CS3231
Object Oriented Programming I

Name: _________________________________ ( ) Date: ________________

Chapter 6: Loops

Lesson Objectives

After completing the lesson, the student will be able to

- *write programs for executing statements repeatedly using a while loop*
- *follow the loop design strategy to develop loops*
- *control a loop with a sentinel value*
- *obtain large input from a file using input redirection rather than typing from the keyboard*
- *write loops using do-while statements*
- *write loops using for statements*
- *discover the similarities and differences of three types of loop statements*
- *write nested loops*
- *use counter, sentinel and accumulator in loops*
- *implement program control with break and continue*

6.1 Loop

A loop can be used to tell a program to execute statements repeatedly. Suppose that you need to display a string (e.g., Welcome to Java!) a hundred times. It would be tedious to have to write the following statement a hundred times:

```
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
...
System.out.println("Welcome to Java!");
```

Java provides a powerful construct called **a loop that controls how many times an operation or a sequence of operations is performed in succession**. Using a loop statement, you simply tell the computer to display a string a hundred times without having to code the print statement a hundred times.

Loops are constructs that control repeated executions of a block of statements. The concept of looping is fundamental to programming. Java provides three types of loop statements:

- `while` loops
- `do-while` loops
- `for` loops.

.

6.2 `while` Loop

A `while` loop executes statements repeatedly while the condition is true. The syntax for the while loop is:

```
while (loop-continuation-condition) {
 // Loop body
 Statement(s);
}
```

Figure 6.1a shows the `while`-loop flowchart. The part of the loop that contains the statements to be repeated is called the **loop body**. **A one-time execution of a loop body is referred to as**

an iteration (or repetition) of the loop. Each loop contains a **loop-continuation-condition**, a Boolean expression that controls the execution of the body. It is evaluated each time to determine if the loop body is executed. If its evaluation is true, the loop body is executed; if its evaluation is false, the entire loop terminates and the program control turns to the statement that follows the while loop.

The problem for displaying `Welcome to Java!` a hundred times introduced in the preceding section is represented in a `while` loop. Its flowchart is shown in Figure 6.1b.

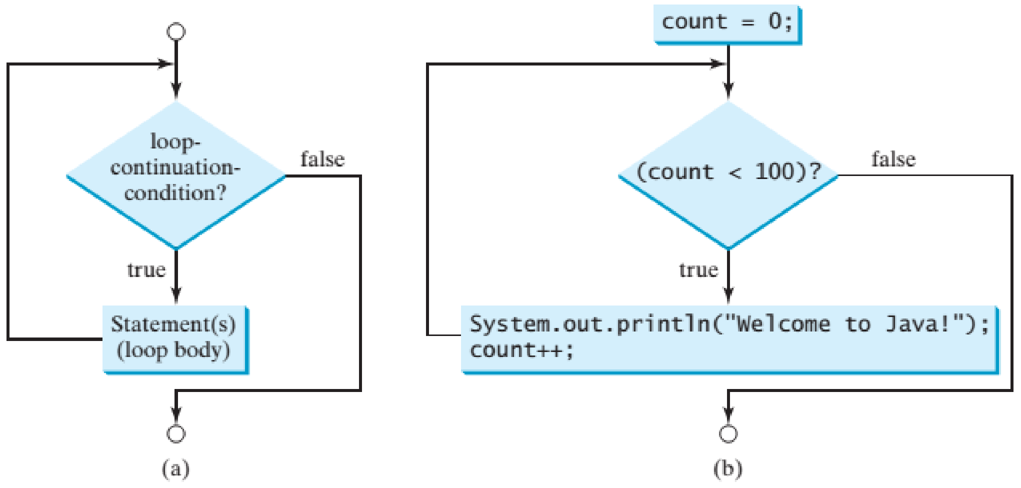
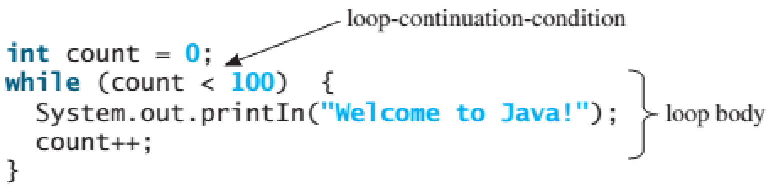


Figure 6.1 The while loop repeatedly executes the statements in the loop body when the loop-continuation-condition evaluates to true.

The loop-continuation-condition is `count < 100` and the loop body contains the following two statements:



```
int count = 0;
while (count < 100)  {
  System.out.println("Welcome to Java!");
  count++;
}
```

In this example, you know exactly how many times the loop body needs to be executed because the control variable `count` is used to count the number of executions. This type of loop is known as a **counter-controlled** loop.

The loop-continuation-condition must always appear inside the parentheses. **The braces enclosing the loop body can be omitted only if the loop body contains one or no statement.**

PROGRAM 6-1

```
public class SumOfNumbers {
 public static void main(String[] args) {
   int sum = 0, i = 1;
   while (i < 10) {
    sum = sum + i;
    i++;
   }
   System.out.println("sum is " + sum); // sum is 45
 }
}
```

PROGRAM 6-1 further shows how a loop works. If `i < 10` is true, the program adds i to `sum`. Variable i is initially set to 1, then is incremented to 2, 3, and up to 10. When i is 10, `i < 10` is false, so the loop exits. Therefore, the sum is 1 + 2 + 3 + ... + 9 = 45.

6.2.1 Infinite Loop

What happens if the loop is mistakenly written as follows?

```
int sum = 0, i = 1;
while (i < 10) {
 sum = sum + i;
}
```

This loop is infinite, because i is always 1 and `i < 10` will always be true. Make sure that the loop-continuation-condition eventually becomes false so that the loop will terminate. **A common programming error involves infinite loops** (i. e., the loop runs forever). If your program takes an unusually long time to run and does not stop, it may have an infinite loop.

6.2.2 Off-By-One Error

Programmers often make the mistake of executing a loop one more or less time. This is commonly known as the **off-by-one error**. For example, the following loop displays `Welcome to Java` 101 times rather than 100 times. The error lies in the condition, which should be `count < 100` rather than `count <= 100`.

```
int count = 0;
while (count <= 100) {
 System.out.println("Welcome to Java!");
 count++;
}
```

6.3 `do..while` Loop

A `do-while` loop is the same as a `while` loop except that it executes the loop body first and then checks the loop continuation condition.

The `do-while` loop is a variation of the `while` loop. Its syntax is:

```
do {
      // Loop body;
      Statement(s);
} while (loop-continuation-condition);
```

Its execution flowchart is shown in Figure 6.2. **The loop body is executed first, and then the loop-continuation-condition is evaluated. If the evaluation is true, the loop body is executed again; if it is false, the `do-while` loop terminates**.
You can write a loop using either the `while` loop or the `do-while` loop. Sometimes one is a more convenient choice than the other. For example, you can rewrite the `while` loop in PROGRAM 6-1 using a `do-while` loop, as shown in PROGRAM 6-2.

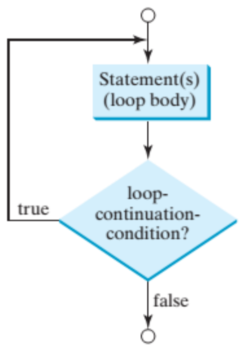


Figure 6.2 The `do-while` loop executes the loop body first, then checks the loopcontinuation-condition to determine whether to continue or terminate the loop.

PROGRAM 6-2

```
import java.util.Scanner;
public class TestDoWhile {
 public static void main(String[] args) {
   int data;
   int sum = 0;
   Scanner input = new Scanner(System.in);
   // Keep reading data until the input is 0
   do {
   // Read the next data
      System.out.print("Enter an integer (the input ends if it is 0):
")
      data = input.nextInt();
      sum += data;
   }while (data != 0);
   System.out.println("The sum is " + sum); }
}
```

PROGRAM OUTPUT

```
Enter an integer (the input ends if it is 0): 3
Enter an integer (the input ends if it is 0): 5
Enter an integer (the input ends if it is 0): 6
Enter an integer (the input ends if it is 0): 0
The sum is 14
```

6.4 `for` Loop

A `for` loop has a concise syntax for writing loops.

In general, the syntax of a `for` loop is:

```
for (initial-action;loop-continuation-condition;action-after-each-iteration) {
 // Loop body;
  Statement(s);
}
```

The flowchart of the `for` loop is shown in Figure 6.3a.

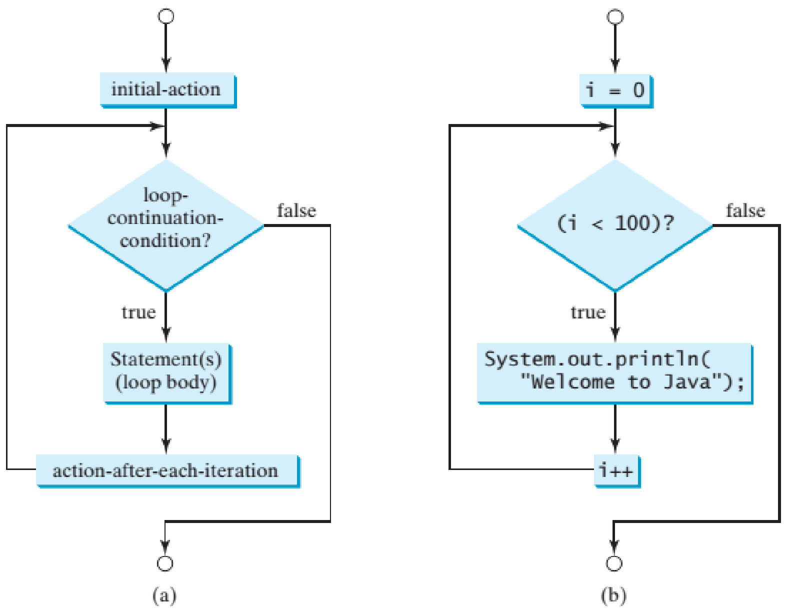

initial-action
loop-continuation-condition?
false
true
Statement(s) (loop body)
action-after-each-iteration
(a)
i = 0
(i < 100)?
false
true
System.out.println("Welcome to Java");
i++
(b)

Figure 6.3 A `for` loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the loop-continuation-condition evaluates to true.

The `for` loop statement starts with the keyword `for`, followed by a pair of parentheses enclosing the control structure of the loop. This structure consists of **initial-action**, **loop-continuation-condition**, and **action-after-each-iteration** separated by semi-colon. The control structure is followed by the loop body enclosed inside braces.

A `for` loop generally uses a variable to control how many times the loop body is executed and when the loop terminates. This variable is referred to as a **control variable**. The initial action often initializes a control variable, the action-after-each-iteration usually increments or decrements the control variable, and the loop-continuation-condition tests whether the control variable has reached a termination value. For example, the following `for` loop prints Welcome to Java! a hundred times:

PROGRAM 6-2

```
public class TestForLoop {
 public static void main(String[] args) {
   for (int i = 0; i < 100; i++) {
       System.out.println("Welcome to Java!");
   }
 }
}
```

The flowchart of the statement is shown in Figure 6.3(b). The `for` loop initializes i to 0, then repeatedly executes the `println` statement and evaluates `i++` while i is less than 100.

The initial-action, `i = 0`, initializes the control variable, i. The loop continuation-condition, `i < 100`, is a Boolean expression. **The expression is evaluated right after the initialization and at the beginning of each iteration. If this condition is true, the loop body is executed. If it is false, the loop terminates and the program control turns to the line following the loop.** The action-after-each-iteration, `i++`, is a statement that adjusts the control variable. This statement is executed after each iteration and increments the control variable. Eventually, the value of the control variable should force the loop-continuation-condition to become false; otherwise, the loop is infinite.

If there is only one statement in the loop body, as in this example, the braces can be omitted. The control variable must be declared inside the control structure of the loop or before the loop. If the loop control variable is used only in the loop, and not elsewhere, it is a good programming practice to declare it in the initial-action of the `for` loop.

The initial-action in a `for` loop can be a list of zero or more comma-separated variable declaration statements or assignment expressions. For example:

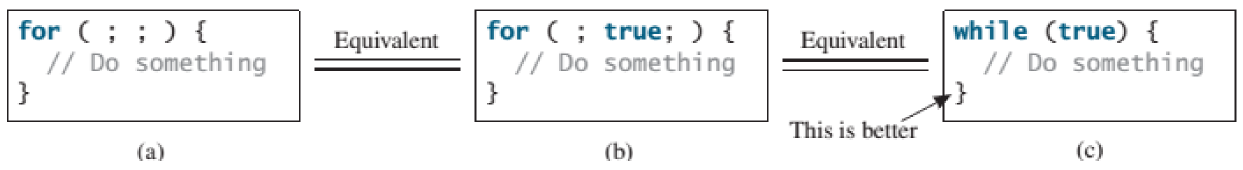
```
for (int i = 0, j = 0; i + j < 10; i++, j++) {
// Do something
}
```

The action-after-each-iteration in a `for` loop can be a list of zero or more comma-separated statements. For example:

```
for (int i = 1; i < 100; System.out.println(i), i++);
```

This example is correct, but it is a bad example, because it makes the code difficult to read. Normally, you declare and initialize a control variable as an initial action and increment or decrement the control variable as an action after each iteration.

Note: If the loop-continuation-condition in a `for` loop is omitted, it is implicitly true. Thus the statement given below in 6.4(a), which is an infinite loop, is the same as in 6.4(b). To avoid confusion, though, it is better to use the equivalent loop in 6.4(c).



```
for ( ; ; ) {
  // Do something
}
```

```
for ( ; true; ) {
  // Do something
}
```

```
while (true) {
  // Do something
}
```

Figure 6.4

6.5 Counter

A counter is a variable that is regularly incremented or decremented each time a loop iterates. Sometimes it is important for a program to control or keep track of the number of iterations a loop performs. For example, the PROGRAM 6-3 displays a table consisting of the numbers 1 through 10 and their squares, so its loop must iterate 10 times.

PROGRAM 6-3

```
public class SquareTable {
 public static void main(String[] args) {
   int num = 1; // Initialize the counter.
   System.out.println("Number          Number Squared");
   System.out.println("-----------------------------");
   while (num <= 10){
      System.out.printf("%d\t\t%d\n", num, (num * num));
      num++; // Increment the counter.
   }
  }
}
```

PROGRAM OUTPUT

```
Number      Number Squared
-------------------------
1           1
2           4
3           9
4           16
5           25
6           36
7           49
8           64
9           81
10          100
```

6.6 Sentinels

A sentinel is a special value that marks the end of a list of values. A sentinel is a special value that cannot be mistaken as a member of the list and signals that there are no more values to be entered. When the user enters the sentinel, the loop terminates.

The PROGRAM 6-4 calculates the total points earned by a soccer team over a series of games. It allows the user to enter the series of game points, then -1 to signal the end of the list. This type of loop is also **sentinel-controlled loop**.

PROGRAM 6-4

```
1   public class Sentinel {
2    public static void main(String[] args) {
3      Scanner input = new Scanner(System.in);
4      int game = 1; // Game counter
5      int points; // To hold a number of points
6      int total = 0; // Accumulator
7      System.out.println("Enter the number of points your team has
8   earned");
9      System.out.print("so far in the season, then enter -1 when
10  done.\n");
11     System.out.printf("Enter the points for game %d: ", game);
12     points = input.nextInt();
13     while (points != -1){
14        total += points;
15        game++;
16        System.out.printf("Enter the points for game %d: ", game);
17        points = input.nextInt();
18     }
19     System.out.printf("\nThe total points are %d", total);
     }
    }
```

PROGRAM OUTPUT

```
Enter the number of points your team has earned
so far in the season, then enter -1 when done.
Enter the points for game 1: 10
Enter the points for game 2: 20
Enter the points for game 3: 30
Enter the points for game 4: -1

The total points are 60
```

The value -1 was chosen for the sentinel in this program because it is not possible for a team to score negative points. Notice that this program performs a preliminary read in Line 10 to get the first value. This makes it possible for the loop to immediately terminate if the user enters -1 as the first value. Also note that the sentinel value is not included in the running total.

6.7 Accumulators

A running total is a sum of numbers that accumulates with each iteration of a loop. **The variable used to keep the running total is called an accumulator**. Many programming tasks require the calculation of the total of a series of numbers. Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

It is often said that the loop keeps a running total because it accumulates the total as it reads each number in the series. When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop.

Notice that the first step in the flowchart Figure 6.5 is to set the accumulator variable to 0. This is a critical step. Each time the loop reads a number, it adds it to the accumulator. If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.

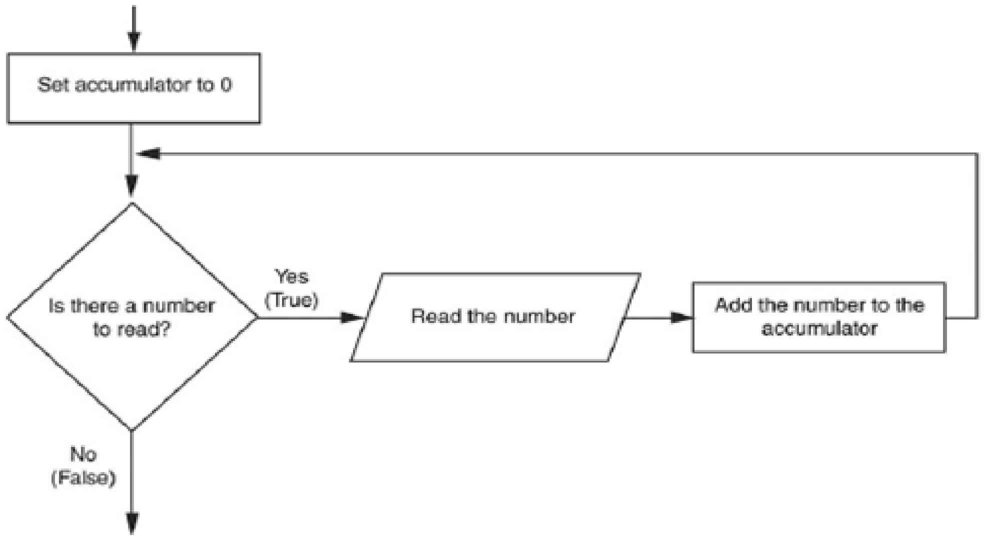


Figure 6.5 The general logic of a loop that calculates running total (Accumulator)

PROGRAM 6-5 calculates a company's total sales over a period of time by taking daily sales figures as input and calculating a running total of them as they accumulate.

PROGRAM 6-5

```
public class Accumulator {
 public static void main(String[] args) {
   Scanner input = new Scanner(System.in);
   int days; // Number of days
   double total = 0.0; // Accumulator, initialized with 0
   double sales = 0.0;
   System.out.println("For how many days do you have sales figures? ");
   days = input.nextInt(); // Get the number of days.
   // Get the sales for each day and accumulate a total.
   for (int count1 = 1; count1 <= days; count1++){
       System.out.printf("Enter the sales for day %d : ", count1);
       sales = input.nextDouble();
        total += sales; // Accumulate the running total.
     }
     System.out.printf("The total sales are $%.2f", total);
  }
}
```

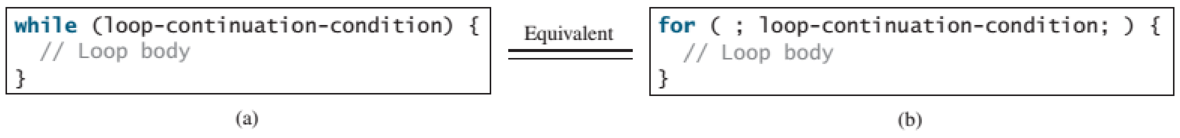
PROGRAM OUTPUT

```
For how many days do you have sales figures?
5
Enter the sales for day 1 : 40
Enter the sales for day 2 : 30
Enter the sales for day 3 : 50
Enter the sales for day 4 : 20
Enter the sales for day 5 : 70
The total sales are $210.00
```

6.8 Which Loop To Use?

You can use any of the loops whichever is convenient. The `while` loop and `for` loop are called **pretest loops** because the continuation condition is checked before the loop body is executed. The **`do-while`** loop is called a **posttest** loop because the condition is checked after the loop body is executed.

The three forms of loop statement — `while`, `do-while`, and `for`—are expressively equivalent; that is, you can write a loop in any of these three forms. For example, a `while` loop in 6.6(a) in the following figure can always be converted into the `for` loop in 6.6(b).

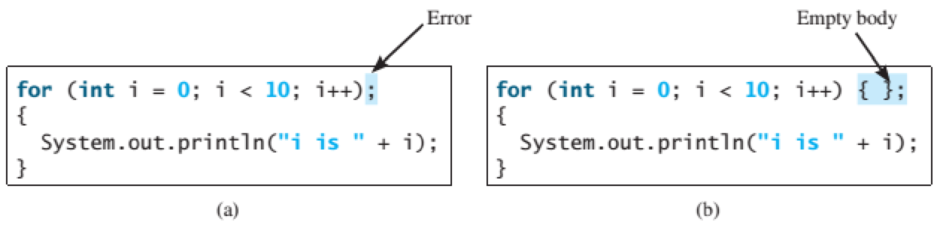


```
while (loop-continuation-condition) {
  // Loop body
}
```

```
for ( ; loop-continuation-condition; ) {
  // Loop body
}
```

Figure 6.6

Use the loop statement that is most intuitive and comfortable for you. In general, a `for` loop may be used if the number of repetitions is known in advance, as, for example, when you need to display a message a hundred times. A `while` loop may be used if the number of repetitions is not fixed, as in the case of reading the numbers until the input is 0. A `do-while` loop can be used to replace a `while` loop if the loop body has to be executed before the continuation condition is tested. Adding a semicolon at the end of the `for` clause before the loop body is a common mistake, as shown below in 6.7(a). In 6.7(a), the semicolon signifies the end of the loop prematurely. The loop body is actually empty, as shown in 6.7(b). 6.7(a) and (b) are equivalent. Both are incorrect.

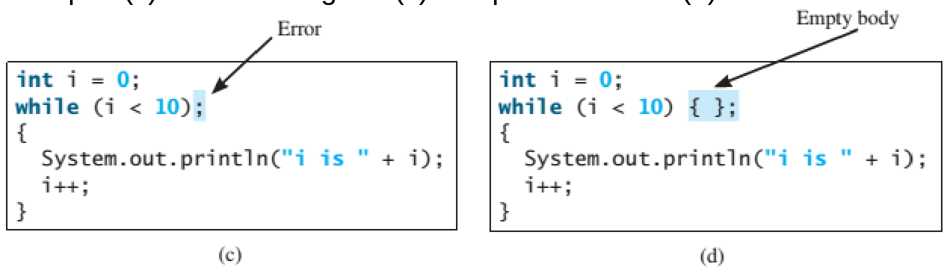


```
for (int i = 0; i < 10; i++);
{
  System.out.println("i is " + i);
}
```

```
for (int i = 0; i < 10; i++) { };
{
  System.out.println("i is " + i);
}
```

Figure 6.7

Similarly, the loop in (c) is also wrong. 6.8(c) is equivalent to 6.8(d). Both are incorrect.

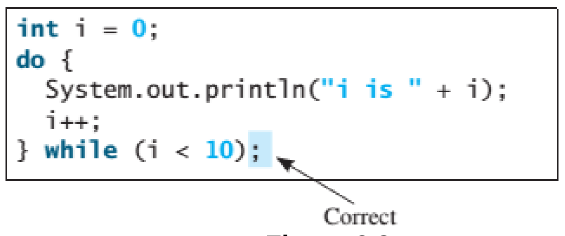


```
int i = 0;
while (i < 10);
{
  System.out.println("i is " + i);
  i++;
}
```

```
int i = 0;
while (i < 10) { };
{
  System.out.println("i is " + i);
  i++;
}
```

Figure 6.8

These errors often occur when you use the next-line block style. Using the end-of-line lock style can avoid errors of this type. In the case of the `do-while` loop, the semicolon is needed to end the loop.



```
int i = 0;
do {
  System.out.println("i is " + i);
  i++;
} while (i < 10);
```

Figure 6.9

6.6 Nested Loops

A loop can be nested inside another loop. Nested loops consist of an **outer loop** and **one or more inner loops**. Each time the outer loop is repeated, the inner loops are reentered, and started anew.

A clock is a good example of something that works like a nested loop. The second hand, minute hand, and hour hand all spin around the face of the clock. The hour hand, however, only makes one revolution for every 12 of the minute hand's revolutions. And it takes 60 revolutions of the second hand for the minute hand to make one revolution. This means that for every complete revolution of the hour hand, the second hand has revolved 720 times. PROGRAM 6-6 simulates a digital clock. It displays the seconds from 0 to 59, etc.

PROGRAM 6-6

```
public class DigitalClockSimulator {
 public static void main(String[] args) {
   int hours, minutes, seconds;
   for (hours = 0; hours < 24; hours++){
     for (minutes = 0; minutes < 60; minutes++){
        for (seconds = 0; seconds < 60; seconds++){
          System.out.printf("%02d:%02d:%02d\n", hours, minutes,
seconds);
        }
     }
   }
  }
}
```

PROGRAM OUTPUT

```
00:00:00
00:00:01
00:00:02
. (The program will count through each second of 24 hours.)
.
23:59:59
```

The innermost loop will iterate 60 times for each iteration of the middle loop. The middle loop will iterate 60 times for each iteration of the outermost loop. When the outermost loop has iterated 24 times, the middle loop will have iterated 1,440 times and the innermost loop will have iterated 86,400 times. The simulated clock example brings up a few points about nested loops:

- An inner loop goes through all of its iterations for each iteration of an outer loop.
- Inner loops complete their iterations faster than outer loops.
- To get the total number of iterations of a nested loop, multiply the number of iterations of all the loops.

6.7 Keywords `break` and `continue`

The `break` and `continue` keywords can be used in loop statements to provide additional controls in a loop. Using `break` and `continue` can simplify programming in some cases. Overusing or improperly using them, however, can make programs difficult to read and debug.

You have used the keyword `break` in a switch statement. PROGRAM 6-7 presents a program to demonstrate the effect of using `break` to immediately terminate in a loop.

PROGRAM 6-7

```
1   public class TestBreak {
2    public static void main(String[] args) {
3      int sum = 0, number = 0;
4      while (number < 20){
5        number++;
6        sum += number;
7        if (sum >= 100)
8          break;
9       }
10     System.out.println("The number is " + number);
11     System.out.println("The sum is " + sum);
12   }
13  }
```

PROGRAM OUTPUT

```
The number is 14
The sum is 105
```

The program adds integers from 1 to 20 in this order to sum until sum is greater than or equal to 100. Without the `if` statement (line 7), the program calculates the sum of the numbers from 1 to 20. But with the `if` statement, the loop terminates when sum becomes greater than or equal to 100.

PROGRAM 6-8 presents a program to demonstrate the effect of using `continue` in a loop. When it is encountered, **it ends the current iteration and program control goes to the end of the loop body**. In other words, `continue` breaks out of an iteration, while the `break` keyword breaks out of a loop.

PROGRAM 6-8

```
1   public class TestContinue {
2    public static void main(String[] args) {
3      int sum = 0;
4      int number = 0;
5      while (number < 20){
6        number++;
7        if (number ==10 || number == 11)
8             continue;
9        sum += number;
10     }
11     System.out.println("The sum is " + sum); }
12  }
```

PROGRAM OUTPUT

```
The sum is 189.
```

PROGRAM 6-7 adds integers from 1 to 20 except 10 and 11 to sum. With the `if` statement in the program (line 8), the `continue` statement is executed when number becomes 10 or 11. The `continue` statement ends the current iteration so that the rest of the statement in the loop body is not executed; therefore, number is not added to sum when it is 10 or 11.

The `continue` statement is always inside a loop. In the `while` and `do-while` loops, the loop-continuation-condition is evaluated immediately after the `continue` statement. In the `for` loop, the action-after-each-iteration is performed, then the loop-continuation-condition is evaluated, immediately after the `continue` statement.

[Reference]

[1] Introduction to Java Programming Comprehensive Version 10th Ed, Daniel Liang, 2016.
[2] Java How To Program 10th Ed, Paul Deitel, Harvey Deitel, 2016

[Self-Review Question]

1. Consider the following segment of code.

```
int sum = 0;
for (int i = 0; i <= 4; ++i){
   for(int j = i; j > 1; --j) {
          sum += (j * i);
   }
}
```

   What is the final value of sum? Rewrite the above code using while loops.

2. Can you use a break; statement to break out of an `if` statement?

3. Suppose the input is 2 3 4 5 0. What is the output of the following code?

```
import java.util.Scanner;
public class Test {
  public static void main(String[] args) {
     Scanner input = new Scanner(System.in);
     int number, sum = 0, count;
     for (count = 0; count < 5; count++) {
        number = input.nextInt();
        sum += number;
     }
     System.out.println("sum is " + sum);
     System.out.println("count is " + count);
  }
}
```

4. Will the following programs terminate? If so, give the output.

```
int balance = 10;
while (true) {
   if (balance < 9)
      break;
   balance = balance - 9;
}
System.out.println("Balance is " + balance);
```

5. Will the following programs terminate? If so, give the output.

```
int balance = 10;
while (true) {

  if (balance < 9)
     continue;
      balance = balance - 9;
}
System.out.println("Balance is " + balance);
```