MODULE 2

Syllabus: Operators: Arithmetic Operators, The Bitwise Operators, Relational Operators, Boolean Logical Operators, The Assignment Operator, The ? Operator, Operator Precedence, Using Parentheses.
Control Statements: Java's Selection Statements, Iteration Statements, Jump Statements.

Operators

Java provides rich set of operators, mainly divided into four groups viz. arithmetic, bitwise, relational and logical. These operators are discussed here.

2.1 Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is a subset of **int**.

Note down following few points about various operators:

- Basic arithmetic operators like +, -, * and / behave as expected for numeric data.
- The – symbol can be used as unary operator to negate a variable.
- If / is operated on two integer operands, then we will get only integral part of the result by truncating the fractional part.
- The % operator returns the remainder after division. It can be applied on integer and floating-point types. For example,

```
int x=57;
double y= 32.8;
System.out.println("on integer " + x%10);    //prints 7
System.out.println("on double " + y%10);     //prints 2.8
```

- Compound assignment operators like += will perform arithmetic operation with assignment. That is,

```
a+=2;    ⟹    a=a+2;
```

- Increment/decrement operators (++ and --) will increase/decrease the operand by 1. That is,

```
a++;    ⟹    a=a+1;
b--;    ⟹    b=b-1;
```

- The ++ and -- operators can be used either as pre-increment/decrement or post-increment/decrement operator. For example,

```
x= 5;
y=x++;          //post increment
```

  Now, value of x (that is 5) is assigned to y first, and x is then incremented to become 6.

```
x= 5;
y=++x;          //pre-increment
```

  Now, x is incremented to 6 and then 6 is assigned to y.

NOTE that in C/C++, the % operator cannot be used on float or double and should be used only on integer variable.

2.2 Bitwise Operators

Java defines several *bitwise operators* that can be applied to **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td>~</td><td>Bitwise unary NOT</td></tr>
<tr><td>&</td><td>Bitwise AND</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td>^</td><td>Bitwise exclusive OR</td></tr>
<tr><td>>></td><td>Shift right</td></tr>
<tr><td>>>></td><td>Shift right zero fill</td></tr>
<tr><td><<</td><td>Shift left</td></tr>
<tr><td>&=</td><td>Bitwise AND assignment</td></tr>
<tr><td>|=</td><td>Bitwise OR assignment</td></tr>
<tr><td>^=</td><td>Bitwise exclusive OR assignment</td></tr>
<tr><td>>>=</td><td>Shift right assignment</td></tr>
<tr><td>>>>=</td><td>Shift right zero fill assignment</td></tr>
<tr><td><<=</td><td>Shift left assignment</td></tr>
</table>

Since bitwise operators manipulate the bits within the integer, let us first understand the bit-representation of integer data in Java.

All of the integer types are represented by binary numbers of varying bit widths. For example, the **byte** value for 42 in binary is 00101010, where each position represents a power of two, starting with 2^0 at the rightmost bit. All of the integer types are signed integers. Java uses an encoding known as *two's complement,* which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result. For example, –42 is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or –42. To decode a negative number, first invert all of the bits, and then add 1. For example, –42, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42.

Bitwise Logical Operators

The bitwise logical operators are &, |, ^ and ~. Following table shows the result of each operation.

A	B	A&B	A|B	A^B	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Bitwise NOT

A unary NOT operator ~, also called as ***bitwise complement*** inverts all the bits of the operand. For example, the number 42, which has the following bit pattern: 00101010 becomes 11010101 after the NOT operator is applied.

Bitwise AND

As the name suggests, initially, operands are converted into binary-format. Then, the AND (&) operation is performed on the corresponding bits of operands. Consider an example –

```
int x=5, y=6,z;
z= x & y;
```

Now, this operation is carried out as –

```
x ⇨       0000 0101
y ⇨  &    0000 0110
          ---------
z ⇨       0000 0100
```

Thus, z will be decimal equivalent of 0000 0100, which is 4.

Bitwise OR

Here, the OR (|) operations is performed on individual bit of operands. For example –

```
int x=5, y=6,z;
z= x | y;
```

Now, this operation is carried out as –

```
x ⇨       0000 0101
y ⇨  |    0000 0110
          ---------
z ⇨       0000 0111
```

Thus, z will be decimal equivalent of 0000 0111, which is 7.

Bitwise XOR

In XOR operation, if both bits are same (either both are 1 or both 0), then the resulting bit will be 0 (false). Otherwise, the resulting bit is 1 (true). For example –

```
int x=5, y=6,z;
z= x ^ y;
```

Now, this operation is carried out as –

```
x  ⇨      0000 0101
y  ⇨  ^   0000 0110
          ---------
z  ⇨      0000 0011
```

Thus, z will be decimal equivalent of 0000 0011, which is 3.

Left Shift

The left shift operator, **<<**, shifts all of the bits in a value to the left by a specified number of times. It has this general form:

value << num

For each shift, one higher order bit is shifted out (or lost) and extra zero is appended as the lower order bit. Thus, for int, after 31 shifts, all the bits will be lost and result will be 0, whereas for long, after 63 shifts, all bits will be lost.

Java's automatic type promotions produce unexpected results when you are shifting **byte** and **short** values. As you know, **byte** and **short** values are promoted to **int** when an expression is evaluated. Furthermore, the result of such an expression is also an **int**. This means that the outcome of a left shift on a **byte** or **short** value will be an **int**, and the bits shifted left will not be lost until they shifted for 31 times. To avoid this problem, we should use type-casting as shown in the following example.

Program 2.1: Demonstration of left-shift operator

```
class ShiftDemo
{
      public static void main(String args[])
      {
            byte a = 64, b;
            int i;
            i = a << 2;
            b = (byte) (a << 2);
            System.out.println("Original value of a: " + a);
            System.out.println("i and b: " + i + " " + b);
      }
}
```

The result would be –

Original value of a: 64
i and b: 256   0

Since **a** is promoted to **int** for evaluation, left-shifting the value 64 (0100 0000) twice results in **i** containing the value 256 (1 0000 0000). However, the value in **b** contains 0 because after the shift, the low-order byte is now zero.

Each left shift can be thought of as multiplying the number by 2. But, one should be careful because once the number crosses its range during left shift, it will become negative. Consider an illustration –

Program 2.2

```
class ShiftDemo1
 {
```

```
        public static void main(String args[])
        {
             int i;
             int num = 0xFFFFFFE;
             for(i=0; i<4; i++)
             {
                  num = num << 1;
                  System.out.println(num);
             }
        }
   }
```

The output would be –

```
536870908
1073741816   //twice the previous value
2147483632   //twice the previous value
-32          //crosses the range of int and hence negative
```

Right Shift

The right shift operator, **>>** shifts all of the bits in a value to the right by a specified number of times. It has this general form:

value >> num

For each shift, one lower order bit is shifted out (or lost) and extra zero is appended as the higher order bit. For example,

```
int a = 35;          //00100011 is the binary equivalent
a = a >> 2;          // now, a contains 8
```

Each right shift can be thought of as dividing the number by 2. When you are shifting right, the top (leftmost) bit is filled with the previous content of the top bit. This is called ***sign extension*** and is needed to preserve the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4, which, in binary, is

```
11111000     (–8)
>>1
11111100     (–4)
```

Unsigned Right Shift

We have seen that right shift always fills the highest order bit with the previous content of the top bit. But when we are using shift operation on non-numeric data, sign-bit has no significance. To ignore the sign-bit, we will go for unsigned right shift. The following code fragment demonstrates the **>>>**. Here, **a** is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets **a** to 255.

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

```
11111111 11111111 11111111 11111111        –1 in binary as an int
>>>24
00000000 00000000 00000000 11111111        255 in binary as an int
```

Bitwise Operator Compound Assignment

We can use compound assignment even with bitwise operators. That is,

a<<=2; implies a=a<<2;
a^=3; implies a=a^3; and so on.

2.3 Relational Operators

The relational operators determine the relationship between two operands. Specifically, they determine equality and ordering among operands. Following table lists the relational operators supported by Java.

Operator	Meaning
==	Equal to (or comparison)
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value. Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, **==**, and the inequality test, **!=**. Only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is greater or less than the other. For example, the following code fragment is perfectly valid:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

In this case, the result of **a<b** (which is **false**) is stored in **c**.

Note that in C/C++ we can have following type of statement –

```
int flag;
…….
if(flag)
    //do something
```

In C/C++, ***true*** is any non-zero number and ***false*** is zero. But in Java, ***true*** and ***false*** are Boolean values and nothing to do with zero or non-zero. Hence, the above set of statements **will cause an error** in Java. We should write –

```
int flag;
………..
if(flag==1)
    //do some thing
```

2.4 Boolean Logical Operators

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Meaning
&	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

The truth table is given below for few operations:

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

Program 2.3 Demonstration of Boolean Logical operators

```
class BoolLogic
{
      public static void main(String args[])
      {
            boolean a = true;
            boolean b = false;
            boolean c = a | b;
            boolean d = a & b;
            boolean e = a ^ b;
            boolean f = (!a & b) | (a & !b);
            boolean g = !a;

            System.out.println(" a = " + a);
            System.out.println(" b = " + b);
            System.out.println(" a|b = " + c);
            System.out.println(" a&b = " + d);
            System.out.println(" a^b = " + e);
            System.out.println("!a&b|a&!b = " + f);
            System.out.println(" !a = " + g);

            boolean h = b & (a=!a);

            System.out.println("b & (a=!a) =" +h);
            System.out.println("New a is "+a);
      }
}
```

The output would be –

a = true
b = false
a|b = true
a&b = false

a^b = true
!a&b|a&!b = true
!a = false
b & (a=!a) =false
New a is false

Note: In C/C++, the logical AND/OR operations never evaluates the second operand if the value of first operand itself can judge the result. That is, if the first operand is *false*, then second operand is not evaluated in AND operation and result will be false. Similarly, if the first operand is *true* in OR operation, without evaluating the second operand, it results *true.* But in Java, Boolean logical operators will not act so. Even if the first operand is decisive, the second operand is evaluated. This can be observed in the above program while evaluating h= b& (a= !a). Here, *b* is *false* and hence ANDed with anything results *false*. But, still the second operand (a= !a) is evaluated resulting *a* as false.

If we don't want the second operand to be evaluated, we can use *short-circuit logical* operators.

Short-Circuit Logical Operators

The short-circuit AND (&&) and OR (||) operators will not evaluate the second operand if the first is decisive. For example,

```
int x=0, n=5;
……..
if(x!=0 && n/x > 0)
     //do something
```

Here, the first operand x!= 0 is false. If we use logical AND (&) then the second operand n/x>0 will be evaluated and we will get DivisionByZero Exception. So, to avoid this problem we use && operator which will never evaluated second operand if the first operand results into false.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```
if(c==1 & e++ < 100)
     d = 100;
```

Here, using a single **&** ensures that the increment operation will be applied to **e** whether **c** is equal to 1 or not.

2.5 The Assignment Operator

The *assignment operator* is the single equal sign, **=**. It has this general form:

```
var = expression;
```

Here, the type of *var* must be compatible with the type of *expression.* It allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the **=** is an operator that yields the value of the right-hand expression. Thus, the value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**. Using a "chain of assignment" is an easy way to set a group of variables to a common value.

2.6 The ?: Operator

Java supports *ternary operator* which sometimes can be used as an alternative for *if*-then-*else* statement. The general form is –

```
var = expression1 ? expression2 : expression3;
```

Here, *expression1* is evaluated first and it must return Boolean type. If it results *true*, then value of *expression2* is assigned to *var*, otherwise value of *expression3* is assigned to *var.* For example,

```
int a, b, c ;
……….
c= (a>b)?a:b;        //c will be assigned with biggest among a and b
```

2.7 Operator Precedence

Following table describes the precedence of operators. Though parenthesis, square brackets etc. are separators, they do behave like operators in expressions. Operators at same precedence level will be evaluated from left to right, whichever comes first.

Highest

<table>
<tr><td>(), [ ], .</td></tr>
<tr><td>++, --, ~, !</td></tr>
<tr><td>*, /, %</td></tr>
<tr><td>+, -</td></tr>
<tr><td>>>, >>>, <<</td></tr>
<tr><td>>, >=, <, <=</td></tr>
<tr><td>==, !=</td></tr>
<tr><td>&</td></tr>
<tr><td>^</td></tr>
<tr><td>|</td></tr>
<tr><td>&&</td></tr>
<tr><td>||</td></tr>
<tr><td>?:</td></tr>
<tr><td>=, op=</td></tr>
</table>

Lowest

2.8 Using Parentheses

Parentheses always make the expression within them to execute first. This is necessary sometimes. For example,

```
a= b – c * d;
```

Here, c and d are multiplied first and then the result is subtracted from b. If we want subtraction first, we should use parenthesis like

```
a= (b-c)*d;
```

Sometimes, parenthesis is useful for clarifying the meaning of an expression and for making readers to understand the code. For example,

`a | 4 + c >> b & 7` can be written as `(a | (((4 + c) >> b) & 7))`

In such situations, though parenthesis seems to be redundant, it existence will not reduce the performance of the program.

By: Dr. Chetana Hegde, Associate Professor, RNS Institute of Technology, Bangalore – 98
Email: chetanahegde@ieee.org

2.9 Control Statements

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: **selection, iteration,** and **jump**. *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

2.9.1 Java's Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

if Statement

The general form is –

```
if (condition)
{
	//true block
}
else
{
	//false block
}
```

If the *condition* is true, then the statements written within *true block* will be executed, otherwise *false block* will be executed. The *condition* should result into **Boolean** type. For example,

```
int a, b, max;
…………
if(a>b)
	max=a;
else
	max=b;
```

Nested-if Statement

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. For example,

```
if(i == 10)
{
	if(j < 20)
		a = b;
	if(k > 100)
		c = d;
	else
		a = c;
}
else
	a = d;
```

The if-else-if Statement

The general form is –

```
if(condition1)
      block1;
else if(condition2)
      block2;
…………..
…………..
else
      blockn
```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the block associated with that **if** is executed, and the rest of the ladder is bypassed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed.

switch Statement

The **switch** statement is Java's 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. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```
switch (expression)
{
      case value1:
            // statement sequence
            break;
      case value2:
            // statement sequence
            break;
      …………....
      case valueN:
            // statement sequence
            break;
      default:
            // default statement sequence
}
```

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken. The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of "jumping out" of the **switch**. The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**.

NOTE:

- We can even nest switch statements one within the other.
- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.
- A **switch** statement is usually more efficient than a set of nested **if**s.

The last point is particularly interesting because it gives insight into how the Java compiler works. When it compiles a **switch** statement, the Java compiler will inspect each of the **case** constants and create a "jump table" that it will use for selecting the path of execution depending on the value of the expression. Therefore, if you need to select among a large group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-else**s. The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression. The compiler has no such knowledge of a long list of **if** expressions.

2.9.2 Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops.* A loop repeatedly executes the same set of instructions until a termination condition is met.

while Loop

The general form is –

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

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop.

do- while Loop

The general form is –

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

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.

for Loop

The general form is –

```
for(initialization; condition; updation)
{
    // body of loop
}
```

When the loop first starts, the *initialization* portion of the loop is executed. Generally, this 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 initialization expression is only executed once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *updation* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

for-each Loop

The for-each style of **for** is also referred to as the *enhanced* **for** loop. The general form of the for-each version of the **for** is shown here:

```
for(type itr-var : collection)
      statement-block
```

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by *collection*. There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array. With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained.

Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the base type of the array.

Consider an example –

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++)
      sum += nums[i];
```

The above set of statements can be optimized as follows –

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums)
      sum += x;
```

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on. Not only is the syntax streamlined, but it also prevents boundary errors.

For multi-dimensional arrays:

The for-each version also works for multi-dimensional arrays. Since a 2-d array is an array of 1-d array, the iteration variable must be a reference to 1-d array. In general, when using the for-each **for** to iterate over an array of *N* dimensions, the objects obtained will be arrays of *N*–1 dimensions.

Consider the following example –

Program 2.4 Demonstration of *for-each* version of *for* loop

```
class ForEach
{
	public static void main(String args[])
	{
		int sum = 0;
		int nums[][] = new int[2][3];

		// give nums some values
		for(int i = 0; i < 2; i++)
			for(int j=0; j < 3; j++)
				nums[i][j] = (i+1)*(j+1);

		for(int x[ ] : nums)  //nums is a 2-d array and x is 1-d array
		{
			for(int y : x)  // y refers elements in 1-d array x
			{
				System.out.println("Value is: " +y);
				sum += y;
			}
		}
		System.out.println("Summation: " + sum);
	}
}
```

The output would be –

Value is: 1
Value is: 2
Value is: 3
Value is: 2
Value is: 4
Value is: 6
Summation: 18

The for-each version of **for** has several applications viz. Finding average of numbers, finding minimum and maximum of a set, checking for duplicate entry in an array, searching for an element in unsorted list etc. The following program illustrates the sequential (linear) search.

Program 2.5 Linear/Sequential Search

```
	class SeqSearch
	{
		public static void main(String args[])
		{
			int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
			int val = 5;
			boolean found = false;

			for(int x : nums)
			{
				if(x == val)
				{
```

```
                                found = true;
                                break;
                        }
                }
                if(found)
                        System.out.println("Value found!");
        }
}
```

The output would be –

Value found !

2.9.3 Jump Statements

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.

Using break

In java, ***break*** can be used in 3 different situations:

- To terminate statement sequence in **switch**
- To exit from a loop
- Can be used as a *civilized* version of ***goto***

Following is an example showing terminating a loop using break.

```
for (int i=0;i<20;i++)
        if(i==5)
                break;
        else
                System.out.println(" i= " + i);
```

The above code snippet prints values from 0 to 4 and when i become 5, the loop is terminated.

Using break as a form of goto

Java does not have a goto statement because it is an un-conditional jump and may end up with an infinite loop. But in some situations, goto will be useful. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can, for example, break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be any block. Further, you can specify precisely where execution will resume, because this form of **break** works with a label. As you will see, **break** gives you the benefits of a goto without its problems. The general form of labeled break is:

break label;

Program 2.6 Illustration of *break* statement with labels

```
class Break
{
        public static void main(String args[])
        {
                boolean t = true;

                first:
                {
```

```
                second:
                {
                     third:
                     {
                          System.out.println("Before the break.");
                          if(t)
                             break second;      // break out of second block
                          System.out.println("This won't execute");
                     }
                     System.out.println("This won't execute");
                }
                System.out.println("This is after second block.");
           }
     }
}
```

The output would be –

Before the break
This is after second block

As we can see in the above program, the usage of ***break*** with a label takes the control out of the second block directly.

Using continue

Sometimes, we may need to proceed towards next iteration in the loop by leaving some statements. In such situations, we can use ***continue*** statement within ***for, while*** and ***do-while***. For example –

```
for (int i=1; i<20;i++)
     if (i%2 == 0)
          continue;
     else
          System.out.println("i = " + i);
```

The above code snippet prints only the odd numbers in the range of 1 to 20.

Using return

The ***return*** statement is used to explicitly return the method. Based on some condition, we may need to go back to the calling method sometimes. So, we can use ***return*** in such situations.

QUESTION BANK:

1. What are different types of operators in Java? Explain any two of them.
2. Discuss ternary operator with examples.
3. Differentiate >> and >>> with suitable examples.
4. Briefly explain short-circuit logical operators with examples.
5. Explain different types of iteration statements with examples.
6. Discuss various selective control structures.
7. Write a note on jump statements in Java.
8. Discuss different versions of for - loop with examples.
9. Write a program to illustrate break statement with labels.