CHAPTER

6 LOOPING

In this chapter, you will:

- Learn about the loop structure
- Use a `while` loop
- Use shortcut arithmetic operators
- Use a `for` loop
- Learn how and when to use a `do...while` loop
- Learn about nested loops

Still having problems?" asks Lynn Greenbrier, who is watching intently as you code a new program.

"My programs can finally respond to questions and make choices," you reply. "But I still feel like something is missing."

"It's probably because you want to do the same task more than once without writing the procedure more than one time," Lynn suggests. "Let me introduce you to the control structure called looping."

Previewing the EvenInt Program

To preview the EvenInt program:

1. In your text editor, open the **Chap6EvenInt.java** file from the Chapter.06 folder on your Student Disk and examine the code. This file contains a definition for a class that loops through all integers from 1 to 100, and during looping prints each integer and all the integers by which it can be evenly divided. You will create a similar class file in this chapter.
2. At the command line, compile the **Chap6EvenInt.java** file using the command **`javac Chap6EvenInt.java`**.
3. Execute the program by typing the command **`java Chap6EvenInt`**. Press **[Enter]** repeatedly to see the evenly divisible numbers. They are displayed twenty at a time.

Learning About the Loop Structure

If making decisions is what makes programs seem smart, then looping is what makes programs seem powerful. A **loop** is a structure that allows repeated execution of a block of statements. Within a looping structure, a Boolean expression is evaluated. If it is `true`, then a block of statements, called the **loop body**, executes and the Boolean expression is evaluated again. As long as the expression is `true`, the statements in the loop body continue to execute. When the Boolean evaluation is `false`, the loop ends. Figure 6-1 shows a diagram of the logic of a loop.

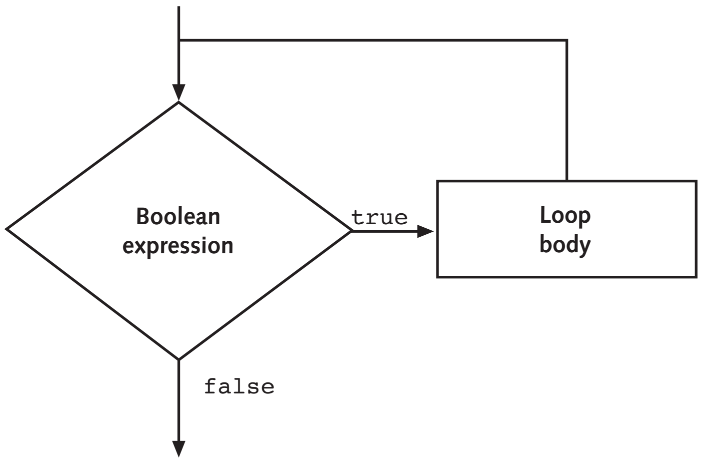


Figure 6-1 Logic of a loop

One execution of any loop is called an iteration.

USING A `while` LOOP

You can use a **`while` loop** to execute a body of statements continually as long as the Boolean expression continues to be `true`. A `while` loop consists of the keyword `while` followed by a Boolean expression within parentheses, followed by the body of the loop, which can be a single statement or a block of statements surrounded by curly braces.

You can use a `while` loop when you need to perform a task a predetermined number of times. A loop that executes a specific number of times is a **definite loop** or a counted loop. To write a definite loop, you initialize a loop control variable, and while the loop control variable does not pass a limit, you continue to execute the body of the `while` statement. You must include in the body of the `while` loop a statement that alters the loop control variable. For example, the program segment shown in Figure 6-2 prints the series of integers 1 through 10. The variable val starts the loop holding a value of 1, and while the value remains under 11, the val continues to print and be incremented.

```
int val = 1;
while (val < 11)
{
  System.out.println(val);
  ++val;
}
```

Figure 6-2 Printing the integers 1 through 10 with a `while` loop

The code shown in Figure 6-3 causes the message "Hello" to display (theoretically) forever because there is no code to end the loop. A loop that never ends is called an **infinite loop**.

An infinite loop might not actually execute infinitely. Eventually the computer memory will be exhausted (literally and figuratively) and execution will stop. Also, it's possible that the processor has a time-out feature. Either way, and depending on your system, quite a bit of time could pass before the loop stops running.

```
while (4 > 2)
  System.out.println("Hello");
```

Figure 6-3 An infinite loop

In Figure 6-3, the expression `4 > 2` evaluates to `true`. You obviously never need to make such an evaluation, but if you do so in this `while` loop, the body of the loop is entered and "Hello" displays. Next, the expression is evaluated again. The expression `4 > 2` is still `true`, so the body is entered again. "Hello" displays repeatedly; the loop never finishes because `4 > 2` is never `false`.

It is a bad idea to intentionally write an infinite loop. However, even experienced programmers write them by accident. So, before you start writing any loops, it is good to know how to break out of an infinite loop in case you find yourself in the midst of one. If you think your program is in a loop, you can press and hold [Ctrl], and then press C or [Break].

To prevent a `while` loop from executing infinitely, four separate actions must occur:

1. A named loop control variable is initialized to a starting value.
2. The loop control variable is tested in the `while` statement.
3. If the test expression is `true`, the body of the `while` statement must take some action that alters the value of the loop control variable.
4. The action of the body statement(s) must eventually change the value of the control variable so that the test of the `while` statement evaluates to `false`.

All of these conditions are met by the example in Figure 6-4. First, a loop control variable loopCount is named and set to a value of 1. Second, the statement `while(loopCount < 3)` is tested. Third, the loop body is executed because the loopControl variable loopCount is less than 3. Note that the loop body shown in Figure 6-4 consists of two statements made into a block by their surrounding curly braces. The first statement prints "Hello," and then the second statement adds one to loopCount. The next time loopCount is evaluated, it is 2. It is still less than 3, so the loop body executes again. "Hello" prints a second time, and loopCount becomes 3. Fourth, because the expression `loopCount < 3` now evaluates to `false`, the loop ends. Program execution then continues with any subsequent statements.

Tip

To an algebra student, a statement such as `loopCount = loopCount + 1` looks wrong—a value can never be one more than itself. In programming, however, `loopCount = loopCount + 1;` takes the value of loopCount, adds one to it, and then assigns the new value back into loopCount.

```
loopCount = 1;
while (loopCount < 3)
{
  System.out.println("Hello");
  loopCount = loopCount + 1;
}
```

Figure 6-4 A simple loop that executes twice

If the curly braces are omitted from the code shown in Figure 6-4, the `while` loop stops at the end of the "Hello" statement. Adding one to the loopCount is no longer part of a block that contains the loop, so an infinite situation is created.

Also, if a semicolon is mistakenly placed at the end of the partial statement `while (loopCount < 3);`, the loop is also infinite. This loop has an **empty body**, or a body with no statements in it, so the Boolean expression is evaluated, and because it is `true`, the loop body is entered. Because the loop body is empty, no action is taken, and the Boolean expression is evaluated again. Nothing has changed, so it is still `true`, the empty body is entered, and the infinite loop continues.

It is very common to alter the value of a loop control variable by adding one to it, or **incrementing** the variable. However, not all loops are controlled by adding one. The loop shown in Figure 6-5 prints "Hello" twice, just as the loop in Figure 6-4 does, but its loop is controlled by subtracting 1 from a loop control variable, or **decrementing** it.

```
loopCount = 10;
while (loopCount > 8)
{
  System.out.println("Hello");
  loopCount = loopCount - 1;
}
```

Figure 6-5 Another simple loop that executes twice

In the program segment shown in Figure 6-5, the variable loopCount begins with a value of 10. The loopCount is greater than 8, so the loop body prints "Hello" and decrements loopCount so it becomes 9. The Boolean expression in the `while` loop is tested again. Because 9 is more than 8, "Hello" prints again and loopCount becomes 8. Now loopCount is not greater than 8, so the loop ends.

The possibilities are endless. Figure 6-6 shows the loopCount being increased by 5, and the results are still the same—the loop prints "Hello" twice. In general, you should not use such unusual methods because they simply make a program confusing. The clearest and best method is to start loopCount at 0 or 1, and continue while it is less than 2 or 3, incrementing by one each time through the loop.

```
loopCount = 30;
while (loopCount < 40)
{
  System.out.println("Hello");
  loopCount = loopCount + 5;
}
```

Figure 6-6 A third simple loop that executes twice

Within a loop, you are not required to alter the loop control variable by adding to it or subtracting from it. When you write a loop that is controlled by an arithmetic result, you need to know how many times you want a loop to execute. Often, the value of a loop control variable is not altered by arithmetic, but instead is altered by user input. For example, perhaps you want to continue performing some task as long as the user indicates a desire to continue. In that case, when you write the program you do not know whether the loop will be executed two times, 200 times, or at all.

Tip

Unlike a loop that you program to execute a fixed number of times, a loop controlled by the user is a type of indefinite loop because you don't know how many times it will eventually loop.

Consider a simple program in which you display a bank balance and ask the user whether he or she wants to see what the balance will be after interest has accumulated for each year. Each time the user indicates a desire to continue, an increased balance appears, reflecting one more year of accumulated interest. When the user finally indicates a desire to exit, the program ends. The program appears in Figure 6-7.

```
public class LoopingBankBal
{
  public static void main(String[] args) throws Exception
  {
    double bankBal = 1000;
    double intRate = 0.04;
    char response;
    System.out.println
      ("Do you want to see your balance? Y or N");
    response = (char)System.in.read();
    System.in.read(); System.in.read();
        // Absorbs Enter key
    while (response == 'Y')
    {
      System.out.println("Bank balance is " + bankBal);
      bankBal = bankBal + bankBal * intRate;
      System.out.println
        ("Do you want to see next year's balance? Y or N");
      response = (char)System.in.read();
      System.in.read(); System.in.read();
        // Absorbs Enter key
    }
    System.out.println("Have a nice day!");
  }
}
```

Figure 6-7 LoopingBankBal program

The program shown in Figure 6-7 continues to display bank balances while the response is Y. It could also be written to display while the response is not N, as in `while (response != 'N')....` A value that a user must supply to stop a loop is called a **sentinel value**.

The program shown in Figure 6-7 contains three variables: a bank balance, an interest rate, and a response. The program asks the user, "Do you want to see your balance?" and reads the response. Recall that the second and third read() statements are required to accept [Enter] that is typed after the Y or N entry. The loop in the program begins with `while(response == 'Y')`. If the user types any response other than Y, then the loop body never executes; instead, the next statement to execute is the "Have a nice day!" statement at the bottom of the program. However, if the user enters Y, then all five statements within the loop body will execute. The current balance will display, and then the program increases the balance by the interest rate value. The program then prompts the user to type Y or N, and two characters are entered—the response and [Enter]. The loop ends with a closing curly brace, and program control returns to the top of the loop, where the Boolean expression in the `while` loop is tested again. If the user typed Y, then the loop is entered and the increased bankBal value that was calculated is finally displayed.

Next you will improve the ChooseManager4 program so the user cannot make an invalid choice for the type of event.

To improve the ChooseManager4 program:

1. If necessary, start your text editor and then open the **ChooseManager4.java** text file from the Chapter.06 folder on your Student Disk.
2. Position the insertion point to the right of the statement that reads in the character for the event type, press **[Enter]** to start a new line, and then type the beginning of the `while` loop that will continue to execute while the user's entry is not one of the three allowed event types:

```
while (eventType != 'C' && eventType != 'P'
&& eventType != 'N')
```

3. On a new line, type the opening curly brace for the `while` loop, press **[Enter]**, and then type the two statements that will accept the [Enter] key carriage return and line feed remaining from the user's data entry:

```
System.in.read(); System.in.read();
```

4. On a new line, display the following message so the user knows the data entry was invalid:

```
System.out.println("Entry must be C or P or N!");
```

5. On another new line, read in the eventType value again by typing:

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

6. On a new line, type the closing curly brace for the `while` loop. After making it through the `while` loop you just added, the program is guaranteed that the eventType is C, P, or N, so you can make the nonprofit event the default case.
7. Change the statement `case 'N':` to **`default:`**, and then delete the three lines that represent the current default case—the existing `default` keyword and the `System.out.printIn("Invalid Entry");` statement that follow it.
8. Save, compile, and test the program. No matter how many invalid entries you make, the program will continue to prompt you until you enter C, P, or N.

Using Shortcut Arithmetic Operators

It is common to increase the value of a variable in a program. As you saw in the last section, many loops are controlled by continually adding one to some variable, or incrementing, as in `count = count + 1;`. Similarly, in the looping bank balance program, the program increased a bank balance by an interest amount with the statement `bankBal = bankBal + bankBal * intRate;`. In other words, the bank balance became its old value *plus* a new interest amount; this process is known as **accumulating**.

Because increasing a variable is so common, Java provides you with several shortcuts for incrementing and accumulating. The statement `count += 1;` is identical in meaning to `count = count + 1`. The `+=` adds and assigns in one operation. Similarly, `bankBal += bankBal * intRate;` increases a bankBal by a calculated interest amount.

When you want to increase a variable's value by exactly one, you can use two other shortcut operators—the **prefix ++** and the **postfix ++**. To use a prefix ++, you type two plus signs before the variable name. The statement `someValue = 6;`followed by `++someValue;` results in someValue holding 7—one more than it held before you applied the ++. To use a postfix ++, you type two plus signs just after a variable name. The statements `anotherValue = 56; anotherValue++;` result in anotherValue containing 57.

Tip

You can use the prefix ++ and postfix ++ with variables, but not with constants. An expression such as `++84;` is illegal because an 84 must always remain an 84. However, you can create a variable as `int val = 84` and then write `++val;` or `val++;` to increase the variable's value.

The prefix and postfix increment operators are unary operators because you use them with one value. Most arithmetic operators, such as those used for addition and multiplication, are **binary** operators—they operate on two values.

When you simply want to increase a variable's value by one, there is no difference between using the prefix and postfix increment operators. Each operator results in increasing the variable by one. However, these operators do function differently. When you use the prefix ++, the result is calculated and stored, and then the variable is used. For example, if `b = 4;` and `c = ++b;`, this results in both b and c holding the value 5. When you use the postfix ++, the variable is used and then the result is calculated and stored. For example, if `b = 4;` and then `c = b++;`, 4 will be assigned to c, and then after the assignment, b is increased and takes the value 5. In other words, if `b = 4`, then the value of `b++` is also 4, but after the statement is completed, the value of b will be 5. If `d = 8` and `e = 8`, both `++d == 9` and `e++ == 8` are `true` expressions.

Similar logic can be applied when you use the prefix and the postfix decrement operators. For example, if `b = 4;` and `c = b–;`, 4 will be assigned to c, and then after the assignment, b is decreased and takes the value 3. If `b = 4;` and `c = –b;`, b will be decreased to 3 and 3 will be assigned to c.

Next you will add a program that demonstrates how prefix and postfix operators are used in incrementation, and how incrementing affects the expressions that contain these operators.

To demonstrate the effect of the prefix and postfix increment operators:

1. Open a new text file and begin a demonstration class named DemoIncrement by typing:

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

2. On a new line, add a variable v and assign it a value of 4. Then declare a variable named plusPlusV and assign it a value of `++v` by typing:

```
int v = 4;
int plusPlusV = ++v;
```

The last statement, `int plusPlusV = ++v;`, will increase v to 5, so before declaring a vPlusPlus variable to which you assign `v++`, reset v to 4 by typing:

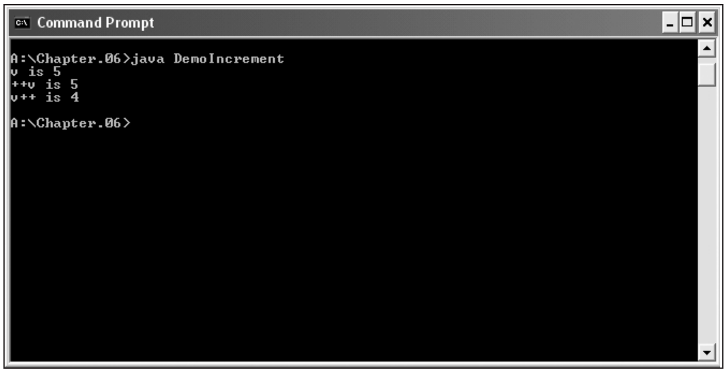
```
v = 4;
int vPlusPlus = v++;
```

4. Add the following statements to print the three values:

```
System.out.println("v is " + v);
System.out.println("++v is " + plusPlusV);
System.out.println("v++ is " + vPlusPlus);
```

5. Add the closing curly brace for the main() method and the closing curly brace for the DemoIncrement class.

6. Save the file as **DemoIncrement.java** in the Chapter.06 folder on your Student Disk. Compile and execute the program. Your output should look like Figure 6-8.



```
A:\Chapter.06>java DemoIncrement
v is 5
++v is 5
v++ is 4

A:\Chapter.06>
```

Figure 6-8 Output of the DemoIncrement program

To illustrate how comparisons are made, add a few more variables to the DemoIncrement program.

7. Position the insertion point to the right of the last println() statement, and then press **[Enter]** to start a new line.
8. Add three new integer variables and two new Boolean variables. The first Boolean variable compares `++w` to y; the second Boolean variable compares `x++` to y:

   ```
   int w = 17, x = 17, y = 18;
   boolean compare1 = (++w == y);
   boolean compare2 = (x++ == y);
   ```

9. Add the following statements to display the values stored in the compare variables:

   ```
   System.out.println("First compare is " + compare1);
   System.out.println("Second compare is " + compare2);
   ```

10. Save, run, and compile the program.

Besides using the shortcut operator `+=`, you can use `-=`, `*=`, and `/=`. Each of these operators is used to perform the operation and assign the result in one step. For example, `balanceDue -= payment` subtracts payment from balanceDue and assigns the result to balanceDue.

Using a for Loop

A **for loop** is a special loop that is used when a definite number of loop iterations is required. Recall that a while loop, also a definite loop type, has a variable number of iterations and may not execute the body of its loop at all if the initial test condition is false. The for loop is used when one or more loop iterations are known to be needed. You will need definite loops frequently when you write programs, and the for loop provides you with a shorthand notation that you can use to create those definite loops. When you use a for loop, you can indicate the starting value for the loop control variable, the test condition that controls loop entry, and the expression that alters the loop control variable all in one convenient place.

You begin a for loop with the keyword for followed by a set of parentheses. Within the parentheses there are three sections separated by exactly two semicolons. The three sections are usually used for the following:

- Initializing the loop control variable
- Testing the loop control variable
- Updating the loop control variable

The body of the for statement follows the parentheses. As with an if statement or a while loop, you can use a single statement as the body of a for loop, or you can use a block of statements enclosed in curly braces. The for statement shown in Figure 6-9 produces the same output as the while statement shown previously in Figure 6-2—it prints the integers 1 through 10.

```
for(int val = 1; val < 11; ++val)
{
  System.out.println(val);
}
```

Figure 6-9 Printing the integers 1 through 10 with a for loop

Tip

You did not have to declare the variable val within the for statement. If you declared val earlier in the program block as int val; before the for statement begins, then the for statement would be for(val = 1; val < 11; ++val). In other words, the for statement does not need to declare a variable; it can simply give a starting value to a previously declared variable. Generally, it is not considered to be good practice to reuse variables in for statements.

Within the parentheses of the for statement shown in Figure 6-9, the first section prior to the first semicolon declares a variable named val and initializes it to 1. The program will execute this statement once, no matter how many times the body of the for loop executes.

After the initialization expression executes, program control passes to the middle, or test section, of the `for` statement. If the Boolean expression found there evaluates to `true`, then the body of the `for` loop is entered. In the program segment shown in Figure 6-9, val is set to 1, so when `val < 11` is tested, it evaluates to `true`. The loop body prints the val.

After the loop body executes, the final one-third of the `for` loop executes, and val is increased to 2. Following the third section, program control returns to the second section, where val is compared to 11 a second time. Because val is still less than 11, the body executes: val (now 2) prints, and then the third, altering portion of the `for` loop executes again. The variable val increases to 3, and the `for` loop continues.

Eventually, when val is not less than 11 (after 1 through 10 have printed), the `for` loop ends, and the program continues with any statements that follow the `for` loop.

Although the three sections of the `for` loop are most commonly used for initializing, testing, and incrementing, you can also perform the following tasks:

- Initialization of more than one variable by placing commas between the separate statements, as in `for(g = 0, h = 1; g < 6; ++g)`
- Performance of more than one test, as in `for(g = 0; g < 3 && h > 1; ++g)`
- Decrementation or performance of some other task, as in `for(g = 5; g >= 1; —g)`
- Leaving one or more portions of the `for` loop empty, although the two semicolons are still required as placeholders

Usually you should use the `for` loop for its intended purpose—a shorthand way of programming a definite loop. Occasionally, you will encounter a `for` loop that contains no body, such as `for(x = 0; x < 100000; ++x);`. This kind of loop exists simply to use time—for instance, when a brief pause is desired during program execution.

Tip

Java also contains a built-in method to pause program execution. The sleep() method is part of the Thread class in the java.lang package.

Learning How and When to Use a `do...while` Loop

With all the loops you have written so far, the loop body might execute many times, but it is also possible that the loop will not execute at all. For example, recall the bank balance program that displays compound interest, part of which is shown in Figure 6-10.

The program segment begins with the user prompt, "Do you want to see your balance? Y or N". If the user does not type Y, the loop body never executes. The while loop checks a value at the "top" of the loop before the body has a chance to execute. Sometimes you might need a loop body to execute at least one time. If so, you want to write a loop that checks at the "bottom" of the loop after the first iteration. The **do...while loop** checks the bottom of the loop after one repetition has occurred.

```
System.out.println("Do you want to see your balance? Y or N");
response = (char)System.in.read();
System.in.read(); System.in.read(); // Absorbs Enter key
while (response == 'Y')
{
  System.out.println("Bank balance is " + bankBal);
  bankBal = bankBal + bankBal * intRate;
  System.out.println
    ("Do you want to see next year's balance? Y or N");
  response = (char)System.in.read();
  System.in.read();  // Absorbs Enter key
}
```

Figure 6-10 Part of the bank balance program with a while loop

Figure 6-11 shows a do...while loop for the bank balance program. The loop starts with the keyword do. The body of the loop follows and is contained within curly braces. The bankBal variable is output before the user has any option of responding. At the end of the loop, the user is prompted, "Do you want to see next year's balance? Y or N". Now the user has the option of seeing more balances, but the first prompt was unavoidable. The user's response is checked at the bottom of the loop. If it is Y, then the loop repeats.

```
do
{
  System.out.println("Bank balance is " + bankBal);
  bankBal = bankBal + bankBal * intRate;
  System.out.println
    ("Do you want to see next year's balance? Y or N");
  response = (char)System.in.read();
  System.in.read(); System.in.read(); // Absorbs Enter key
} while (response == 'Y');
```

Figure 6-11 Part of the bank balance program with a do...while loop

In any situation where you want to loop, you are never required to use a do...while loop. Within the bank balance example, you could simply unconditionally display the bank balance once, prompt the user, and then start a while loop that might not be entered. However, when you know you want to perform some task at least one time, the do...while loop is convenient.

LEARNING ABOUT NESTED LOOPS

Just as `if` statements can be nested, so can loops. You can place a `while` loop within a `while` loop, a `for` loop within a `for` loop, a `while` loop within a `for` loop, or use any combination you can think of.

For example, suppose you want to find all the numbers that divide evenly into 100. You can write a `for` loop that sets a variable to 1 and increments it to 100. Each of the 99 times through the loop, if 100 is evenly divisible by the number (that is, if 100%num is equivalent to 0), then the program prints the number. Next you will write a program that determines all the integers that divide evenly into 100.

Tip

To find all the numbers that divide evenly into 100, you actually have to test divisors only through 50. You cannot evenly divide any number by a number that is more than half of the original number.

To write a program that finds the values that divide evenly into 100:

1. Open a new text file.
2. Begin the program named EvenInt by typing the following code to declare an integer variable named num:

```
public class EvenInt
{
 public static void main(String[] args)
 {
  int num;
```

3. Type a statement that explains the purpose of the program:

```
System.out.print("100 is evenly divisible by ");
```

4. Write the `for` loop that varies num from 1 to 99. With each iteration of the loop, test whether 100%num is 0. If you divide 100 by a number and there is no remainder, then the number goes into 100 evenly.

```
for(num = 1; num < 100; ++num)
 if(100%num == 0)
  System.out.print(num + "  ");
  // Print the number and two spaces
```

5. Add an empty println() statement to advance the insertion point to the next line by typing **`System.out.println();`**.
6. Type the closing curly braces for the main() method and the EvenInt class.
7. Save the program as **EvenInt.java** in the Chapter.06 folder on your Student Disk. Compile and run the program. The program prints 100 is evenly divisible by 1 2 4 5 10 20 25 50.

What if you want to know what number goes evenly into 100, but also what every number up to 100 can be evenly divided by? You can write 99 more loops, or you can place the current loop inside a different, outer loop, as you will do next.

When you use a loop within a loop, you should always think of the outer loop as the all-encompassing loop. When you describe the task at hand, you often use the word "each" when referring to the inner loop. For example, if you want to print three mailing labels each for 20 customers, the label variable would control the inner loop:

```
for(customer = 1; customer <= 20; ++customer)
for(label = 1; label <=3; ++label)
 printLabelMethod();
```

If you want to print divisors for each number from 1 to 100, then the loop that varies the number to be divided is the outside loop. You need to perform 100 mathematical calculations on each number, so that constitutes the "smaller" or inside loop.

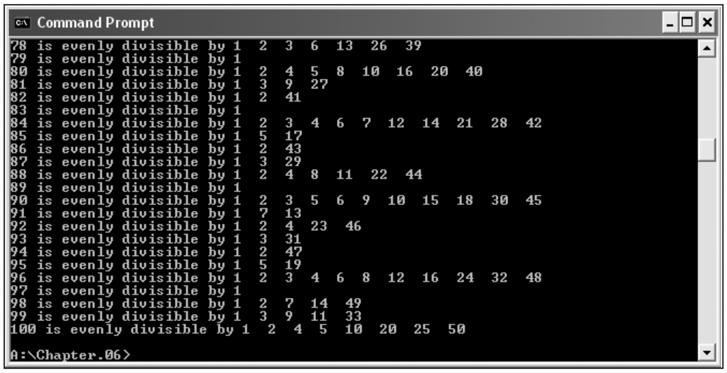
To create a nested loop to print even divisors for every number up to 100:

1. Open the file **EvenInt.java** in your text editor, and then save the class as **`EventInt2`**. Create an outer loop that uses the variable testNum to test every number from 1 to 100. Position the insertion point after the declaration of num but before the semicolon that ends the declaration, and then type a comma and **`testNum`**.
2. Position the insertion point to the right of the line with the variable declarations, press **[Enter]** to start a new line, and then type the other `for` loop:

   `for(testNum = 1; testNum <= 100; ++testNum)`
3. Press **[Enter]**, and then type the opening curly brace for this loop on the next line.
4. Change the statement that prints "100 is evenly divisible by " to the following:

   `System.out.print(testNum + " is evenly divisible by ");`
5. Change the `for` statement that varies num from 1 to 100 so it only varies num from 1 to testNum. For example, during each iteration, if testNum is 46, you want to divide it only by numbers that are 45 or less. Type the following code to make this change:

   `for(num = 1; num < testNum; ++num)`
6. Change the statement that tests 100%num to **`if(testNum%num == 0)`**.
7. Following the empty println() statement, add the closing curly brace for the outer `for` loop.
8. Save the file as **EventInt2**, compile, and run the program. The output will scroll on the screen. When it stops, it should look similar to Figure 6-12.



```
78 is evenly divisible by 1  2  3  6  13  26  39
79 is evenly divisible by 1
80 is evenly divisible by 1  2  4  5  8  10  16  20  40
81 is evenly divisible by 1  3  9  27
82 is evenly divisible by 1  2  41
83 is evenly divisible by 1
84 is evenly divisible by 1  2  3  4  6  7  12  14  21  28  42
85 is evenly divisible by 1  5  17
86 is evenly divisible by 1  2  43
87 is evenly divisible by 1  3  29
88 is evenly divisible by 1  2  4  8  11  22  44
89 is evenly divisible by 1
90 is evenly divisible by 1  2  3  5  6  9  10  15  18  30  45
91 is evenly divisible by 1  7  13
92 is evenly divisible by 1  2  4  23  46
93 is evenly divisible by 1  3  31
94 is evenly divisible by 1  2  47
95 is evenly divisible by 1  5  19
96 is evenly divisible by 1  2  3  4  6  8  12  16  24  32  48
97 is evenly divisible by 1
98 is evenly divisible by 1  2  7  14  49
99 is evenly divisible by 1  3  9  11  33
100 is evenly divisible by 1  2  4  5  10  20  25  50

A:\Chapter.06>
```

Depending on your monitor resolution, you may see more lines than are displayed here

Figure 6-12 Output of the EvenInt2 program

When the program executes, 100 lines of output display on the screen. But, as Figure 6-12 shows, the first 76 (or so) lines scroll so rapidly that you can't read them. It would help if you could stop the output after every 20 lines or so; then you would have time to read the messages. Next you will use the modulus operator for this task. If you stop output when testNum is 20, 40, 60, and 80, then you can test testNum to see if it is evenly divisible by 20. When it is, you can pause program execution by asking the user to press [Enter] and accept keyboard input.

To pause your program after every 20 lines of output:

1. At the end of the EvenInt file and just prior to the closing brace for the `for` loop, type the following code to test testNum to determine if 20 divides into it evenly, then tell the user to press [Enter] and accept an [Enter] key from the keyboard (you don't have to store the entered key in a variable):

```
if(testNum % 20 == 0)
{
 System.out.println("Press Enter to continue");
 System.in.read(); System.in.read();
}
```

2. Because the program now uses the System.in.read() method, you must position the insertion point at the end of the main() method header line and add **`throws Exception`**.

3. Save, compile, and test the program. It will pause after every 20 lines of output and wait until you press [Enter] before continuing until the program ends.

Chapter Summary

- A loop is a structure that allows repeated execution of a block of statements. A loop that never ends is called an infinite loop. A loop that executes a specific number of times is a definite loop or counted loop. You can nest loops.

- Within a looping structure, a Boolean expression is evaluated, and if it is `true`, a block of statements called the loop body executes; then the Boolean expression is evaluated again.
- You can use a `while` loop to execute a body of statements continually `while` some condition continues to be `true`.
- To execute a `while` loop, you initialize a loop control variable, test it in a `while` statement, and then alter the loop control variable in the body of the `while` structure.
- The `+=` operator adds and assigns in one operation.
- The prefix ++ and the postfix ++ increase a variable's value by one. The prefix -- and postfix -- decrement operators reduce a variable's value by one. When you use the prefix ++, the result is calculated and stored, and then the variable is used. When you use the postfix ++, the variable is used, and then the result is calculated and stored.
- Unary operators are used with one value. Most arithmetic operators are binary operators that operate on two values.
- The shortcut operators `+=`, `-=`, `*=`, and `/=` perform operations and assign the result in one step.
- A `for` loop initializes, tests, and increments in one statement. There are three sections within the parentheses of a `for` loop that are separated by exactly two semicolons.
- The `do...while` loop tests a Boolean expression after one repetition has taken place, at the bottom of the loop.

Review Questions

1. A structure that allows repeated execution of a block of statements is a(n) ________________.
   a. cycle
   b. loop
   c. ring
   d. iteration
2. A loop that never ends is a(n) ________________ loop.
   a. iterative
   b. infinite
   c. structured
   d. illegal

3. To construct a loop that works correctly, you should initialize a loop control ________________.
   a. variable
   b. constant
   c. structure
   d. condition
4. What is the output of the following code?
   ```
   b = 1;
   while (b < 4)
   System.out.println(b + " ");
   ```
   a. 1
   b. 1 2 3
   c. 1 2 3 4
   d. 1 1 1 1 1 1...
5. What is the output of the following code?
   ```
   b = 1;
   while (b < 4)
   {
    System.out.println(b + " ");
    b = b + 1;
   }
   ```
   a. 1
   b. 1 2 3
   c. 1 2 3 4
   d. 1 1 1 1 1...
6. What is the output of the following code?
   ```
   e = 1;
   while (e < 4);
   System.out.println(e + " ");
   ```
   a. 1
   b. 1 1 1 1 1 1...
   c. 1 2 3 4
   d. 4 4 4 4 4 4...

7. If `total = 100` and `amt = 200`, then after the statement `total += amt`, ________________.
   a. total is equal to 200
   b. total is equal to 300
   c. amt is equal to 100
   d. amt is equal to 300
8. The modulus operator `%` is a ________________ operator.
   a. unary
   b. binary
   c. tertiary
   d. postfix

9. The prefix ++ is a ________________ operator.
   a. unary
   b. binary
   c. tertiary
   d. postfix
10. If `g = 5`, then the value of the expression `++g` is ________________.
    a. 4
    b. 5
    c. 6
    d. 7
11. If `h = 9`, then the value of the expression `h++` is ________________.
    a. 8
    b. 9
    c. 10
    d. 11
12. If `j = 5` and `k = 6`, then the value of `j++ == k` is ________________.
    a. 5
    b. 6
    c. `true`
    d. `false`

13. You must always include ________________ in a `for` loop's parentheses.
    a. two semicolons
    b. three semicolons
    c. two commas
    d. three commas
14. The statement `for(a = 0; a < 5; ++a) System.out.print(a + " ");` prints ________________.
    a. 0 0 0 0 0
    b. 0 1 2 3 4
    c. 0 1 2 3 4 5
    d. nothing
15. The statement `for(b = 1; b > 3; ++b) System.out.print(b + " ");` prints ________________.
    a. 1 1 1
    b. 1 2 3
    c. 1 2 3 4
    d. nothing
16. What does the following statement print?

```
for(f = 1, g = 4; f < g; ++f, —g)
 System.out.print(f + " " + g + " ");
```

    a. 1 4 2 5 3 6 4 7...
    b. 1 4 2 3 3 2
    c. 1 4 2 3
    d. nothing
17. The loop that performs its conditional check at the bottom of the loop is a ________________ loop.
    a. `while`
    b. `do...while`
    c. `for`
    d. `for...while`

18. What does this program segment print?

```
d = 0;
do
{
 System.out.print(d + " ");
 d++;
} while (d < 2);
```

   a. 0

   b. 0 1

   c. 0 1 2

   d. nothing

19. What does this program segment print?

```
for(f = 0; f < 3; ++f)
 for(g = 0; g < 2; ++g)
 System.out.print(f + " " + g + " ");
```

   a. 0 0 0 1 1 0 1 1 2 0 2 1

   b. 0 1 0 2 0 3 1 1 1 2 1 3

   c. 0 1 0 2 1 1 1 2

   d. 0 0 0 1 0 2 1 0 1 1 1 2 2 0 2 1 2 2

20. What does this program segment print?

```
for(m = 0; m < 4; ++m);
 for(n = 0; n < 2; ++n);
 System.out.print(m + " " + n + " ");
```

   a. 0 0 0 1 1 0 1 1 2 0 2 1 3 0 3 1

   b. 0 1 0 2 1 1 1 2 2 1 2 2

   c. 4 2

   d. 3 1

EXERCISES

1. Write a program that prints all even numbers from 2 to 100 inclusive. Save the program as **EvenNums.java** in the Chapter.06 folder on your Student Disk.
2. Write a program that asks a user to type A, B, C, or Q. When the user types Q, the program ends. When the user types A, B, or C, the program displays the message "Good job!" and then asks for another input. When the user types anything else, issue an error message and then ask for another input. Save the program as **ABCInput.java** in the Chapter.06 folder on your Student Disk.
3. Write a program that prints every integer value from 1 to 20 along with its squared value. Save the program as **TableOfSquares.java** in the Chapter.06 folder on your Student Disk.

4. Write a program that sums the integers from 1 to 50 (that is, 1 + 2 + 3... + 50). Save the program as **Sum50.java** in the Chapter.06 folder on your Student Disk.
5. Write a program that shows the sum of 1 to n for every n from 1 to 50. That is, the program prints 1, 3 (the sum of 1 and 2), 6 (the sum of 1, 2, and 3), and so on. Save the program as **EverySum.java** in the Chapter.06 folder on your Student Disk.
6. Write a program that prints every perfect number from 1 through 1000. A perfect number is one that equals the sum of all the numbers that divide evenly into it. For example, 6 is perfect because 1, 2, and 3 divide evenly into it, and their sum is 6. Save the program as **Perfect.java** in the Chapter.06 folder on your Student Disk.
7. Write a program that calculates the amount of money earned on an investment; that amount should include 12 percent interest. Prompt the user to choose the investment amount from one menu and the number of years for the investment from a second menu. Display the total amount (balance) for each year of the investment. Use a loop instruction to calculate the balance for each year. Use the formula amount = investment * (1 + interest) raised to a power equal to the year to calculate the balance. Use the Math power function from Chapter 4, Math.pow(x,y) where x = (1 + interest) and y is the year. Save the program as **Investment.java** in the Chapter.06 folder on your Student Disk.
8. Write a program that creates a quiz that contains questions about a hobby, popular music, astronomy, or any other personal interest. After the user selects a topic, display a series of questions. The user should answer the questions with one character for multiple choice, true/false, or yes/no. If the user responds to a question correctly, display an appropriate message. If the user responds to a question incorrectly, display an appropriate response and the correct answer. At the end of the quiz, display the number of correct answers. Save the program as **Quiz.java** file in the Chapter.06 folder on your Student Disk.
9. Write a program that displays a series of survey questions, with one-character answers. At the end of the survey, ask the user if he or she wants to enter another set of responses. If the user responds no, then display the results of the survey for each question. Enter several sets of responses to test the program. Save the program as **Survey.java** in the Chapter.06 folder on your Student Disk.
10. Write a program that displays the results of a coin toss. The user is prompted to enter a for heads and b for tails. Use the Math.random() function from Chapter 4 to generate a number between 0 and 1. Use the probabilities of .5 for either a head or tail. Make sure that the user enters an a or b by continuing to loop until a or b is entered. Print either "You win" or "You lose" as the output. Save the program as **FlipCoin.java** in the Chapter.06 folder on your Student Disk.
11. Write a program that will count and display the number of heads and tails for a coin flipped from 1 to 9 times. The user is prompted for an integer from 1 to 9. Use the Math.random() function from Chapter 4 to generate a number between 0 and 1. Use the probabilities of .5 for either a head or a tail. Make use of the fact that the ASCII code for the numerals 0 to 9 is decimal 48 through 57. Save the program as **CountFlips.java** in the Chapter.06 folder on your Student Disk.

12. Each of the following files in the Chapter.06 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 DebugSix1.java as FixDebugSix1.java.
    a. DebugSix1.java
    b. DebugSix2.java
    c. DebugSix3.java
    d. DebugSix4.java

Case Project

Local Caterer's Company operates a small Mom and Pop catering service. They want you to write an object-oriented program for them to schedule their catering events. They mostly cater special events, but they occasionally cater corporate, private and nonprofit events as well. Mom and Pop are both managers; Mom manages nonprofit and special events, and Pop manages the corporate and private events. All catered events have an event minimum rate shown in the table below:

Local Caterer's Company fee schedule

	Event Minimum Rate	Manager
Corporate	$500.00	Pop
Private	$300.00	Pop
Nonprofit	$150.00	Mom
Special	$200.00	Mom

Write a program for Local Caterer's Company that contains a class that has methods for the event minimum rate and event type. Write a test program that accepts keyboard input and checks the event type input for errors until a valid event type is entered. After a valid input type is entered, print the manager's name for that event, the type of event chosen, and the minimum rate to be charged.