Chapter Table of Contents

Chapter 1.2

Operators and Looping Constructs

Aim

To provide the students, knowledge of operators and looping constructs Java programming

Instructional Objectives

After completing this chapter, you should be able to:

- Illustrate various operators in Java
- Explain operator precedence
- Demonstrate different control statements in Java

Learning Outcomes

At the end of this chapter, you are expected to:

- Describe the role of operators in java with example
- Apply operator precedence in an arithmetic expression by Converting an infix expression to postfix expression
- Develop programs to demonstrate different operators and control statements in Java
- Differentiate between break and continue statement

1.2.1 Operators in Java

Operators in Java are used to perform arithmetic operations. They comprises of rich set of operators to manipulate variables. Following are some of the operators:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators

(i) Relational Operator

The operands of the arithmetic operators are of a numeric type because you cannot use them on Boolean types. However, you can use them on char types since the char type in Java is, essentially, a subset of int. Arithmetic Operators:

Arithmetic operators are operators used to perform mathematical expression. It takes numerical expression as their operands and return a single value as a result. Following are the standard arithmetic operators.

Operators	Symbol
Addition	+
Subtraction(also unary minus)	-
Multiplication	*
Division	/
Modulus	%
Increment	++
Addition assignment	+=
Multiplication assignment	*=
Subtraction assignment	-=
Division assignment	/=
Modulus assignment	%=
Decrement	- -

Table 1.2.1. Arithmetic Operators

Syntax:

Operand 1 + Operand 2

Subtraction:

Provide difference by subtracting two operands

Syntax:

Operand 1 – Operand 2

Multiplication:

Product of two operands can be yielded using multiplication operators.

Syntax:

Operand 1 * Operand 2

Division:

Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor.

Syntax:

Operand1 / Operand 2

Operators and their uses:

Operators	Syntax	Uses
Addition	Operand 1 + operand 2	Sum two operands
Subtraction	Operand 1 – Operand 2	Provides difference of operands
Multiplication	Operand 1 * Operand 2	Product of two operands can be yielded
Division	Operand 1 / Operand 2	Produces the quotient of operands

Table 1.2.2: Operators and their Uses

(ii) Bitwise Operator

Bitwise operators - Binary operators treat their operands as a sequence of bits (zeroes and ones), rather than as decimal, hexadecimal, or octal numbers.

For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

Java defines some bitwise operators that can be implemented to the integer types, long, int, short, char and byte. These operators act upon the individual bits of their operands.

(iii) Relational Operator

In Java, relational operators are used to check relation between two variables or numbers. They are also called as comparison operators since the outcome will be of true or false (Boolean) values. Relational operators are commonly used in conditional statement (if statement / looping statement) in order to check the conditions (true or false).

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Table: 1.2.3: Relational Operator

Sample program

The result of these operations is a Boolean value. The relational operators are most commonly used in the definitions that control the 'if' statement and the multiple loop statements. Any type in Java, containing integers, floating-point numbers, characters and Booleans can be compared using the inequality test, ==, != equality test. Notice that in Java equality is denoted with two equal signs, not one. (Remember: a single equal sign is the distribution operator.) Only numeric types can be compared using the ordering operators. Means, only integer, floating-point and character operands may be compared to see which is greater or less than the other. As stated, the result produced by a relational operator is a Boolean value.

For example,

```
class Relational {
  public static void main(String[] args){
    int value1 = 1;
    int value2 = 2;
    if(value1 == value2)
      System.out.println("value1 == value2");
    if(value1 != value2)
      System.out.println("value1 != value2");
    if(value1 > value2)
      System.out.println("value1 > value2");
    if(value1 < value2)
      System.out.println("value1 < value2");
    if(value1 <= value2)
      System.out.println("value1 <= value2");
  }
}
```

Output:

```
value1 != value2
value1 <  value2
value1 <= value2
```

(iv) Boolean Logical Operation

The Boolean logical operators displayed here operate only on Boolean operands. Each of the binary logical operators combines two Boolean values to determine a resultant Boolean value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
= =	Equal to
!=	Not equal to
?:	Ternary if-then-else

Table 1.2.4: Boolean Logical

The Logical Boolean operators, &, | and, ^, ! Operate on Boolean conditions in the similar way that operators work on the bits of an integer. The logical! Operator inverts the Boolean state:

!true == false and! False == true.

The following table shows the outcome of each logical operation:

A	B	A|B	A&B	A^B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Table 1.2.5: Outcome of each Logical Operation

Sample Program

```
class BooleanLogic
{
   public static void main(String args[ ])
   {
      boolean A = true;
      boolean B = false;         // these are boolean variables
      System.out.println ("A|B = "+(A|B));
      System.out.println ("A&B = "+(A&B));
      System.out.println ("!A = "+(!A));
      System.out.println ("A^B = "+(A^B));
      System.out.println ("(A|B)&A = "+((A|B)&A));
      System.out.println ("(A&&B)&A = "+((A&&B)&A));
      System.out.println ("(A==B)&A = "+((A==B)&A));
   }
}
```

Here is the Output of this program:

```
A|B=true

A&B=false

!A=false

A^B=true

(A|B)&A=true

(A&&B)&A=false

(A==B)&A=false
```

(v) Assignment Operators

The assignment operator is the individual equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:

var = expression;

Here, the type of var must be compatible with the type of expression.

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. ***For example,*** consider this fragment:

```
int i, j, k;
i = j = k = 200; // set i, j and k to 200
```

This fragment sets the variables i, j and k to 200 using a single statement. It works because the = is an operator that yields the value of the right-hand expression. Thus, the value of k = 200 is 200, which is then assigned to j, which in turn is assigned to i. Using a “chain of assignment” is a simple procedure to set a group of variables to a standard value.

(vi) Operator Precedence

Precedence by name implies the order of code execution. ***For example,*** addition and subtraction has lower precedence than multiplication and division.

Explicit parenthesis are used to override the precedence. Operator precedence in Java can be classified as

- Precedence order
- Associativity
- Precedence and Associativity of Java

Precedence Order:

Operator with higher precedence goes first, when two operators share their operand.

For example,

1+4/2 is treated as 1+ (4 / 2)

Associativity:

Associativity means the order in evaluating the expression with similar precedence.

Let’s see an example of how associativity works:

a = b = c = 15 can be written as a = (b = (c = 15))

This happens because, = operator has right-to-left associativity and an assignment statement always evaluates to the value on the right hand side.

Precedence and Associativity of Java:

Let’s have a look on the table below which shows all Java operators from highest to lowest precedence, along with their associativity.

Notify that the first row shows items that you may not usually think of as operators: parentheses, square brackets and the dot operator. Technically, these are called separators, but they act like operators in an expression. Parentheses are used to alter the precedence of an operation. As you know from the previous chapter, the square brackets give array indexing.

Operator	Description	Associativity
[ ] . ( ) ++ --	Access array element Access object member Invoke a method Post increment Post decrement	**Left to Right**
+ + - - + - ! ~	Pre – Increment Pre – Decrement Unary plus Unary minus Logical NOT Bitwise NOT	**Right to Left**
* / %	Multiplicative	**Left to Right**
< < = > > =	Relational Type Conversion	**Left to Right**
= = ! =	Equality	**Left to Right**
& ^ | && || ?:	Bitwise AND Bitwise XOR Bitwise OR Conditional AND Conditional OR Conditional	**Left to Right**

Table 1.2.6: Precedence and Associativity of Java

Self-assessment Questions

1) The highest order precedence operator(s) in Java is/are _______________.

   a) ( ) b) { }

   c) Both a and b d) [ ]

2) In Java which of the following assignment operators are does not exist?

   a) % b) >>>

   c) <<< d) >>

3) The output of a relational operator is a Boolean value.

   a) True b) False

1.2.2 Control Statements

In Java, control statements are used to control the order of execution of the program. They are split up into **three** categories:

1. Selection statement
2. Iteration statement
3. Jump statement

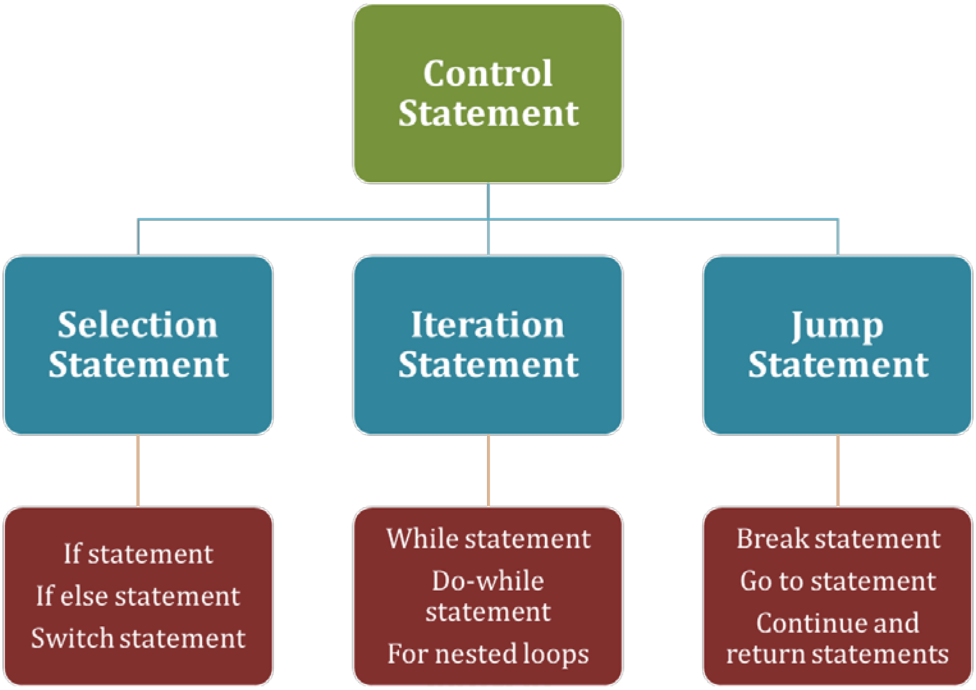

Figure 1.2.1: Control Statements

(i) Selection Statement

In Java, selection statements are used to

- Find different path of execution based on the outcome of an expression or state of a variable.
- Control to the execution of the program.

If statement

If statements also called as conditional statement. If statement executes the statements associated with it only if the specified condition is true. Else it will skip the execution and continue with the rest of program, if else keyword is specified.

Sample program

```
import java.util.Scanner;
public class If
{
   public static void main(String  args[ ])
   {
      int age;
      Scanner inputDevice = new Scanner (System.in);
      System.out.println("Enter Your Age");
      age = inputDevice.nextInt();
      if(age>15)
      System.out.println("above 15");
   }
}
```

Output:

```
Enter Your Age  20

above 15
```

If else statement:

In the preceding section, we used only a simple if statement. If a condition turns out to be true, then it will execute statements in the curly braces. What if that condition was false? The execution will go out of the if block to the next line of code. But, you may want to perform some action or show some message if the condition is false. The Java if-else statement can be used with the if statement when a condition is false.

Syntax for if and else statement:

```
If(condition)
{
   Statement ;  // Code to be executed if condition is true
}
else
{
   Statement ;
}   // Code to be executed if condition is true
```

Sample Program:

```
import java.util.Scanner;
public class Demo1
{
   public static void main( String  args[ ])
   {
      int age;
      Scanner inputDevice = new Scanner (System.in);
      System.out.println (“Enter Age”);
      Age = inputDevice.nextInt ();
      If (age>=15)
      System.out.println (“above 15);
      else
      System.out.println (“below 15”);
   }
}
```

Output:

```
Enter Age 16

Above 15

Enter Age 11

Below 15
```

Switch Statement:

The switch statement is a multi-way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. Such as, it often provides a better alternative than an extensive series of if-else-if statements. Here is the general form of a switch statement:

General form of switch statement (syntax):

```
switch(expression)
{
   case value1:
   //statement sequence
   break;
   case value1:
   //statement sequence
   break;
   .
   .
   .
   case valueN:
   //statement sequence
   break;
   default:
   // default statement sequence
}
```

The expression must be the byte, char, int, or short; each of the values specified in the case statements must be of a type compatible with the expression. Each case value must be an incomparable literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed.

Working of the switch statement: The value of the expression is associated with each of the literal values in the case statements. If a match is detected, the code sequence following that case statement is done. If constants do not match with the value of the expression, then the default statement is done. Though, the default statement is arbitrary. If no case matches and also no default is being, then no more action is taken.

The break statement is applied within the switch statement to end a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement. It has the effect of "jumping out" of the switch.

Here is a simple example that uses a switch statement:

```
// A switch statement example.
class SwitchStatement
{
   public static void main(String args[])
   {
      for(int p=0; p<5; p++)
      switch(p)
      {
        case 0:
        System.out.println("p is zero.");
        break;
        case 1:
        System.out.println("p is one.");
        break;
        case 2:
        System.out.println("p is two.");
        break;
        case 3:
        System.out.println("p is three.");
        break;
        default:
        System.out.println("p is greater than 3.");
      }
   }
}
```

Output for this program:

```
p is zero.

p is one.

p is two.

p is three.

p is greater than 3.

p is greater than 3.
```

As you can see in this above example, each time through the loop, the statements correlated with the case constant that matches p are executed. All others are bypassed. After p is greater than 3, no case statements match, so the default statement is executed.

Nested switch statements:

You can apply a switch as part of the statement series of an external switch. It is called a nested switch. Since a switch statement defines its block, no conflicts arise between the case constants in the internal switch and those in the external switch.

For example, the following

```
the fragment is entirely valid:
switch(count)
{
   case 1:
   switch(target)
   {
      // nested switch
      case 0:
      System.out.println("target is zero");
      break;
      case 1: // no conflicts with the external switch
      System.out.println("target is one");
      break;
   }
   break;
   case 2: // ...
}
```

Here, the case 1: statement in the inside switch does not conflict with the case 1: statement in the external switch. The count variable is only compared with the list of cases at the outer level. If the count is 1, then the target is compared with the inner list cases.

In summary, there are three main features of the switch statement to note:

- The switch differs from the if in that switch can only test for equality, whereas if can evaluate any Boolean expression. That is, the switch looks only for a match between the value of the phrase and one of its case constants.
- No two case connected to the same switch can have the same values. A switch statement and an enclosing without switch statement can have case constants in common.
- A switch statement is usually more efficient than a set of nested ifs.

(ii) Iteration Statement

Java's iteration statements are for "for ", "while " and "do-while ". These statements create what we commonly call loops. As you reasonably know, a loop regularly executes the same set of guidance until a termination condition is met. As you will see, Java has a loop to fit any programming need.

There are three types of iteration statements. They are:

1. while statement
2. do-while statement
3. for statement

While statement:

Let's see what does while statement do in Java programming:

- It uses a Boolean expression to control iteration.
- It executes as long as the Boolean expression is true.
- Set of statements are repeated until the condition for termination is met.

Syntax:

```
while (condition)
{
   //body of the while loop
}
```

Sample program

```
Class Example
{
   public static void main (String  args[ ])
   {
      int count =1;
      While (count<5)
      {
        System.out.println (“count is:”+count);
        count++;
      }
   }
}
```

Output:

```
   1 2 3 4
```

Do-while statement:

As you just noticed, if the conditional expression controlling a while loop is initially invalid, then the body of the loop will not be executed at all. Though, sometimes it is desirable to run the body of a loop at least once, despite if the conditional expression is wrong to begin with. In other words, when you would like to check the termination expression at the end of the loop rather than at the beginning. Fortunately, Java provides a circuit that does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

Its general form is:

Syntax for do-while statement:

```
do
{
   (statements);
}
while (expression);
```

Let's see a reworked version of the “*tick*” program that demonstrates the do-while loop. It generates the same output as before.

```
// Demonstrate the do-while loop.
class DoWhileStatement
{
   public static void main(String args[])
   {
      int i = 7;
      do
      {
        System.out.println("tick " + i);
        i--;
      } while(i > 0);
   }
}
```

The loop in the preceding program, while technically correct, can be written more efficiently as follows:

```
do
{
   System.out.println("tick " + i);
}
while(--i > 0);
```

In this example, the expression (– –i > 0) combines the decrement of i and the test for zero into one expression. Here is how it works. First, the – –i statement executes, decrementing i and returning the new value of i. This value is then compared with zero. If it is greater than zero, the loop continues; otherwise it terminates.

Sample program

Let's see a do-while to process a menu selection:

```
class MenuSelection
{
   public static void main(String args[])
   throws java.io.IOException
   {
     char choice;
     do
     {
        System.out.println("Help on:");
        System.out.println(" 0. if");
        System.out.println(" 1. switch");
        System.out.println(" 2. while");
        System.out.println(" 3. do-while");
        System.out.println(" 4. for\n");
        System.out.println("Choose zero:");
        choice = (char) System.in.read();
     }
     while( choice < '0' || choice > '4');
     System.out.println("\n");
     switch(choice)
     {
        case '1':
        System.out.println("The if:\n");
        System.out.println("if(condition) statement;");
        System.out.println("else statement;");
        break;
        case '2':
        System.out.println("The switch:\n");
        System.out.println("switch(expression) {");
        System.out.println(" case constant:");
        System.out.println(" statement sequence");
        System.out.println(" break;");
        System.out.println(" // ...");
        System.out.println("}");
        break;
        case '3':
        System.out.println("The while:\n");
        System.out.println("while(condition) statement;");
        break;
        case '4':
        System.out.println("The do-while:\n");
        System.out.println("do {");
        System.out.println(" statement;");
        System.out.println("} while (condition);");
        break;
        case '5':
        System.out.println("The for:\n");
        System.out.print("for(init; condition; iteration)");
        System.out.println(" statement;");
        break;
     }
   }
}
```

Here you can see a sample run produced by this above program:

Help on:

1. if
2. switch
3. while
4. do-while
5. for

Choose one: 4

The do-while:

```
do
{
   statement;
} while (condition);
```

In the program, the do-while loop is used to verify that the user has entered a valid choice. If not, then the user is prompted. The menu should be produced at least once during execution, the while is the perfect loop to accomplish this in this statement.

For statement

Starting with JDK 5, there are two forms of the "for" loop. The first one is the old form that has been in use since the original version of Java. The second is the new “for-each” form. Both types of for loops are explained here, starting with the old form.

Here is the general form of the traditional for statement:

Syntax:

```
for (initialisation; condition; increment/decrement)
{
   Statements;
}
```

If only one statement is being renewed, there is no need for the curly braces. The for loop operates as follows. When the loop first starts, the initialisation division of the loop is executed. It is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialisation expression is only executed once. Next, the condition is evaluated. Be a Boolean expression. It tests the loop control variable upon a destination value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.

Next, the emphasis portion of the loop is executed. It is an expression that increments or decrements the loop switch variable. The loop then iterates, first judging the conditional expression, then performing the body of the loop and then executing the emphasis expression with each pass. This process repeats continuously until the controlling expression is false.

Let's see a version of the “tick” program that uses a for loop:

```
// Demonstrate the for loop.
class ForTickLoop
{
   public static void main(String args[])
   {
     int n;
     for(i=9; i>0; i--)
     System.out.println("tick " + i);
   }
}
```

Sample program:

```
Class Demo
{
  Public static void main (string  args[ ])
  {
    for (int i=1; i<3; i++)
    {
      Sytem.out.println (“count is:” + i);
    }
  }
}
```

Output:

```
count is: 1

count is: 2
```

Nested Loops:

Nested loops are very helpful in processing information. It can be described as, if one loop contains another then the second loop is said to be nested inside the first.

In Java programming, any number of loops can be nested.

Following section shows the syntax of nested loops.

Syntax - nested loop statement

```
for(initialisation; test condition; increment/decrement)
{
   statements;
   while(expression)
   {
      statements;
      |
      |
      |
      do
      {
         statements;
      }while(expression);
   }
}
```

Syntax – nested for statement

```
for(initialisation; test condition; increment/decrement)
{
  statements;
  for(initialisation; test condition; increment/decrement)
  {
    statements;
    |
    |
    |
    for(initialisation; testcondition; increment/decrement)
    {
      statements;
    }
  }
}
```

Syntax – nested while

```
While(expression)
{
  statements;
  |
  |
  |
  While(expression)
  {
    statements;
  }
}
```

Sample Program

```
public class Nested
{
  Public static void main (string  args[ ])
  {
    for (int i=1; i<=5; i++)
    {
      System.out.println (“”);
      for (int j=1; j<=i; j++)
      {
       System.out.println (j);
      }
    }
    System.out.println (“”);
  }
}
```

Output:

```
1

12

123

1234

12345
```

(iii) Jump Statements

Java supports three jump statements: ***break, continue and return***. Certain statements shift control to another part of the program. Each is examined here.

1. Break
2. Continue
3. Return

Break Statement:

In Java Language, the break statement has three uses. First, as you have seen, it terminates a state In Java, the break statement has three purposes. First, as you have seen, it ends a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be utilised as a "civilised" form of goto. One statements are explained here.

Break statements are used to:

- Terminate the statement sequence.
- Transfer controls to other parts of the program.
- Exit from a loop, if any specified conditions evaluates to true, then the break statement can be used to terminate the loop instantly.

Syntax for break statement:

```
break;
```

Using break to exit a loop:

Let's see an ***example*** that how break exit from a loop:

```
// Using break to exit a loop.
class BreakLoopExit
{
  public static void main(String args[])
  {
    for(int p=0; p<50; p++)
    {
      if(p == 10) break; // terminate loop if p is 10
      System.out.println("p: " + p);
    }
    System.out.println("Loop complete.");
  }
}
```

The above program generates the following output:

```
p: 0

p: 1

p: 2

p: 3

p: 4

p: 5

p: 6

p: 7

p: 8

p: 9

Loop complete.
```

Now you can see, though the for loop is designed to run from 0 to 49, the break statement causes it to ends early when p equals 10(0 to 9).

Goto Statement

Goto statement is used to transfer the control to the user desired location. This statement refer the label in the same function only.

Syntax:

goto label;

Sample program

```
import java.util.Scanner;
class Demoo
{
  public static void main(String args[])
  {
    int num, i,sum=0;
    go:
    {
      Scanner data = new Scanner(System.in);
      System.out.println("Enter a number :");
      num=data.nextInt();
      for(i=0;i<50;i++)
      {
       sum=sum+i;
       if(i==num)
       break go;
      }
    }
    System.out.println("Sum of odd number:"+sum);
  }
}
```

Here is the output for this above example:

```
Enter a number: 35

Sum of odd number: 630
```

It means goto statement is used to transfer the control to the user defined value location. When will user enter the random value between 0 to 49, it will show you only the sum of odd numbers. This statement refers the label in the same function only.

Continue Statement:

Continue statements are used

- Inside the bounds of loop statement itself.
- To continue the conditional statements until it satisfies the specified condition.
- To skip the rest of the statements in the body of the loop and continue with the next iteration of the loop.

Syntax

continue;

Sample program:

```
public class Demo3
{
  public static void main (String args[ ])
  {
    int [ ] numbers = {5,10,15,20,25,30,35,40,45,50};
    for ( int x: numbers)
    {
      if(x= = 40)
      {
        Continue;
      }
      System.out.println (x);
      System.out.println (“\n”);
    }
  }
}
```

Output:

```
5

10

15

20

25

30

35

45

50
```

Break statement	Continue statement
A **break** statement results in the termination of the statement to which it applies (Switch, for, do, or while).	A **continue** statement is used to end the current loop iteration and return control to the loop statement.
The break statement results in the termination of the loop, it will come out of the loop and stops further iterations.	The continue statement stops the current execution of the iteration and proceeds to the next iteration. The return statement takes you out of the method. It stops executing the method and returns from the method execution.
A break statement when applied to a loop ends the statement.	A continue statement ends the iteration of the current loop and returns the control to the loop statement.
If the break keyword is followed by an identifier that is the label of a random enclosing statement, execution transfers out of that enclosing statement.	If they continue keyword is followed by an identifier that is the label of an enclosing loop, execution skips to the end of that loop instead.
a break statement discontinues the execution of the loop and gets the control out of the loop after meeting the condition	On the other hand the continue statement on meeting the condition, goes to the beginning of the loop and re-executes it.

Table 1.2.7: Difference between Break and Continue Statements

Return Statement:

In Java return statements are used to:

- Return from a method.
- Transfer back the control to the caller of the method.
- Terminate the method in which it is performing currently.

Syntax

```
return value;
```

Sample Program

```
public int mul(int a, int b)
{
   int c=a*b;
   return c;
}
public static void main( String args[ ])
{
   sample obj= new sample();
   int x=obj.mul(4,3);
   System.out.println(x);
}
```

Output

```
12
```

Self-assessment Questions

6) Switch statements cannot have identical values in two cases.

   a) True b) False

7) Identify loop statement from the following

   a) If statement b) Switch statement

   c) If else d) For statement

8) _____________is a statement written in sequence and repeated until a specified condition is satisfied.

   a) Format b) Case

   c) Loop d) Condition

Summary

- Operators and looping constructs directs the user how to perform, manipulate and control Java programs in its environment.
- Operators in Java are used to perform arithmetic operations.
- Relational operators are used to check relation between two variables or number.
- Precedence by name implies the order of code execution of a program.
- Selection statements allow the program to choose different ways of execution based upon the result of an expression or the state of a variable.
- Iteration statements allow program execution to repeat multiple statements (that is, iteration statements form loops).Jump statements allow your program to perform in a nonlinear fashion. All of Java control statements are examined here.
- In Java Language, the break statement has three uses. First, as you have seen, it terminates a stateIn Java, the break statement has three purposes. First, as you have seen, it ends a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be utilised as a "civilised" form of goto.
- Java provides a circuit that does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

Terminal Questions

1. Describe the role of operators in Java with an example.
2. Explain control statements and its functions with examples
3. What are loops? Explain how loops controls the program execution.

Answer Keys

Self-assessment Questions	
Question No.	Answer
1	c
2	b
3	a
4	b
5	d
6	c

Activity

Activity Type: Online **Duration**: 30 Minutes

Description:

Create a table which lists all the operators in Java and describes its functions.

Bibliography

e-References

- *The complete reference Java-2* - Herbert Schildt-7th edition- McGraw Hill professional (Schildt, 2016). Retrieved on 12th April 2016, from https://iamgodsom.files.wordpress.com/2014/08/java-the-complete-reference-7th-edition.pdf

- Schildt, H. (2016). *The complete reference Java* (7th ed.). Retrieved on 12th April 2016, from http://iiti.ac.in/people/~tanimad/JavaTheCompleteReference.pdf

Video Links

Topic	Link
Operators	http://www.freejavaguide.com/boolean_operators.htm www.youtube.com/watch?v=lKGQU-d1Y6E
Selection statement	http://www.javasimplified.com/selection-statements-in-java/
Arrays	https://www.youtube.com/watch?v=o4VzblFTwII
Control statement	https://www.youtube.com/watch?v=7x4u8ItJlCA

Notes: