print the value of regularPay without calculating it. However, if you have assigned a value to regularPay, you can compile the program, but the output still will not be what you intended. Within the code segment shown in Figure 5-20, if hoursWorked is greater than 40, then the program properly calculates both regular and overtime pay. Because `hoursWorked > 40` is `true`, the regularPay calculation is made. The overTimePay calculation and the println() statements will execute as well because they are just statements that always execute and do not depend on the `if` statement.

However, in Figure 5-20, if the hoursWorked value is 40 or less—30, for example—then the regularPay calculation will not execute (it executes only `if(hoursWorked > 40)`), but the next three independent statements will execute. The variable regularPay will hold whatever value you have previously assigned to it—0.0, for example—and the program will calculate the value of overTimePay as a negative number (because 30 - 40 results in -10). Therefore, the output will be incorrect.

Just as you can block statements to depend on an `if`, you can also block statements to depend on an `else`. Figure 5-21 shows an `if` structure with two dependent statements and an `else` with two dependent statements. The program executes the final two println() statements without regard to the hoursWorked variable's value; the println() statements are not part of the `if` structure.

```
if(hoursWorked > 40)
{
  regularPay = 40 * rate;
  overTimePay = (hoursWorked - 40) * 1.5 * rate;
  // Time and a half for hours over 40
}
else
{
  regularPay = hours * rate;
  overTimePay = 0.0;
}
System.out.println("Regular pay is " + regularPay);
System.out.println("Overtime pay is " + overTimePay);
```

Figure 5-21 An `if...else` statement with multiple dependent statements

Next you will create an Event class. Each Event object includes two data fields: the type of event, and the base price Event Handlers charges per hour for the event type. The Event class also contains a constructor method and get methods for the two fields.

To create the Event class:

1. Open a new text file, and type the class header for the Event class, **`public class Event.`** Press **[Enter]** and then type the opening curly brace for the class on the new line.
2. Type the following declarations for two data fields to hold the type of event and the minimum hourly rate that Event Handlers charges:

   ```
   private char eventType;
   private double eventMinRate;
   ```

3. Type the following constructor for the Event class. The constructor will require two arguments with which you will fill the two data fields.

   ```
   public Event(char event, double rate)
   {
    eventType = event;
    eventMinRate = rate;
   }
   ```

4. On new lines, type the following two get methods that return the field values:

```
public char getEventType()
{
 return eventType;
}
public double getEventMinRate()
{
 return eventMinRate;
}
```

5. Type the closing curly brace for the class.
6. Save the file as **Event.java** in the Chapter.05 folder on your Student Disk, then compile the file and correct any errors.

Now that you have created an Event class, you can modify the ChooseManager program to perform multiple tasks based on user input. You will display a message to indicate which manager is assigned to the event, and you will also instantiate a unique Event object, with different minimum rates to charge based on the type of event.

To modify the ChooseManager program:

1. Open the **ChooseManager.java** text file from the Chapter.05 folder on your Student Disk, and change the class name to **ChooseManager2**. You will declare two constants to hold the corporate hourly rate and the private hourly rate. If these hourly rates change in the future, they will be easy to locate at the top of the file, where you can change their values.
2. Position the insertion point just after the opening curly brace for the ChooseManager2 class, press **[Enter]** to start a new line, and then type the following two constants:

```
static final double CORP_RATE = 75.99;
static final double PRI_RATE = 47.99;
```

   Within the main() method of the ChooseManager2 class, you will define an Event object named anEvent. You do not want to construct the Event object until you discover whether it will be a corporate or private event; you simply want to declare it now.

3. Position the insertion point to the right of the statement that declares the eventType character variable, press **[Enter]** to start a new line, and then type the event declaration as **`Event anEvent;`**.
4. Next type the following lines to modify the `if...else` structure that currently prints a manager's name, so that the `if` and `else` each control a block of two statements. The first statement in each block still prints the manager's name. The second statement constructs an appropriate Event object.

```
if(eventType == 'C')
{
  System.out.println("Dustin Britt");
  anEvent = new Event(eventType, CORP_RATE);
}
else
{
  System.out.println("Carmen Lindsey");
  anEvent = new Event(eventType, PRI_RATE);
}
```

5. To confirm that the event was constructed properly, type the following two println() statements immediately after the closing brace for the `if...else` structure:

```
System.out.println("Event type is " + anEvent.
getEventType());
System.out.println("Minimum rate charged is $" +
anEvent.getEventMinRate());
```

6. Save the program as **ChooseManager2.java**. Compile and run the program several times with different input at the prompt. Confirm that the output shows that the event has the correct manager, type, and rate based on how you respond to the prompt (with C or P).

NESTING `if` AND `if...else` STATEMENTS

Within an `if` or an `else` statement, you can code as many dependent statements as you need, including other `if` and `else` statements. Just as spoons are nested inside each other in a drawer, statements with an `if` inside another `if` commonly are called **nested `if` statements**. Nested `if` statements are particularly useful when two conditions must be met before some action is taken.

For example, suppose you want to pay a $50 bonus to a salesperson only if the salesperson sells more than three items that total more than $1,000 in value during a specified time. Figure 5-22 shows the logic for this situation. Figure 5-23 shows the code to solve the problem.

Notice there are no semicolons in the code shown in Figure 5-23 until after the `bonus = 50;` statement. The expression `itemsSold > 3` is evaluated. If this expression is `true`, then the program evaluates the second Boolean expression `(totalValue > 1000)`. If that expression is also `true`, then the bonus assignment executes and the `if` structure ends.

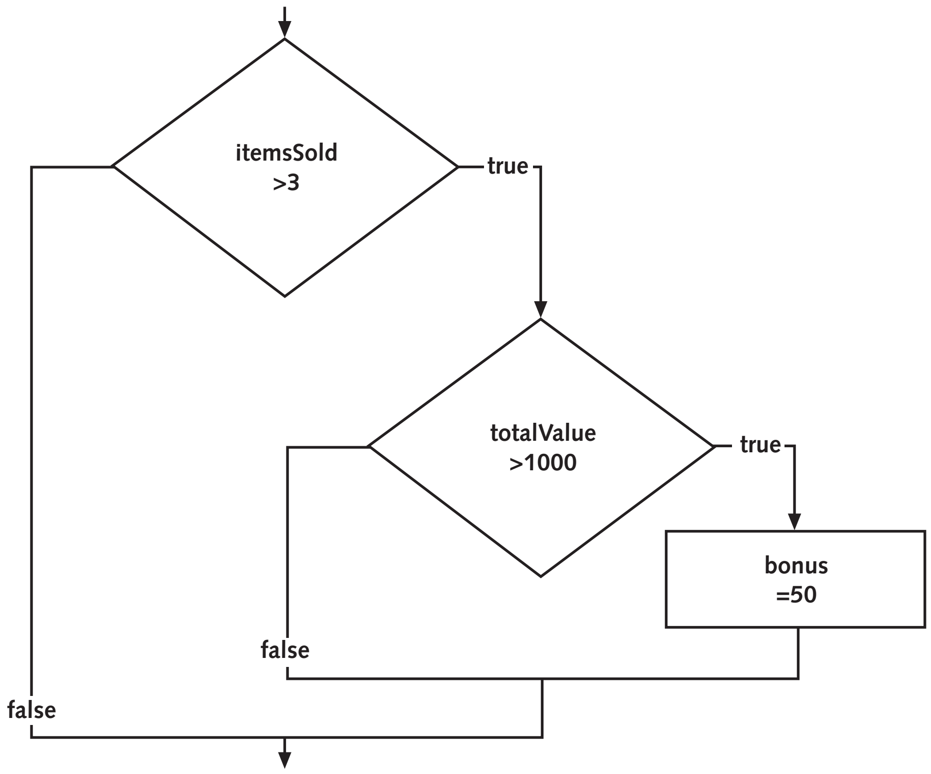


Figure 5-22 Nested `if` structure

```
if(itemsSold > 3)
  if(totalValue > 1000)
    bonus = 50;
```

Figure 5-23 Nested `if` statement

Using AND and OR Operators

For an alternative to nested `if` statements, you can use the **AND operator** within a Boolean expression to determine whether two expressions are both `true`. The AND operator is written as two ampersands (&&). For example, the code shown in Figure 5-24 works exactly the same as the code shown in Figure 5-23. The itemsSold variable is tested, and if it is greater than 3, then the totalValue is tested. If totalValue is greater than $1000, then the bonus is set to $50.

```
if(itemsSold > 3 && totalValue > 1000)
  bonus = 50;
```

Figure 5-24 Using the AND operator

You are never required to use the AND operator because using nested `if` statements always achieves the same result, but using the AND operator often makes your code more concise, less error prone, and easier to understand.

It is important to note that when you use the AND operator, you must include a complete Boolean expression on each side of the && operator. If you want to set a bonus to $400 if a saleAmount is both over $1000 and under $5000, the correct statement is `if(saleAmount > 1000 && saleAmount < 5000) bonus = 400;`. Even though the saleAmount variable is used on both sides of the AND expression, the statement `if(saleAmount > 1000 && < 5000)...` is incorrect and will not compile.

With the AND operator, both Boolean expressions must be `true` before the action in the statement can occur. You can use the **OR operator**, which is written as `||`, when you want some action to occur, even if only one of two conditions is true. For example, if you want to give a bonus of $200 if a salesperson satisfies at least one of two conditions—selling more than 100 items or selling any number of items that total more than $3000 in value—then you can write the code using either of the ways shown in Figure 5-25. Figure 5-26 shows the program logic.

Tip

A common use of the OR operator is to decide to take action whether a character variable is uppercase or lowercase, as in `if(selection == 'A' || selection == 'a') ...`. The subsequent action occurs whether the selection variable holds an uppercase or lowercase A.

```
// Using two ifs
if(itemsSold > 100)
  bonus = 200;
else if(totalValue > 3000)
  bonus = 200;
// Using the OR operator
if(itemsSold > 100 || totalValue > 3000)
  bonus = 200;
```

Figure 5-25 Code using two `if` statements and the OR operator

Sometimes situations arise in which there are more than two possible courses of action to take. Consider a situation in which salespeople can receive one of three possible commission rates based on their sales. For example, a sale totaling $1001 or more earns the salesperson an eight percent commission, a sale totaling $500 to $1000 earns six percent of the sale amount, and any sale totaling $499 or less earns five percent. Using three separate `if` statements to test single Boolean expressions results in some incorrect commissions. Examine the code shown in Figure 5-27.

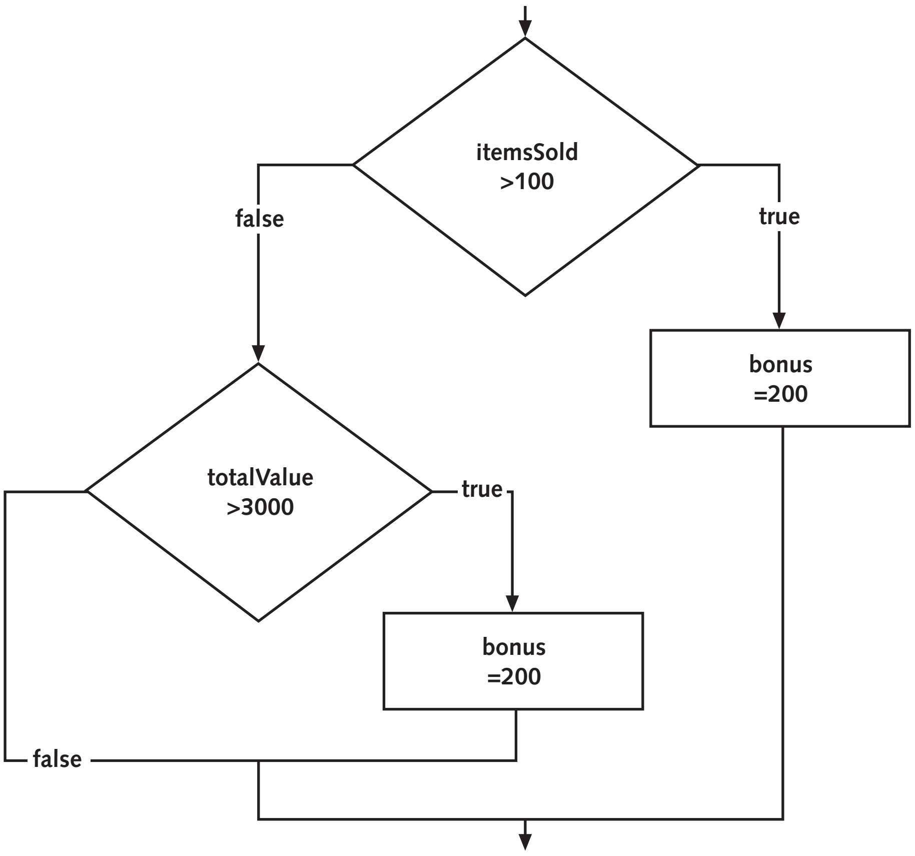


Figure 5-26 Diagram of the OR logic

```
if(saleAmount > 1000)
  commRate = .08;
if(saleAmount > 500)
  commRate = .06;
if(saleAmount <= 500)
  commRate = .05;
System.out.println("Commission rate is " + commRate);
```

Figure 5-27 Incorrect assignment of three commissions

Tip

As long as you are dealing with whole dollar amounts, the expression `if(saleAmount > 1000)` can be expressed just as well as `if(saleAmount >= 1001)`. Additionally, `if(1000 < saleAmount)` and `if(1001 <= saleAmount)` have the same meaning. Use whichever has the clearest meaning for you.

Using the code shown in Figure 5-27, if a saleAmount is $5000, the first `if` statement executes. The Boolean expression `(saleAmount > 1000)` evaluates as `true`, and .08 is correctly assigned to commRate. However, when a saleAmount is $5000, the next `if` expression, `(saleAmount > 500)`, also evaluates as `true`, so the commRate, which was eight percent, is incorrectly reset to six percent.

A partial solution to this problem is to use an `else` statement following the `if(saleAmount >= 1000)` expression, as shown in Figure 5-28.

```
if(saleAmount > 1000)
  commRate = .08;
else if(saleAmount > 500)  // Notice the else
  commRate = .06;
if(saleAmount <= 500)
  commRate = .05;
System.out.println("Commission rate is " + commRate);
```

Figure 5-28 Inefficient assignment of three commissions

Tip

You can place and indent the `if` following an `else`, but a program with many nested `if...else` combinations soon grows very long and "deep," and with indentations, later statements in the nest would move farther and farther to the right on the page. For easier-to-read code, Java programmers commonly place each `else` and its subsequent `if` on the same line.

With the new code in Figure 5-28, when the saleAmount is $5000, the expression `(saleAmount > 1000)` is `true` and the commRate becomes eight percent. When the saleAmount is not greater than $1000, the `else` statement executes and correctly sets the commRate to six percent.

The code shown in Figure 5-28 works, but it is somewhat inefficient. When the saleAmount is any amount over $500, either the first `if` sets commRate to eight percent for amounts over $500, or its `else` sets commRate to six percent for amounts over $500. The Boolean value tested in the next statement, `if(saleAmount <= 500)`, is always `false`. Rather than unconditionally asking `if(saleAmount <= 500)`, it's easier to use an `else`. If the saleAmount is not over $1000 and it is also not over $500, it must, by default, be less than or equal to 500. Figure 5-29 shows this improved logic and Figure 5-30 shows its code. Within a nested `if...else`, it is most efficient to ask the most likely question first. In other words, if you know that most saleAmount values are over 1000, compare saleAmount to that value first. If, however, you know that most saleAmounts are small, you should ask `if(saleAmount <= 500)` first.

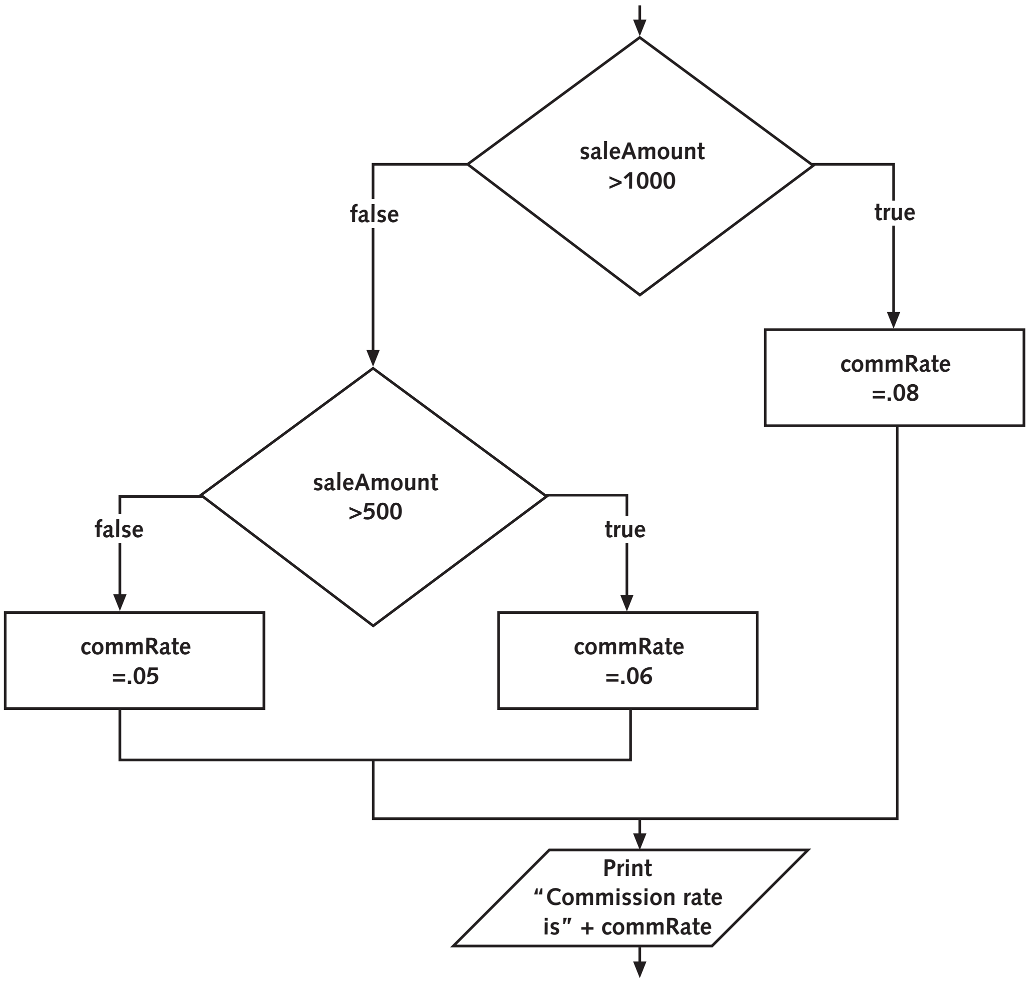


Figure 5-29 Improved OR logic

```
if(saleAmount >= 1000)
  commRate = .08;
else if(saleAmount > 500)
  commRate = .06;
else commRate = .05;
System.out.println("Commission rate is " + commRate);
```

Figure 5-30 Correct assignment of three commissions

Currently, the ChooseManager2 program identifies an event as a corporate event and assigns an appropriate manager and rate when the user enters C at the event-type prompt. When the user enters any other character, the event is considered to be private. Next you will improve the ChooseManager2 program so that if the user does not enter either C or P, an Event object with an "invalid" X-type is instantiated.

To improve the ChooseManager2 program:

1. If necessary, start your text editor and open the **ChooseManager2.java** file from the Chapter.05 folder on your Student Disk, and then change the class name to **ChooseManager3**.
2. Change the `if...else` structure that tests the eventType so that it becomes a nested `if...else` with three possibilities. When the user inputs anything other than C or P, display an error message and create an Event object with a code 'X' for the eventType and a rate of 0.0 for the minEventRate.

Tip

Remember that the Event constructor that you created earlier in this chapter requires both a character and a double argument.

```
if(eventType == 'C')
{
 System.out.println("Dustin Britt");
 anEvent = new Event(eventType, CORP_RATE);
}
else if(eventType == 'P')
{
 System.out.println("Carmen Lindsey");
 anEvent = new Event(eventType, PRI_RATE);
}
else
{
 System.out.println("Invalid entry!");
 anEvent = new Event('X',0.0);
}
```

3. Save the program as **ChooseManager3.java**, compile it, and then run the program several times to confirm that user responses of C or P result in valid Event objects, and that other responses result in an error message and an event of type X.

USING THE `switch` STATEMENT

By nesting a series of `if` and `else` statements, you can choose from any number of alternatives. For example, suppose you want to print a student's class year based on a stored number. Figure 5-31 shows the program.

```
if(year == 1)
  System.out.println("Freshman");
else if(year == 2)
  System.out.println("Sophomore");
else if(year == 3)
  System.out.println("Junior");
else if(year == 4)
  System.out.println("Senior");
else System.out.println("Invalid year");
```

Figure 5-31 Multiple alternatives

An alternative to the series of nested `if` statements is to use the `switch` statement. The **`switch` statement** is useful when you need to test a single variable against a series of exact integer or character values. The `switch` structure uses four keywords:

- `switch` starts the structure and is followed immediately by a test expression enclosed in parentheses
- `case` is followed by one of the possible values for the test expression and a colon
- `break` optionally terminates a switch structure at the end of each case
- `default` optionally is used prior to any action that should occur if the test variable does not match any case

Tip

You are not required to list the case values in ascending order as shown here. It is most efficient to list the most common case first, instead of the case with the lowest value.

```
switch(year)
{
  case 1:
    System.out.println("Freshman");
    break;
  case 2:
    System.out.println("Sophomore");
    break;
  case 3:
    System.out.println("Junior");
    break;
  case 4:
    System.out.println("Senior");
    break;
  default:
    System.out.println("Invalid year");
}
```

Figure 5-32 Sample `case` structure

Figure 5-32 shows the `case` structure used to print the four school years.

The `switch` structure shown in Figure 5-32 begins by evaluating the year variable shown in the `switch` statement. If the year is equal to the first `case` value, which is 1, then the statement that prints "Freshman" will execute. The `break` statement bypasses the rest of the `switch` structure, and execution continues with any statement after the closing curly brace of the `switch` structure.

If the year variable is not equivalent to the first `case` value of 1, then the next case value is compared, and so on. If the year variable does not contain the same value as any of the `case` statements, then the `default` statement or statements execute.

You can leave out the `break` statements in a `switch` structure. However, when you omit the `break`, if the program finds a match for the test variable, then all the statements within the `switch` statement from that point forward will execute. For example, if you omit each `break` statement in the code shown in Figure 5-32, when the year is 3, the first two cases will be bypassed, but "Junior," "Senior," and "Invalid year" all will print. You should intentionally omit the `break` statements if you want all subsequent cases to execute once the test variable is matched.

You are never required to use a `switch` structure; you can always achieve the same results with nested `if` statements. The `switch` structure is simply convenient to use when there are several alternate courses of action which depend on a single integer or character variable. Additionally, it makes sense to use `switch` only when there are a reasonable number of specific matching values to be tested. For example, if every sale amount from $1 to $500 requires a five percent commission, it is not reasonable to test every possible dollar amount using the following code:

```
switch(saleAmount)
{
 case 1:
  commRate = .05;
  break;
 case 2:
  commRate = .05;
  break;
 case 3:
  commRate = .05;
  break;
 // ...and so on for several hundred more cases
}
```

Because 500 different dollar values result in the same commission, one test—`if(saleAmount <= 500)`—is far more reasonable than listing 500 separate cases.

Next you will modify the ChooseManager3 program to account for a new type of event. Besides corporate and private, there will be special rates for nonprofit organizations. The entered code for nonprofit events will be N, and Robin Armanetti will be the manager assigned to these events. You will convert your nested `if` statements to a `switch` structure.

To convert the ChooseManager3 decision-making process to a switch structure:

1. Open the **ChooseManager3.java** file and change the class name to **ChooseManager4**. Position the insertion point at the end of the statement that declares the constant for PRI_RATE, press **[Enter]** to start a new line, and then type the constant for NON_PROF_RATE, **`static final double NON_PROF_RATE = 40.99;`**.
2. To the list of current prompts, add the following prompt to tell the user to enter N for nonprofit organization events:

   ```
   System.out.println("N for non-profit event");
   ```
3. Delete the `if...else` statements that presently are used to determine whether the user entered C or P, and then replace them with the following `switch` structure:

   ```
   switch(eventType)
   {
    case 'C':
     System.out.println("Dustin Britt");
     anEvent = new Event(eventType, CORP_RATE);
     break;
    case 'P':
     System.out.println("Carmen Lindsey");
     anEvent = new Event(eventType, PRI_RATE);
     break;
    case 'N':
     System.out.println("Robin Armanetti");
     anEvent = new Event(eventType, NON_PROF_RATE);
     break;
    default:
     System.out.println("Invalid entry!");
     anEvent = new Event('X',0.0);
   }
   ```

Tip

Remember from Chapter 2 that characters are stored as integers. That is why they are allowed as the case variables in a `switch` statement.

4. Save the file as **ChooseManager4.java**, compile, and test the program. Make sure the correct output appears when you enter C, P, N, or some other value as keyboard input.

Using the Conditional and NOT Operators

Java provides one more way to make decisions. The **conditional operator** requires three expressions separated with a question mark and a colon, and it is used as an abbreviated version of the `if...else` structure. As with the `switch` structure, you are never required to use the conditional operator; it is simply a convenient shortcut. The syntax of the conditional operator is `testExpression ? true Result : false Result;`.

The first expression, testExpression, is a Boolean expression that is evaluated as `true` or `false`. If it is `true`, then the entire conditional expression takes on the value of the expression following the question mark (`trueResult`). If the value of the testExpression is `false`, then the entire expression takes on the value of `falseResult`. For example, suppose that you want to assign the smallest price to a sale item. Let the variable *a* be the advertised price and the variable *b* be the discounted price on the sale tag. The expression for assigning the smallest cost is `smallerNum = (a < b) ? a : b;`. When evaluating the expression `a < b`, where *a* is less than *b*, the entire conditional expression takes the value of *a*, which then is assigned to smallerNum. If *a* is not less than *b*, then the expression assumes the value of *b*, and *b* is assigned to smallerNum.

You use the **NOT operator**, which is written as the exclamation point (!), to negate the result of any Boolean expression. Any expression that evaluates as `true` becomes `false` when preceded by the NOT operator, and accordingly, any `false` expression preceded by the NOT operator becomes `true`.

For example, suppose a monthly car insurance premium is $200 if the driver is age 25 or younger, and $125 if the driver is age 26 or older. Each of the following `if` statements (which have been placed on single lines for convenience) correctly assigns the premium values:

```
if(age <= 25)  premium = 200;    else premium = 125;
if(!(age <= 25)) premium = 125;   else premium = 200;
if(age >= 26) premium = 125;   else premium = 200;
if(!(age >= 26)) premium = 200;    else premium = 125;
```

The statements with the NOT operator are somewhat harder to read, particularly because they require the double set of parentheses, but the result of the decision-making process is the same in each case. Using the NOT operator is clearer when the value of a Boolean variable is tested. For example, a variable initialized as `Boolean oldEnough = (age >= 25);` can become part of the relatively easy-to-read expression `if(!oldEnough)....`

UNDERSTANDING PRECEDENCE

You learned in Chapter 2 that operations have higher and lower precedences. For example, within an arithmetic expression, multiplication and division are always performed prior to addition or subtraction. Table 5-2 shows the precedence of the operators you have used so far.

Table 5-1 Operator precedence for operators used so far

Precedence	Operator(s)	Symbol(s)
Highest	Multiplication, division	* / %
	Addition, subtraction	+ -
	Relational	> < >= <=
	Equality	== !=
	Logical AND	&&
	Logical OR	||
	Conditional	?:
Lowest	Assignment	=

In general, the order of precedence agrees with common algebraic usage. For example, in any mathematical expression such as `x = a + b`, the arithmetic is done first and the assignment is done last, as you would expect. The relationship of && and || might not be as obvious. Consider the program segment shown in Figure 5-33 and try to predict its output.

```
int tickets = 4;
int age = 40;
char gender = 'F';
if(tickets > 3 || age < 25 && gender == 'M')
  System.out.println("Do not insure");
if((tickets > 3 || age < 25) && gender == 'M')
  System.out.println("Bad risk");
```

Figure 5-33 Demonstrating AND and OR operator precedence

With the first `if` statement, the AND operator takes precedence over the OR operator, so `age < 25 && gender == 'M'` is evaluated first. The value is `false` because age is not less than 25 and gender is not 'M.' So the expression is reduced to "tickets > 3" or `false`. Because the value of the tickets variable is greater than 3, the entire expression is `true`, and "Do not insure" is printed.

Tip: Even though the AND operator is evaluated first in the expression `age < 25 && gender == 'M' || tickets > 3`, you can add extra parentheses, as in `(age < 25 && gender == 'M') || tickets > 3`. The outcome is the same, but the intent is clearer to someone reading your code.

In the second `if` statement shown in Figure 5-33, parentheses have been added so the OR operator is evaluated first. The expression `tickets > 3 || age < 25` is `true` because tickets is greater than 3. So the expression evolves to `true && gender == 'M'.` Because gender is not 'M,' the value of the entire expression is `false`, and the "Bad risk" statement does not print. The following two conventions are important to keep in mind:

- The order in which you use operators makes a difference.

- You can always use parentheses to change precedence or make your intentions clearer.

Chapter Summary

- An interactive program accepts values at run time. When you write interactive programs, it is often a good idea to echo the input so the user can confirm visually that the data entered is correct. The message that requests user input commonly is called a prompt.
- An exception is an error situation. You can let the compiler handle the problem by throwing the exception, or you can pass the error to the operating system. When the System.in.read() method accepts input from the keyboard it `throws Exception`. The method System.in.read() accepts a byte from the keyboard and returns an integer value.
- The JOptionPane component, part of the javax.swing package, can be used to create standard dialog boxes. These dialog boxes are small windows that ask a question, warn a user, or provide brief important user messages. As such, they provide a GUI interface to communicate with the user as opposed to the nonwindowed standard input and output methods presented thus far.
- Three standard dialog boxes of the JOptionPane class are: an input dialog box that prompts the user for text input, a message dialog box that displays a user message, and a confirm dialog box that asks the user a question using buttons for Yes, No, and Cancel responses.
- Making a decision involves choosing between two alternate courses of action based on some value within a program. You can use the `if` statement to make a decision based on a Boolean expression that evaluates as `true` or `false`. If the Boolean expression enclosed in parentheses within an `if` statement is `true`, then the subsequent statement or block will execute.
- A single-alternative `if` performs an action based on one alternative; a dual-alternative `if`, or `if...else`, provides the mechanism for performing one action when a Boolean expression evaluates as `true`. When a Boolean expression evaluates as `false`, a different action occurs.
- To execute more than one statement that depends on the evaluation of a Boolean expression, you use a pair of curly braces to place the dependent statements within a block. Within an `if` or an `else` statement, you can code as many dependent statements as you need, including other `if` and `else` statements. Nested `if` statements are particularly useful when two conditions must be met before some action occurs.
- You can use the AND operator (&&) within a Boolean expression to determine whether two expressions are both `true`. You use the OR operator (||) when you want to carry out some action even if only one of two conditions is `true`.
- You use the `switch` statement to test a single variable against a series of exact integer or character values.

- The conditional operator requires three expressions, a question mark, and a colon, and it is used as an abbreviated version of the `if...else` statement.
- You use the NOT operator (!) to negate the result of any Boolean expression.
- Operator precedence makes a difference. You can always use parentheses to change precedence or make your intentions clearer.

REVIEW QUESTIONS

1. Which of the following is typically used in a flowchart to indicate a decision?
   a. square
   b. rectangle
   c. diamond
   d. oval
2. A message that requests user input is commonly called a ________________.
   a. coach
   b. prompt
   c. hint
   d. port
3. Which of the following is not a type of `if` statement?
   a. single-alternative `if`
   b. dual-alternative `if`
   c. double-alternative `if`
   d. nested `if`
4. Standard dialog boxes of the JOptionPane class include ________________.
   a. InputDialog—Prompts the user for text input
   b. MessageDialog—Displays a user message
   c. ConfirmationDialog—Asks the user a question, with buttons for Yes, No, and Cancel responses
   d. all of the above
5. The JOptionPane class contains methods to create a(n) ________________.
   a. input dialog box
   b. message dialog box
   c. confirm dialog box
   d. all the above

6. An easy way to create a Yes, No, Cancel dialog box is with the ________________.
   a. `showConfirmDialog()` method
   b. `showInputDialog()` method
   c. `showMessDialog()` method
   d. none of the above
7. A decision is based on a(n) ________________ value.
   a. Boolean
   b. absolute
   c. definitive
   d. convoluted
8. The value of `(4 > 7)` is ________________.
   a. 4
   b. 7
   c. `true`
   d. `false`
9. Assuming the variable q has been assigned the value 3, which of the following statements prints `XXX`?
   a. `if(q > 0) System.out.println("XXX");`
   b. `if(q > 7); System.out.println("XXX");`
   c. Both of the above statements print `XXX`.
   d. Neither of the above statements prints `XXX`.
10. What is the output of the following code segment?

```
t = 10;
if(t > 7)
{
 System.out.println("AAA");
 System.out.println("BBB");
}
```

   a. `AAA`
   b. `BBB`
   c. `AAA`
      `BBB`
   d. nothing

11. When you code an `if` statement within another `if` statement, as in `if(a > b) if(c > d) x = 0;`, then the `if` statements are ________.
    a. notched
    b. nestled
    c. nested
    d. sheltered
12. The operator that combines two conditions into a single Boolean value that is true when both of the conditions are `true` is ________.
    a. $$
    b. !!
    c. ||
    d. &&
13. The operator that combines two conditions into a single Boolean value that is true when at least one of the conditions is `true` is ________.
    a. $$
    b. !!
    c. ||
    d. &&
14. Assuming a variable f has been initialized to 5, which of the following statements sets g to 0?
    a. `if(f > 6 || f == 5) g = 0;`
    b. `if(f < 3 || f > 4) g = 0;`
    c. `if(f >= 0 || f < 2) g = 0;`
    d. All of the above statements set g to 0.
15. Which if the following groups has the lowest operator precedence?
    a. Relational
    b. Equality
    c. Addition
    d. Logical OR
16. You can use the ________ statement to terminate a `case` in a `switch` structure.
    a. `switch`
    b. `end`
    c. `case`
    d. `break`

17. The `switch` argument within a `switch` structure requires a(n) ________________.
    a. integer value
    b. character value
    c. double value
    d. integer or character value
18. Assuming a variable w has been assigned the value 15, then the statement `w == 15 ? x = 2 : x = 0;` assigns ________________.
    a. 15 to w
    b. 2 to x
    c. 0 to x
    d. nothing
19. Assuming a variable y has been assigned the value 6, then the value of `!(y < 7)` is ________________.
    a. 6
    b. 7
    c. `true`
    d. `false`
20. Assuming `a = 5` and `b = 9`, then the value of `a > 0 &&  b < 10 || b > 1` is ________________.
    a. 5
    b. 9
    c. `true`
    d. `false`

Exercises

In the following exercises, save each program that you create in the Chapter.05 folder on your Student Disk.

1. a. Write a program that prompts the user for a four-character password, accepts four characters, and then echoes the characters to the screen. Save the program as **Password.java** in the Chapter.05 folder on your Student Disk.
   b. Write a program that prompts the user for a four-character password, accepts four characters, and then echoes the characters to the screen. Test the first character. If it is B, issue a message that the password is valid; otherwise issue a message that the password is not valid. Save the program as **Password.java** in the Chapter.05 folder on your Student Disk.

c. Write a program that prompts the user for a four-character password, accepts four characters, and then echoes the characters to the screen. Test all four characters. If the characters spell BOLT, then issue a message that the password is valid; otherwise issue a message that the password is not valid. Save the program as **PasswordC.java** in your Chapter.05 folder on your Student Disk.

2. a. Write a program for a furniture company. Ask the user to choose P for pine, O for oak, or M for mahogany. Show the price of a table manufactured with the chosen wood. Pine tables cost $100, oak tables cost $225, and mahogany tables cost $310. Save the program as **Furniture.java** in the Chapter.05 folder on your Student Disk.

   b. Add a prompt to the program you wrote in Exercise 2a to ask the user to specify a large (L) or a small (S) table. Add $35 to the price of any large table. Save the program as **FurnitureSizes.java** in the Chapter.05 folder on your Student Disk.

3. Write a program for a college's admissions office. Create variables to store a student's numeric high school grade point average (for example, 3.2) and an admission test score. Print the message "Accept" if the student has any of the following:

   - A grade point average of 3.0 or above and an admission test score of at least 60
   - A grade point average below 3.0 and an admission test score of at least 80

   If the student does not meet either of the qualification criteria, print "Reject." Save the program as **Admission.java** in the Chapter.05 folder on your Student Disk.

4. Write a program that stores an hourly pay rate and hours worked. Compute gross pay (hours times rate), withholding tax, and net pay (gross pay minus withholding tax). Withholding tax is computed as a percentage of gross pay based on the following:

Gross Pay	Withholding Percentage
Up to and including 300.00	10
300.01 and up	12

   Save the program as **Payroll.java** in the Chapter.05 folder on your Student Disk.

5. a. Write a program that stores two integers and allows the user to enter a character. If the character is A, add the two integers. If it is S, subtract the second integer from the first; if it is M, multiply the integers. Display the results of the arithmetic. Save the program as **Calculate.java** in the Chapter.05 folder on your Student Disk.

   b. Modify the Calculate program so the user also can enter a D for divide. If the second number is zero, then display an error message; otherwise divide the first number by the second and display the results. Save the program as **Calculate2.java** in the Chapter.05 folder on your Student Disk.

6. a. Write a program for a lawn-mowing service. The lawn-mowing season lasts 20 weeks. The weekly fee for mowing a lot under 400 square feet is $25. The fee for a lot 400 square feet or more but under 600 square feet is $35 per week. The fee for a lot 600 square feet or over is $50 per week. Store the values in

the length and width variables and then print the weekly mowing fee, as well as the seasonal fee. Save the program as **Lawn.java** in the Chapter.05 folder on your Student Disk.

b. To the Lawn program created in 6a, add a prompt that asks the user whether the customer wants to pay A) once, B) twice, or C) 20 times per year. If the user enters A for once, the fee for the season is simply the seasonal total. If the customer requests two payments, each payment is half the seasonal fee plus a $5 service charge. If the user requests 20 separate payments, add a $3 service charge per week. Print the payment amount. Save the program in the Chapter.05 folder on your Student Disk as **Lawn2**.

7. a. Write a program that compares your checking account balance with your savings account balance (two doubles). Assign values to both variables, compare them, and then display either "Checking is higher" or "Checking is not higher". Save the program as **Balance.java** in the Chapter.05 folder on your Student Disk.

   b. Write a program so that it compares your checking account balance and your savings account balance to less than zero. If both statements are true, then display the message "Both accounts in the red". If the first balance is less than the second balance, and the first balance is greater than or equal to zero, then display the message "Both accounts in the black". Save the program as **Balance2.java** in the Chapter.05 folder on your Student Disk.

8. Write a program that asks a user to input an initial. Display the full name of an employee who matches the initial: A is Armando, B is Bruno, and Z is Zachary. All other entries should cause a "No such employee" message to display. Save the program as **PickEmployee.java** in the Chapter.05 folder on your Student Disk.

9. Write a program that asks the user to type a digit from the keyboard. If the character entered is not a digit, display an error message. Save the program as **GetDigit.java** in the Chapter.05 folder on your Student Disk.

10. Write a program that stores an IQ score. If the score is a number less than 0 or greater than 200, issue an error message; otherwise, issue an "above average", "average", or "below average" message for scores over, at, or under 100, respectively. Save the program as **IQ.java** in the Chapter.05 folder on your Student Disk.

11. Write a program for a college's admissions office. Create variables that store a numeric high school grade point average (for example, 3.2) and an admission test score. Print the message "Accept" if the student has any of the following:

    - A grade point average of 3.6 or above and an admission test score of at least 60
    - A grade point average of 3.0 or above and an admission test score of at least 70
    - A grade point average of 2.6 or above and an admission test score of at least 80
    - A grade point average of 2.0 or above and an admission test score of at least 90

    If the student does not meet any of the qualifications, print "Reject." Save the program as **Admission2.java** in the Chapter.05 folder on your Student Disk.

12. Write a program that stores an employee's hourly pay rate and hours worked. Compute gross pay (hours times rate), withholding tax, and net pay (gross pay minus withholding tax). Withholding tax is computed as a percentage of gross pay based on the following:

Gross Pay	Withholding Percentage
0 to 300.00	10
300.01 to 400.00	12
400.01 to 500.00	15
500.01 and over	20

Save the program as **Payroll2.java** in the Chapter.05 folder on your Student Disk.

13. Write a program that recommends a pet for a user based on the user's lifestyle. Prompt the user to enter whether he or she lives in an apartment, house, or dormitory (A, H, or D) and the number of hours the user is home during the average day. The user will select an hour category from a menu: A) 18 or more; B) 10 to 17; C) 8 to 9; D) 6 to 7; or E) 0 to 5. Print your recommendation based on the following:

Residence	Hours Home	Recommendation
House	18 or more	Pot bellied pig
House	10 through 17	Dog
House	Fewer than 10	Snake
Apartment	10 or more	Cat
Apartment	Fewer than 10	Hamster
Dormitory	6 or more	Fish
Dormitory	Fewer than 6	Ant farm

Save the program as **PetAdvice.java** in the Chapter.05 folder on your Student Disk.

14. Write a program that declares two ints named myNumberOfSiblings and yourNumberOfSiblings. Display an appropriate message to indicate whether your friend has more, fewer, or the same number of siblings as you. Display the number of siblings whether the `if` statement is `true` or not. Save the program as **Siblings.java** in the Chapter.05 folder on your Student Disk.

15. Write a program that compares the number of college credits you have earned with the number of college credits earned by a classmate or friend. Display an appropriate message to indicate whether your classmate has earned more, fewer, or the same number of credits as you. Display the number of college credits whether the `if` statement is `true` or not. Save the program as **Credits.java** in the Chapter.05 folder on your Student Disk.

16. Write a program that displays a menu of three items in a store, with a price for each item. Include characters a, b, and c so the user can select a menu item. Prompt the user to choose an item using the character that corresponds to the item. After the user makes the first selection, show a prompt to ask if another selection will be made. The user should respond Y or N to this prompt (for yes or no). If the user types N,

display the cost of the item. If the user types Y, allow the user to select another item and then display the total cost of the two items. Use the `switch` statement to check the menu selection. Save the program as **Store.java** in the Chapter.05 folder on your Student Disk.

17. Write a program using input, message, and dialog boxes to accept users' first and last names. Display the first and last names as they are entered. Prompt the user to verify that the names entered are correct. Use an `if...else` structure with a confirm dialog box to determine if the user clicks the No or Cancel options. Print appropriate messages if the user clicks No or cancel. Exit the program. Save the program as **DemoDialog.java** in the Chapter.05 folder on your Student Disk.
18. Write a program using the input dialog box that asks the question. "What is your zip code?" and displays "Enter Your Zip Code" in the title bar. The type of the dialog box should be QUESTION_MESSAGE. The program should display the zip code entered. Save the program as **ZipDialog.java** in the Chapter.05 folder on your Student Disk.
19. Write a program using the confirm dialog box that asks "Error reading file. Do you want to try again?". The confirm dialog box title should read "File input error". The user should see an error message icon and have a Yes or No option. Save the program as **ErrorDialog.java** in the Chapter.05 folder on your Student Disk.
20. Write a message dialog box that displays the message, "This program has finished installing." The message dialog box title should read "Installing Program". Save the program in the Chapter.05 folder on your Student Disk as **InstallDialog**.
21. Each of the following files in the Chapter.05 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, save DebugFive1.java as FixDebugFive1.java.
    a. DebugFive1.java
    b. DebugFive2.java
    c. DebugFive3.java
    d. DebugFive4.java

Case Project

Widget Company runs a small factory that makes several types of nuts and bolts. They employ factory workers that are paid one of three hourly rates depending on skill level: (a) 7.00, (b) 10.00, or (c) 12.00. Each factory worker can work: (a) 40 hours, (b) 45 hours, and (c) 50 hours per week. All hours over 40 are paid at double time.

Jack Smith, the factory manager, wants you to write an interactive Java payroll program that will calculate the gross pay for a factory worker. Hours worked and hourly pay rate are to be entered from the keyboard. Once the figures are entered for an employee, the program prints out: (1) the hours worked, (2) the hourly pay rate, (3) the regular pay for 40 hours, and (4) the overtime pay. The class name is pay.java. Save the program as **Pay.java** in the Chapter.05 folder on your Student Disk.