

# Fundamentals of Java

## Session: 5

### Looping Constructs

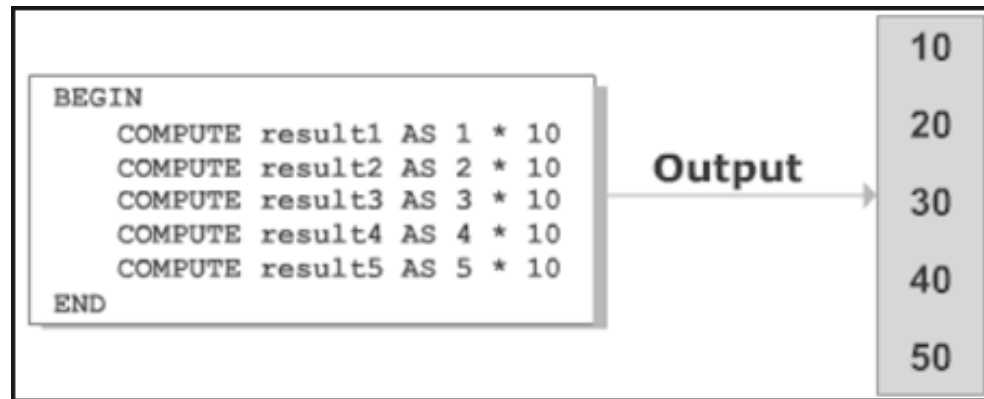




- ◆ List the different types of loops
- ◆ Explain the while statement and the associated rules
- ◆ Identify the purpose of the do-while statement
- ◆ State the need of for statement
- ◆ Describe nested loops
- ◆ Compare the different types of loops
- ◆ State the purpose of jump statements
- ◆ Describe break statement
- ◆ Describe continue statement



- ◆ A computer program consists of a set of statements, which are usually executed sequentially.
- ◆ However, in certain situations, it is necessary to repeat certain steps to meet a specified condition.
- ◆ Following figure shows the program that displays the multiples of 10:



- ◆ The same statement is repeating 5 times to display the multiple of 10 with 1, 2, 3, 4, and 5.
- ◆ Thus, a loop can be used in this situation.



- ◆ The loop statements supported by Java programming language are as follows:

Loops enable programmers to develop concise programs, which otherwise would require thousands of program statements.

Loops consists of statement or a block of statements that are repeatedly executed.

Statements in the loops are executed until a condition evaluates to `true` or `false`.



# 'while' Statement 1-5



- ◆ It is the most fundamental looping statement in Java.
- ◆ It executes a statement or a block of statements until the specified condition is `true`.
- ◆ It is used when the number of times the block has to be executed is not known.
- ◆ The syntax to use the `while` statement is as follows:

## Syntax

```
while (expression) {  
    // one or more statements  
}
```

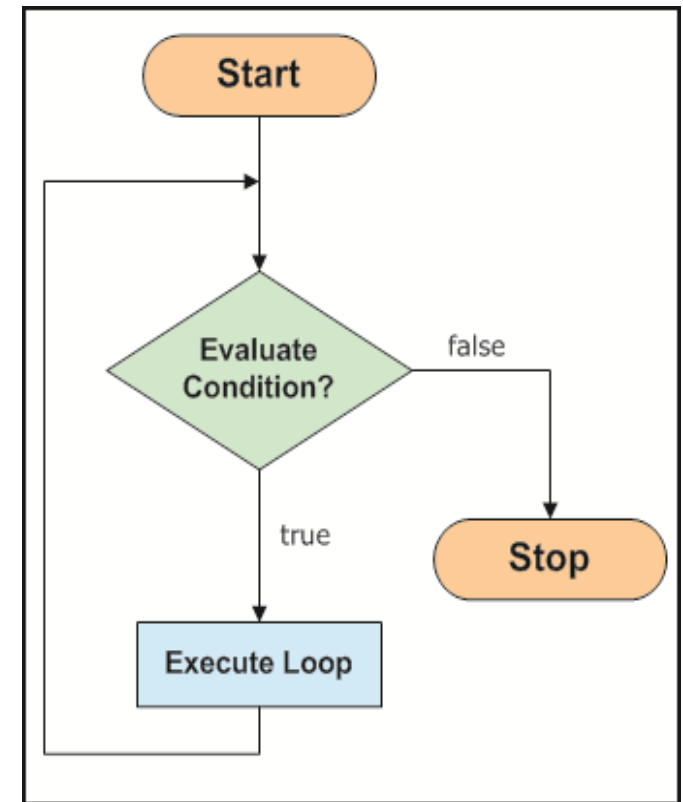
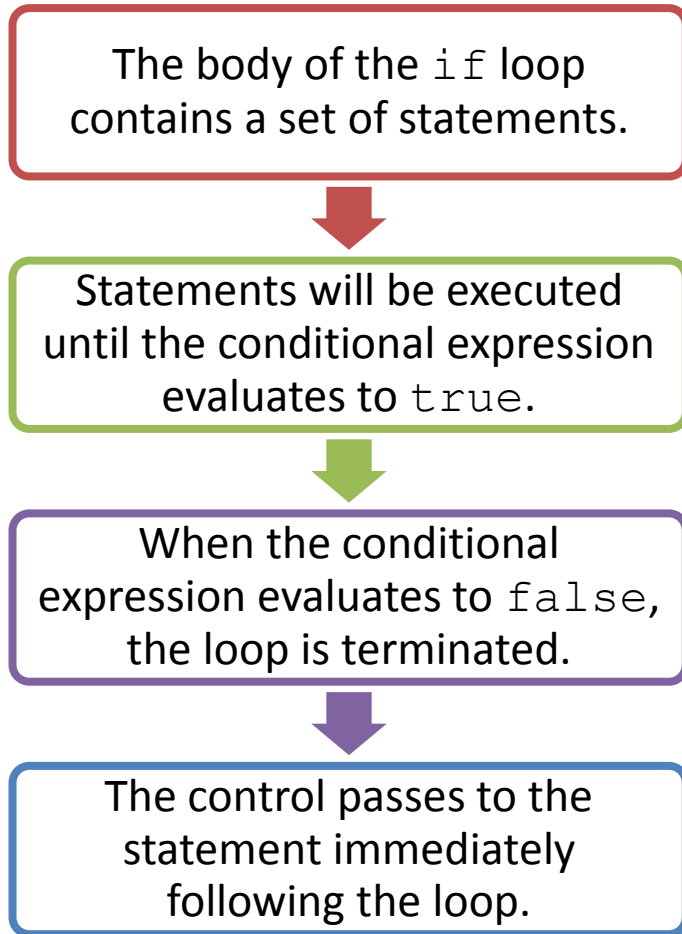
where,

- ◆ `expression`: Is a conditional expression which must return a boolean value, that is, `true` or `false`.
- ◆ The use of curly braces (`{ }`) is optional and can be avoided, if there is only a single statement within the body of the loop. However, providing statements within the curly braces increases the readability of the code.

## 'while' Statement 2-5



- ◆ Following figure shows the flow of execution of `while` loop:



## 'while' Statement 3-5



- ◆ Following code snippet demonstrates the code that displays multiples of 10 using the `while` loop:

```
public class PrintMultiplesWithWhileLoop {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        // Variable, num acts as a counter variable  
        int num = 1;  
        // Variable, product will store the result  
        int product = 0;  
    }  
}
```

- ◆ An integer variable, **num** is declared to store a number and is initialized to 1.
- ◆ It is used in the `while` loop to start multiplication from 1.

# 'while' Statement 4-5



```
// Tests the condition at the beginning of the loop
while (num <= 5) {
    product = num * 10;
    System.out.printf("\n %d * 10 = %d", num, product);
    num++; // Equivalent to n = n + 1
} // Moves the control back to the while statement

// Statement gets printed on loop termination
System.out.println("\n Outside the Loop");
}
```

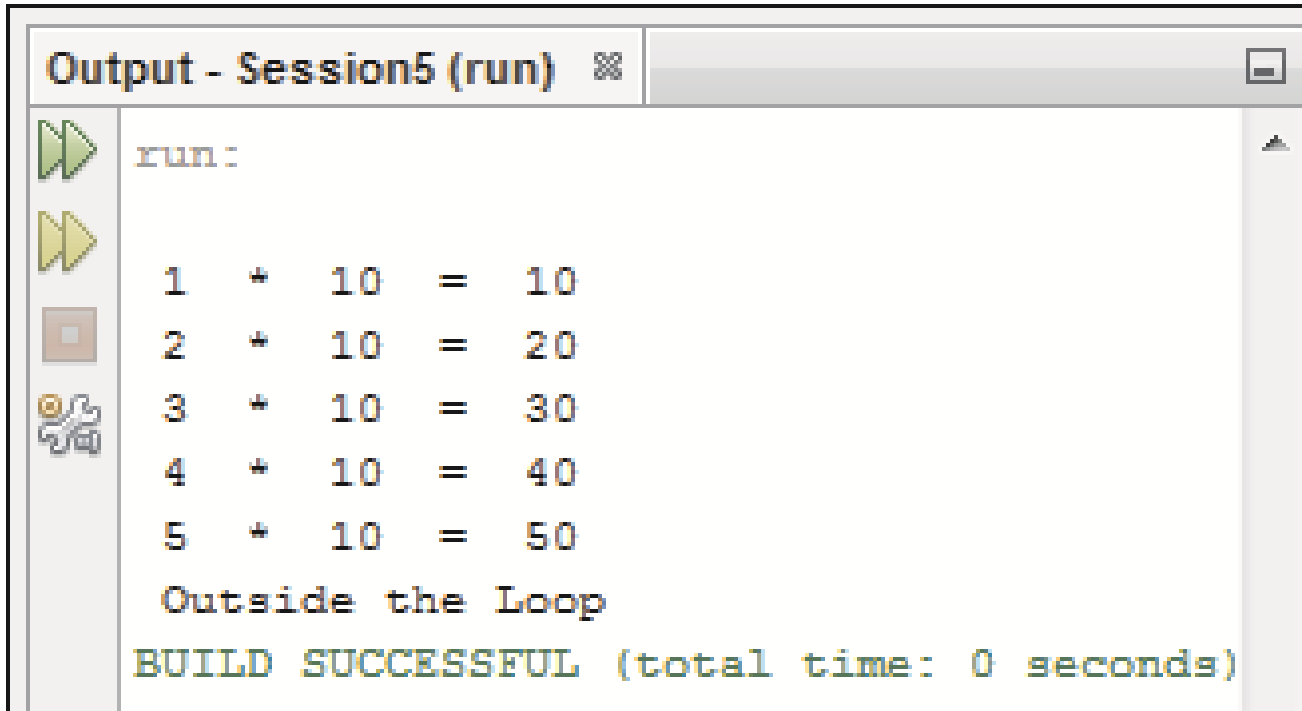
- ◆ The conditional expression:
  - ◆ **num <= 5** is evaluated at the beginning of the while loop. The loop is executed only if the conditional expression evaluates to `true`.
  - ◆ In this case, as the value in the variable, **num** is less than 5, hence, the statements present in the body of the loop is executed.
- ◆ The first statement within the body of the loop calculates the product by multiplying **num** with 10.
- ◆ The next statement prints this value.
- ◆ The last statement **num++** increments the value of **num** by 1.
- ◆ The execution of the loop stops when condition becomes `false`, that is, when the value of **num** reaches 6.



## 'while' Statement 5-5



- ◆ Finally, the statement, '**Outside the Loop**' is displayed.
- ◆ Following figure shows the output of the code:



```
run:
1 * 10 = 10
2 * 10 = 20
3 * 10 = 30
4 * 10 = 40
5 * 10 = 50
Outside the Loop
BUILD SUCCESSFUL (total time: 0 seconds)
```



- ◆ There are situations when it is required to write a loop without any action statement to delay a process.
- ◆ Such a loop is referred to as null statement loop.
- ◆ **Null statement loop:**
  - ◆ There are no statements in the body of the loop.
  - ◆ The loop is terminated with a semicolon.
- ◆ Following code snippet demonstrates a code that prints the midpoint of two numbers with an empty while loop:

```
public class TestWhileEmptyBody {  
    /**  
     * @param args the command line arguments  
     */  
  
    public static void main(String[] args) {  
        int num1 = 1;  
        int num2 = 30;
```

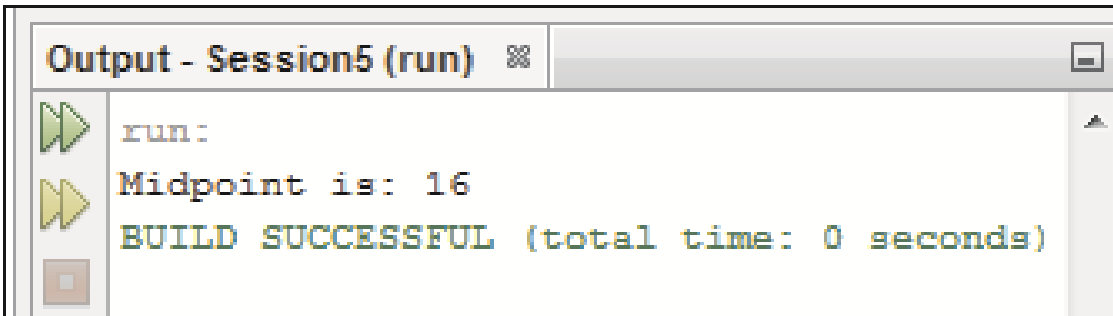
# Null Statement in Loops 2-2



```
// An empty while loop with no statements
while (++num1 < --num2);

// The statement executes after the while loop is completed
System.out.println("Midpoint is: " + num1);
}
}
```

- ◆ The value of **num1** is incremented and the value of **num2** is decremented.
- ◆ The loop repeats till the value of **num1** is equal to or greater than **num2**.
- ◆ Thus, upon exit **num1** will hold a value that is midway between the original values of **num1** and **num2**.
- ◆ Following figure shows the output of the code:



```
Output - Session5 (run)
run:
Midpoint is: 16
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Rules for Using 'while' Loop



- ◆ The following points should be noted when using the `while` statement:
  - ◆ The value of the variables used in the expression must be set once before the execution of the loop. For example, **`num = 1;`**
  - ◆ The body of the loop must have an expression that changes the value of the variable which is a part of the loop's expression. For example, **`num++;`** or **`num--;`**



- ◆ An infinite loop is one which never terminates.
- ◆ It runs infinitely when the conditional expression or the increment/decrement expression of the loop is missing.
- ◆ Following code snippet shows the implementation of an infinite loop using the `while` statement:

```
public class InfiniteWhileLoop {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        /*  
         * Loop begins with a boolean value true and is executed  
         * infinitely as the terminating condition is missing  
         */  
        while (true) {  
            System.out.println("Welcome to Loops...");  
        } //End of the while loop  
    }  
}
```

- ◆ The loop never terminates as the expression always returns a `true` value.

# 'do-while' Statement 1-4



- ◆ It checks the condition at the end of the loop rather than at the beginning.
- ◆ It ensures that the loop is executed at least once.
- ◆ It comprises a condition expression that evaluates to a `boolean` value.
- ◆ The syntax to use the `do-while` statement is as follows:

## Syntax

```
do {  
    statement(s);  
} while (expression);
```

where,

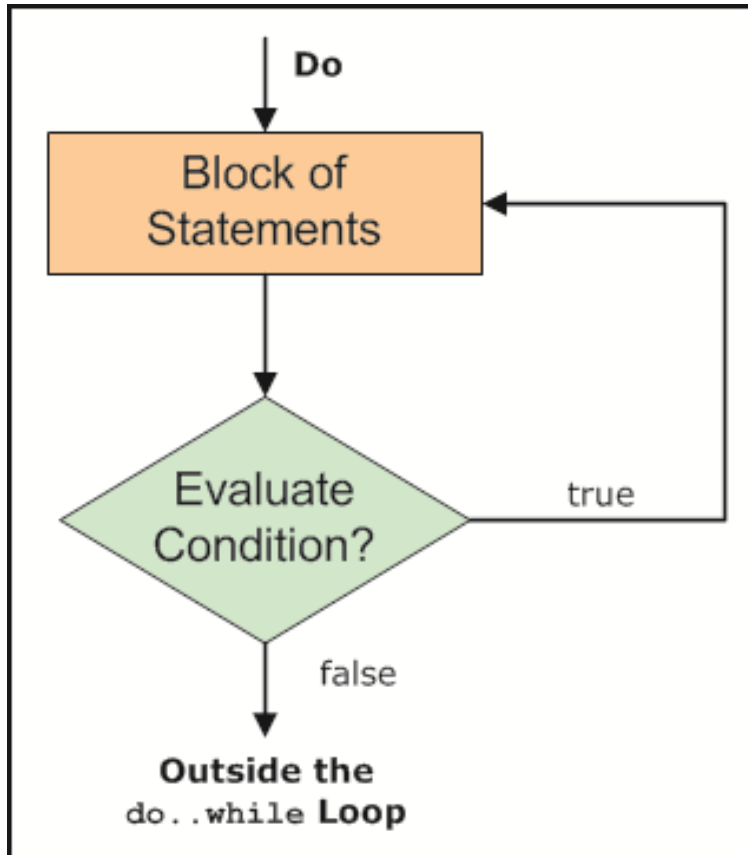
`expression`: A conditional expression which must return a boolean value, that is, `true` or `false`.

`statement(s)`: Indicates body of the loop with a set of statements.

# 'do-while' Statement 2-4



- ◆ Following figure shows the flow of execution for the `do-while` loop:



For each iteration, the `do-while` loop first executes the body of the loop and then, the conditional expression is evaluated.

When the conditional expression evaluates to `true`, the body of the loop executes.

When the conditional expression evaluates to `false`, the loop terminates.

The statement following the loop is executed.

## 'do-while' Statement 3-4



- ◆ Following code snippet demonstrates the use of `do-while` loop for finding the sum of 10 numbers:

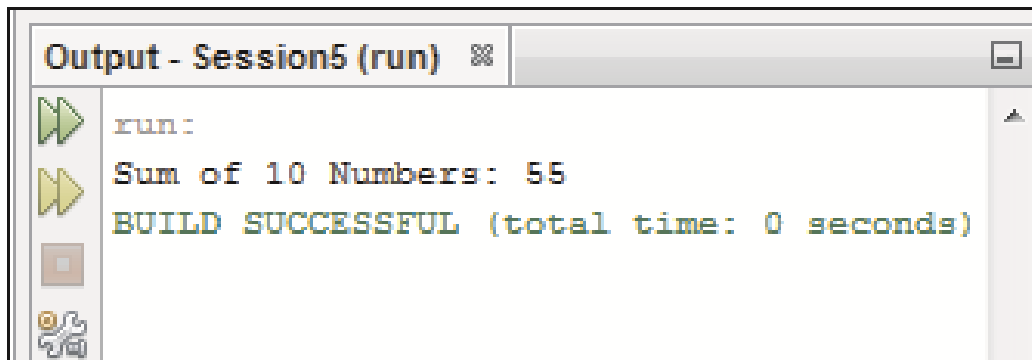
```
public class SumOfNumbers {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        int num = 1, sum = 0;  
  
        /**  
         * The body of the loop is executed first, then the condition is  
         * evaluated  
         */  
        do {  
            sum = sum + num;  
            num++;  
        } while (num <= 10);  
  
        // Prints the value of variable after the loop terminates  
        System.out.printf("Sum of 10 Numbers: %d\n", sum);  
    }  
}
```



# 'do-while' Statement 4-4



- ◆ Two integer variables, **num** and **sum** are declared and initialized to 1 and 0 respectively.
- ◆ The loop block begins with a `do` statement.
- ◆ The first statement in the body of the loop calculates the value of **sum** by adding the current value of **sum** with **num**.
- ◆ The next statement in the loop increments the value of **num** by 1.
- ◆ The condition, **num** `<=` 10, included in the `while` statement is evaluated.
- ◆ If the condition is met, the instructions in the loop are repeated.
- ◆ After the loop terminates, the value in the variable **sum** is printed.
- ◆ Following figure shows the output of the code:



```
run:  
Sum of 10 Numbers: 55  
BUILD SUCCESSFUL (total time: 0 seconds)
```



## 'for' Statement

- Used when the user knows the number of times the statements need to be executed.
- Statements within the body of the loop are executed as long as the condition is `true`.
- Condition is checked before the statements are executed.

# 'for' Statement 2-5



- ◆ The syntax to use the `for` statement is as follows:

## Syntax

```
for(initialization; condition; increment/decrement) {  
    // one or more statements  
}
```

where,

`initialization`: Is an expression that will set the initial value of the loop control variable.

`condition`: Is a boolean expression that tests the value of the loop control variable. If the condition expression evaluates to `true`, the loop executes, else terminates.

`increment/decrement`: Increments or decrements the value of control variable (s) in each iteration, till the condition specified in the condition section is reached. Typically, increment and decrement operators, such as `++`, `--`, and shortcut operators, such as `+=` or `-=` are used in this section.

# 'for' Statement 3-5



- ◆ Following figure shows the flow of execution for the `for` statement:

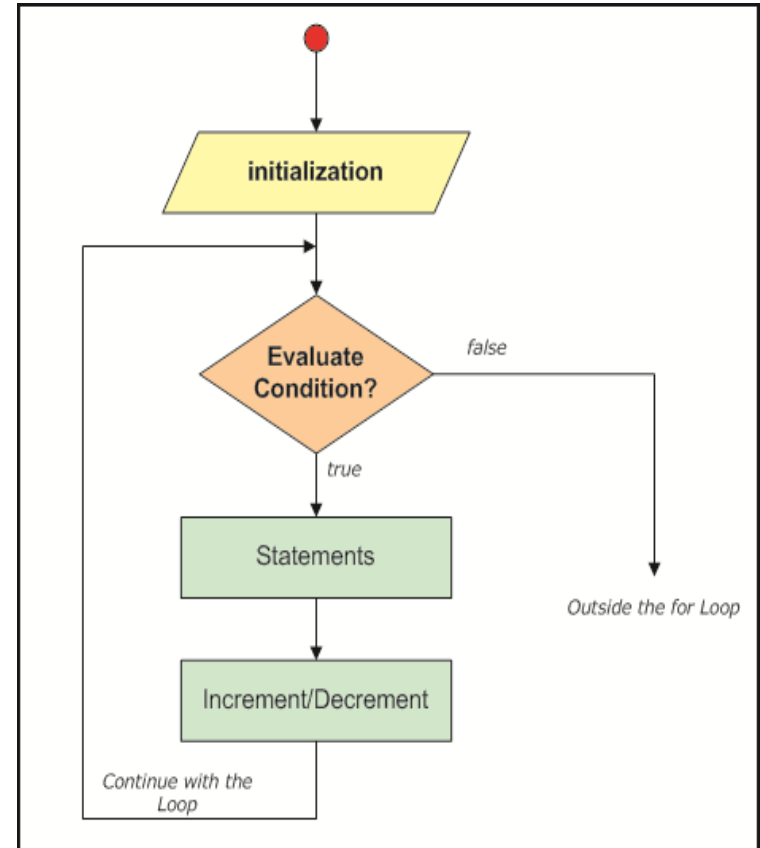
The initialization expression is executed only once, that is, when the loop starts.

Next, the boolean expression is evaluated and tests the loop control variable against a targeted value.

If the expression is `true`, then the body of the loop is executed and if the expression is `false`, then the loop terminates.

Lastly, the iteration portion of the loop is executed. This expression usually increments or decrements value of the control variable.

In the next iteration, again the condition section is evaluated and depending on the result of evaluation the loop is either continued or terminated.



# 'for' Statement 4-5



- ◆ Following code snippet demonstrates the use of `for` statement for displaying the multiples of 10:

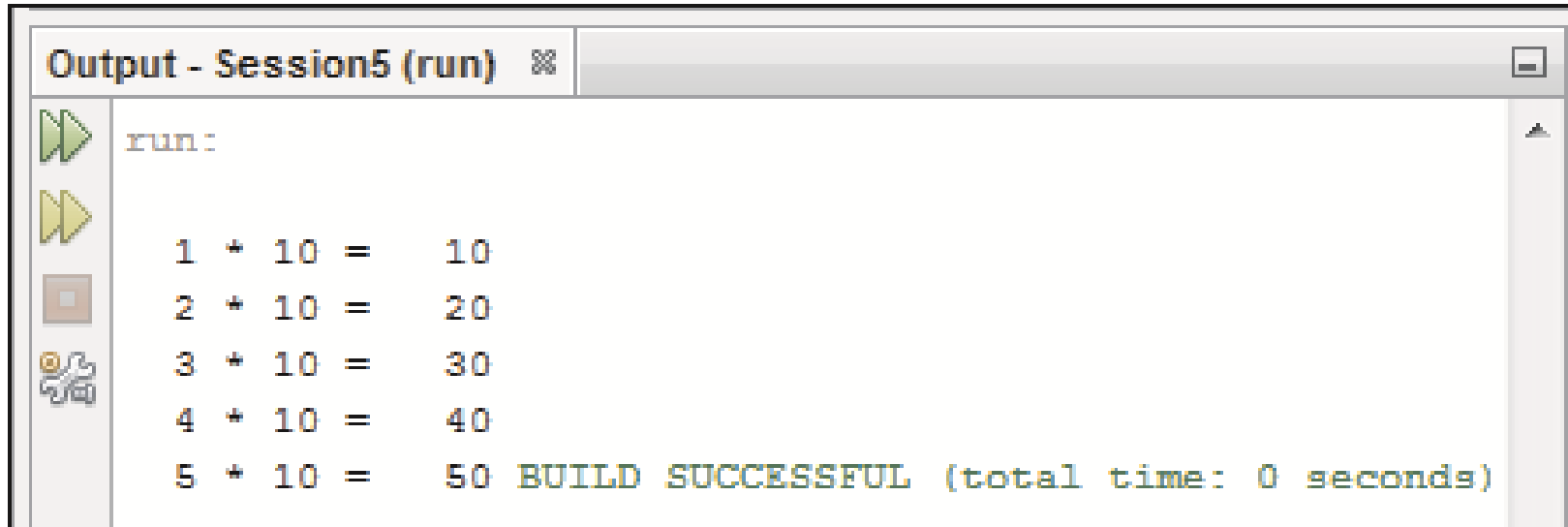
```
public class PrintMultiplesWithForLoop {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        int num, product;  
  
        // The for Loop with all the three declaration parts  
        for (num = 1; num <= 5; num++) {  
            product = num * 10;;  
            System.out.printf("\n % d * 10 = % d ", num, product);  
        } // Moves the control back to the for loop  
    }  
}
```

- ◆ In the initialization section of the `for` loop, the **num** variable is initialized to 1.
- ◆ The condition statement, **num <= 5**, ensures that the **for** loop executes as long as num is less than or equal to 5.
- ◆ The increment statement, **num++**, increments the value of **num** by 1.
- ◆ Finally, the loop terminates when the condition becomes **false**, that is, when the value of **num** becomes equal to 6.

# 'for' Statement 5-5



- ◆ Following figure shows the output of the code:



```
run:
1 * 10 = 10
2 * 10 = 20
3 * 10 = 30
4 * 10 = 40
5 * 10 = 50 BUILD SUCCESSFUL (total time: 0 seconds)
```



## ◆ Control Variables:

- ◆ Are used within the `for` loops and may not be used further in the program.
- ◆ It is possible to restrict the scope of variables by declaring them at the time of initialization.
- ◆ Following code snippet declares the counter variable inside the `for` statement:

```
public class ForLoopWithVariables {  
    /**  
     * @param args the command line arguments  
     */  
  
    public static void main(String[] args) {  
        int product;  
  
        // The counter variable, num is declared inside the for loop
```

## Scope of Control Variable in 'for' Statement 2-2



```
for (int num = 1; num <= 5; num++) {  
  
    product = num * 10;;  
    System.out.printf("\n % d * 10 = % d ", num, product);  
  
} // End of the for loop  
}  
}
```

- ◆ In the code, the variable **num** has been declared inside the `for` statement.
- ◆ This restricts the scope of the variable, **num** to the `for` statement and completes when the loop terminates.





## ◆ Expressions:

- ◆ The `for` statement can be extended by including more than one initialization or increment expressions.
  - ◆ Expressions are separated by using the 'comma' ( , ) operator.
  - ◆ Expressions are evaluated from left to right.
  - ◆ The order of the evaluation is important, if the value of the second expression depends on the newly calculated value.
- ◆ Following code snippet demonstrates the use of `for` loop to print the addition table for two variables using the 'comma' operator:

```
public class ForLoopWithComma {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        int i, j;  
        int max = 10;
```

# Use of Comma Operator in 'for' Statement 2-3



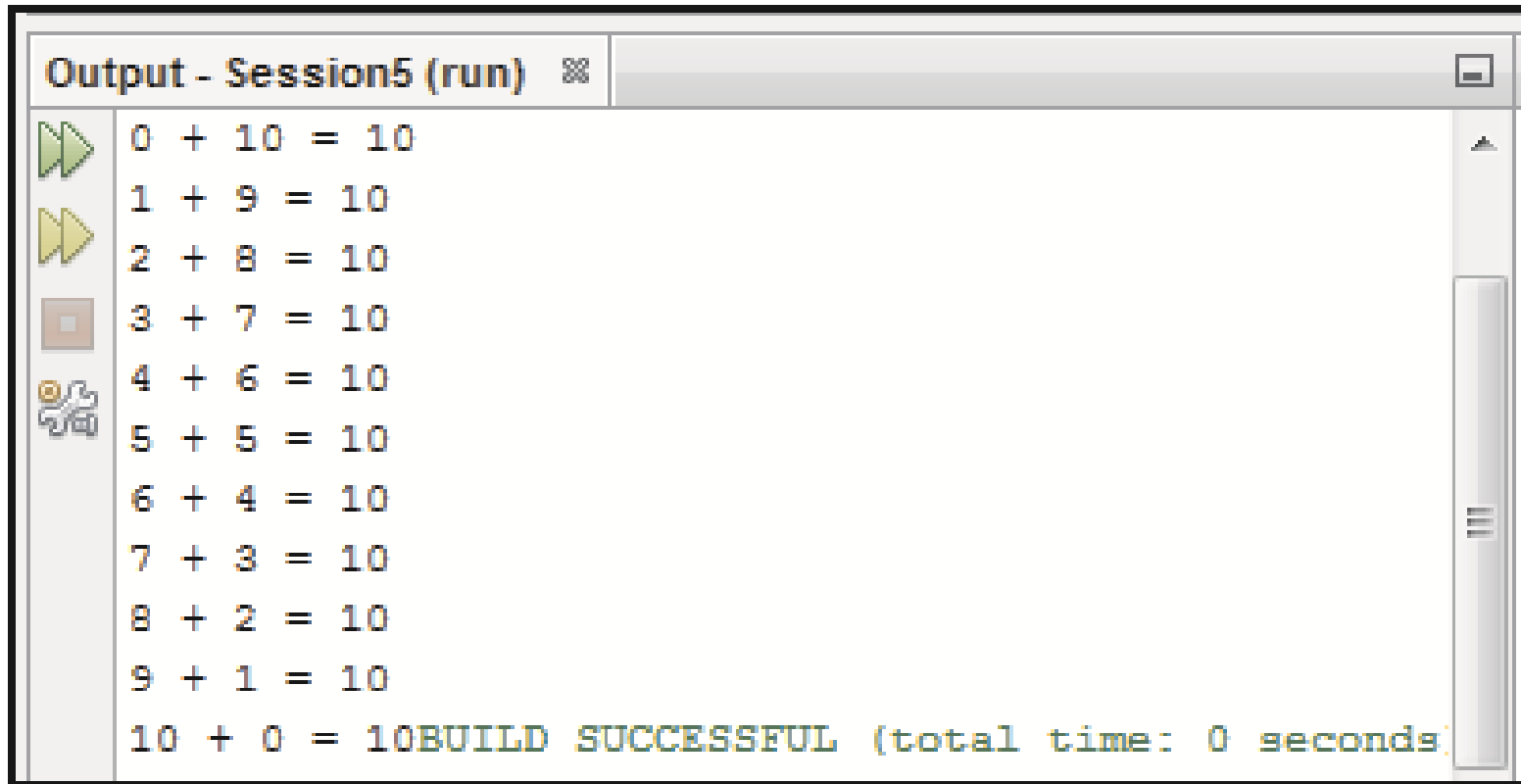
```
/*
 * The initialization and increment/decrement section includes
 * more than one variable
 */
for (i = 0, j = max; i <= max; i++, j--) {
    System.out.printf("\n%d + %d = %d", i, j, i + j);
}
}
```

- ◆ Three integer variables **i**, **j**, and **max** are declared.
- ◆ The variable **max** is assigned a value **10**. The **i** variable is assigned a value of **0** and **j** is assigned the value of **max**, that is, **10**. Thus, two parameters are initialized using a 'comma' operator.
- ◆ The condition statement, **i <= max**, ensures that the `for` loop executes as long as **i** is less than or equal to **max** that is **10**.
- ◆ Finally, the iteration expression again consists of two expressions, **i++**, **j--**.
- ◆ After each iteration, **i** is incremented by 1 and **j** is decremented by 1.
- ◆ The sum of these two variables which is always equal to **max** is printed.

# Use of Comma Operator in 'for' Statement 3-3



- ◆ Following figure shows the output of the code:



The screenshot shows an IDE window titled "Output - Session5 (run)". The output displays a series of arithmetic operations where the sum of two numbers equals 10, ranging from 0 + 10 to 10 + 0. The final line of the output is "BUILD SUCCESSFUL (total time: 0 seconds)".

```
0 + 10 = 10
1 + 9 = 10
2 + 8 = 10
3 + 7 = 10
4 + 6 = 10
5 + 5 = 10
6 + 4 = 10
7 + 3 = 10
8 + 2 = 10
9 + 1 = 10
10 + 0 = 10BUILD SUCCESSFUL (total time: 0 seconds)
```

# Variation in 'for' Loop 1-2



- ◆ The most common variation involves the conditional expression which can be:
  - ◆ Tested with the targeted values, but, it can also be used for testing `boolean` expressions.
- ◆ Alternatively, the initialization or the iteration section in the `for` loop may be left empty, that is, they need not be present in the `for` loop.
- ◆ Following code snippet demonstrates the use of `for` loop without the initialization expression:

```
public class ForLoopWithNoInitialization {  
  
    public static void main(String[] args) {  
  
        /*  
         * Counter variable declared and initialized outside for loop  
         */  
        int num = 1;  
  
        /*  
         * Boolean variable initialized to false  
         */  
        boolean flag = false;  
    }  
}
```

## Variation in 'for' Loop 2-2



```
/*
 * The for loop starts with num value 1 and continues till value of
 * flag is not true
 */
for (; !flag; num++) {
    System.out.println("Value of num: " + num);
    if (num == 5) {
        flag = true;
    }
} // End of for loop
}
```

- ◆ The `for` loop in the code continues to execute till the value of the variable **flag** is set to `true`.
- ◆ Following figure shows the output of the code:

# Infinite 'for' Loop



- ◆ If all the three expressions are left empty, then it will lead to an infinite loop.
- ◆ The infinite `for` loop will run continuously because there is no terminating condition.
- ◆ Following code snippet demonstrates the code for the infinite `for` loop:

```
.....  
  
    for( ; ; ) {  
        System.out.println("This will go on and on");  
    }  
  
.....
```

- ◆ The code will print 'This will go on and on' until the loop is terminated manually.
- ◆ Infinite loops make the program run indefinitely for a long time resulting in the consumption of all resources and stopping the system.



- ◆ It is designed to retrieve or traverse through a collection of objects, such as an array.
- ◆ It is also used to iterate over the elements of the collection objects, such as `ArrayList`, `LinkedList`, `HashSet`, and so on defined in the collection framework.
- ◆ It continues till all the elements from a collection are retrieved.
- ◆ The syntax for using the enhanced `for` loop is as follows:

## Syntax

```
for (type var: collection) {  
    // block of statement  
}
```

where,

`type`: Specifies the type of collection that is traversed.

`var`: Is an iteration variable that stores the elements from the collection.

# Enhanced 'for' Loop 2-2



- ◆ Following table shows the method for retrieving elements from an array object using enhanced `for` loop and its equivalent `for` loop:

for Loop	Enhanced for Loop
<pre>type var; for (int i = 0; i &lt; arr. length; i++) { var = arr[i]; . . . }</pre>	<pre>for (type var : arr) { . . . // Body of the loop . . . }</pre>



# Nested Loop 1-3



- ◆ The placing of a loop statement inside the body of another loop statement is called nesting of loops.
- ◆ There can be any number of combinations between the three types of loops.
- ◆ The most commonly nested loops are formed by `for` statements which can be nested within another `for` loop forming nested-for loop.
- ◆ Following code snippet demonstrates the use of a nested-for loop for displaying a pattern:

```
public class DisplayPattern {  
  
    /**  
     * @param args the command line arguments  
     */  
  
    public static void main(String[] args) {  
        int row, col;  
  
        // The outer for loop executes 5 times  
  
        for (row = 1; row <= 5; row++) {
```

## Nested Loop 2-3



```
/*
 * For each iteration, the inner for loop will execute from col = 1
 * and will continue, till the value of col is less than or equal to row
 */
    for (col = 1; col <= row; col++) {
        System.out.print(" * ");
    } // End of inner for loop

    System.out.println();
} // End of outer for loop
}
```

- ◆ The outer for loop starts with the counter variable **row** whose initial value is set to 1.
- ◆ As the condition, **row < 5** is evaluated to `true`, the body of the outer for loop gets executed.
- ◆ The body contains an inner for loop which starts with the counter variable's value **col** set to 1.





- ◆ Following table lists the differences between `while/for` and `do-while` loops:

<b>while/for</b>	<b>do-while</b>
Loop is pre-tested. The condition is checked before the statements within the loop are executed.	Loop is post-tested. The condition is checked after the statements within the loop are executed.
The loop does not get executed if the condition is not satisfied at the beginning.	The loop gets executed at least once even if the condition is not satisfied at the beginning.



**Java provides two keywords: `break` and `continue` that are used within loops to change the flow of control based on conditions.**

# 'break' Statement 1-3



- ◆ It can be used to terminate a `case` in the `switch` statement.
- ◆ It forces immediate termination of a loop, bypassing the loop's normal conditional test.
- ◆ When the `break` statement is encountered inside a loop, the loop is immediately terminated and the program control is passed to the statement following the loop.
- ◆ If used within a set of nested loops, the `break` statement will terminate the innermost loop.
- ◆ Following code snippet demonstrates the use of `break` statement:

```
public class AcceptNumbers {  
  
    /**  
     * @param args the command line arguments  
     */  
  
    public static void main(String[] args) {  
        int cnt, number; // cnt variable is a counter variable
```

## 'break' Statement 2-3



```
for (cnt = 1, number = 0; cnt <= 10; cnt++) {  
    // Scanner class is used to accept data from the keyboard  
    Scanner input = new Scanner(System.in);  
    System.out.println("Enter a number: ");  
    number = input.nextInt();  
  
    if (number == 0) {  
        // break statement terminates the loop  
        break;  
    } // End if statement  
} // End of for statement  
}
```

- ◆ In the code, the user is prompted to enter a number, and this is stored in the variable, **number**.
- ◆ However, if the user enters the number `zero`, the loop terminates and the control is passed to the next statement after the loop.

## 'break' Statement 3-3



- ◆ Following figure shows the output of the code:

```
run:  
Enter a number:  
8  
Enter a number:  
6  
Enter a number:  
3  
Enter a number:  
0  
BUILD SUCCESSFUL (total time: 8 seconds)
```



# 'continue' Statement 1-2



- ◆ It skips statements within a loop and proceeds to the next iteration of the loop.
- ◆ In `while` and `do-while` loops, a `continue` statement transfers the control to the conditional expression which controls the loop.
- ◆ Following code snippet demonstrates the code that uses `continue` statement in printing the square and cube root of a number:

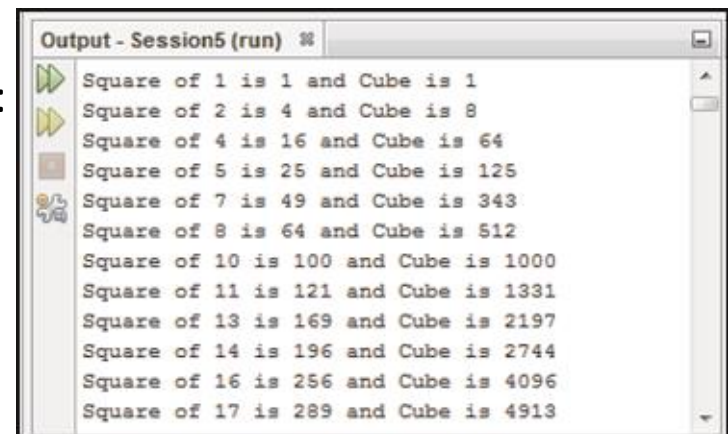
```
public class NumberRoot {  
    /**  
     * @param args the command line arguments  
     */  
  
    public static void main(String[] args) {  
        int cnt, square, cube;  
        // Loop continues till the remainder of the division is 0  
  
        for (cnt = 1; cnt < 300; cnt++) {  
            if (cnt % 3 == 0) {  
                continue;  
            }  
        }  
    }  
}
```

# 'continue' Statement 2-2



```
        square = cnt * cnt;
        cube = cnt * cnt * cnt;
        System.out.printf("\nSquare of %d is %d and Cube is %d", cnt, square,
cube);
    } // End of the for loop
}
```

- ◆ The code declares a variable **cnt** and uses the **for** statement which contains the initialization, termination, and increment expression.
- ◆ The value of **cnt** is divided by 3 and the remainder is checked.
- ◆ If the remainder is 0, the **continue** statement is used to skip the rest of the statements in the body of the loop.
- ◆ If remainder is not 0, the **if** statement evaluates to **false**, and the square and cube of **cnt** is calculated and displayed.
- ◆ Following figure shows the output of the code:





- ◆ Java defines an expanded form of `break` and `continue` statements referred to as labeled statements.
- ◆ **Labeled Statements:**
  - ◆ Are expanded forms that can be used within any block that must be part of a loop or a switch statement.
  - ◆ Can be used to precisely specify the point from which the execution should resume.
  - ◆ Can be used to exit from a set of nested blocks.
- ◆ The syntax to declare the labeled `break` statement is as follows:

## Syntax

```
break label;
```

Where,

`label`: Is an identifier specified to put a name to a block. It can be any valid Java identifier followed by a colon.

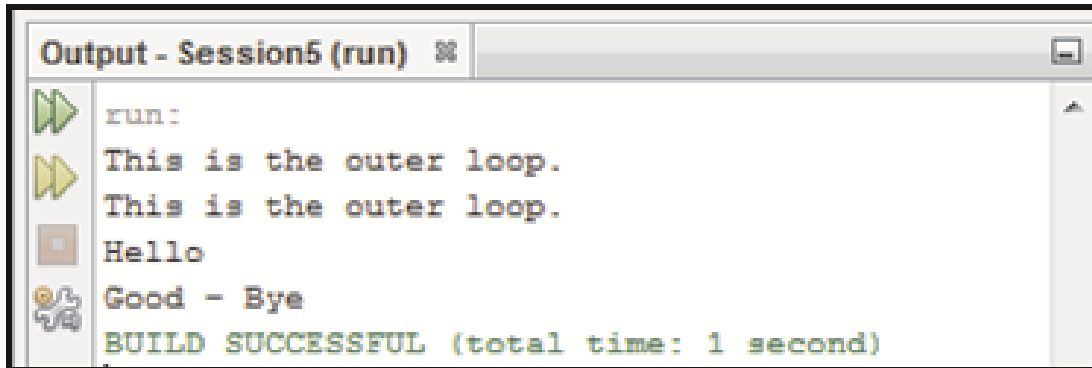


- ◆ Following code snippet demonstrates the use of labeled `break` statement:

```
public class TestLabeledBreak {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        int i;  
  
        outer:  
        for (i = 0; i < 5; i++) {  
            if (i == 2) {  
                System.out.println("Hello");  
                // Break out of outer loop  
                break outer; }  
            System.out.println("This is the outer loop.");  
        }  
  
        System.out.println("Good - Bye");  
    }  
}
```



- ◆ In the code, the loop will execute for five times.
- ◆ The first two times it displays the sentence 'This is the outer loop'.
- ◆ In the third round of iteration the value of **i** is set to 2 and prints 'Hello'.
- ◆ Next, the `break` statement is encountered and the control passes to the label named **outer:**.
- ◆ Thus, the loop terminates and the last statement is printed.
- ◆ Following figure shows the output of the code:



```
run:
This is the outer loop.
This is the outer loop.
Hello
Good - Bye
BUILD SUCCESSFUL (total time: 1 second)
```



- ◆ **Labeled continue Statement:**

- ◆ Similar to labeled break statement, you can specify a label to enclose a loop that continues with the next iteration of the loop.
- ◆ This is done using labeled continue statement.

- ◆ Following code snippet demonstrates the use of labeled continue statement:

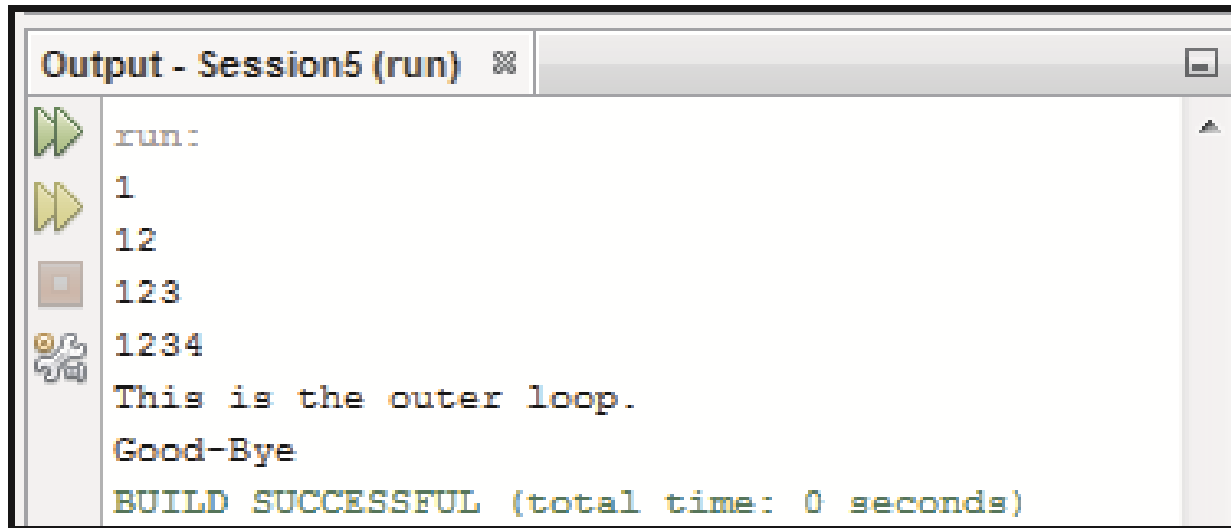
```
public class NumberPyramid {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
  
        outer:  
        for (int i = 1; i < 5; i++) {  
            for (int j = 1; j < 5; j++) {  
                if (j > i) {  
                    System.out.println();  
                    /* Terminates the loop counting j and continues the  
                     * next iteration of the loop counting I */  
                    continue outer;  
                } // End of if statement  
            }  
        }  
    }  
}
```

# Labeled Statements 5-5



```
        System.out.print(j);  
    } // End of inner for loop  
    System.out.println("\nThis is the outer loop.");  
} //End of outer for loop  
System.out.println("Good-Bye");  
}  
}
```

- ◆ Following figure shows the output of the code:



```
Output - Session5 (run) ✖  
run:  
1  
12  
123  
1234  
This is the outer loop.  
Good-Bye  
BUILD SUCCESSFUL (total time: 0 seconds)
```



- ◆ Loops enable programmers to develop concise programs, which otherwise would require thousands of lines of program statements.
- ◆ The loop statements supported by Java are namely, while, do-while, and for.
- ◆ The while loop is used to execute a statement or a block of statements until the specified condition is true.
- ◆ The do-while statement checks for condition at the end of the loop rather than at the beginning to ensure that the loop is executed at least once.
- ◆ The for loop is especially used when the user knows the number of times the statements need to be executed in the code block of the loop. The three parts of for statement are initialization, condition, increment/decrement.
- ◆ The placing of a loop in the body of another loop is called nesting.
- ◆ Java provides two keywords namely, break and continue that serve diverse purposes. However, both are used with loops to change the flow of control.