JAVA PROGRAMMING

UNIT - II

Dr Yogish H K, Professor, Dept. of ISE, RIT, Bengaluru

UNIT - II

Syllabus

Operators: Arithmetic Operators, The Bitwise Operators, Relational Operators, Boolean Logical Operators, The Assignment Operator, The?: Operator, Operator Precedence.

Control Statements: Java's Selection Statements, if, switch, Iteration Statements, while, do-while, for, the For-Each Version of the for Loop, Nested Loops, Jump Statements, using break, using continue.

Arithmetic Operators

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

- 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, essentially, a subset of int.

The Basic Arithmetic Operators

Addition, Subtraction, multiplication , division and unary -


```
class BasicMath {
    public static void main(String args[]) {
        System.out.println("Integer Arithmetic");
        int a = 1 + 1;
        int b = a * 3;
        int c = b / 4;
        int d = c - a;
        int e = -d;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("e = " + e);
```

The Basic Arithmetic Operators


```
    System.out.println("\nFloating Point Arithmetic");
    double da = 1 + 1;
    double db = da * 3;
    double dc = db / 4;
    double dd = dc - a;
    double de = -dd;
    System.out.println("da = " + da);
    System.out.println("db = " + db);
    System.out.println("dc = " + dc);
    System.out.println("dd = " + dd);
    System.out.println("de = " + de);
  }
}
```

The Basic Arithmetic Operators

When you run this program, you will see the following output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1
Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

The Modulus Operator

- The modulus operator, %, returns the remainder of a division operation.
- It can be applied to floating-point types as well as integer types.

```
class Modulus {
        public static void main(String args[]) {
                int x = 42;
                double y = 42.25;
                System.out.println("x mod 10 = " + x % 10);
                System.out.println("y mod 10 = " + y % 10);
        }
}
```

Output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators

- Operators that can be used to combine an arithmetic operation with an assignment.
  - **a = a + 4;**
- In Java, you can rewrite this statement as shown here:
  - **a += 4;**
- This version uses the += compound assignment operator.
- Both statements perform the same action:
  - they increase the value of **a** by **4**.

Arithmetic Compound Assignment Operators

- There are compound assignment operators for all of the arithmetic, binary operators.
- Thus, any statement of the form:
  - **var = var op expression;**
- can be rewritten as:
  - **var op= expression;**
- The compound assignment operators provide two benefits.
  1. They save you a bit of typing, because they are "shorthand" for their equivalent long forms.
  2. They are implemented more efficiently by the Java run-time system than are their equivalent long forms.

Demonstrate several assignment operators.


```
class OpEquals {
  public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;
        a += 5;
        b *= 4;
        c += a * b;
        c %= 6;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

- The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

Increment and Decrement ++ and --

- The increment operator increases its operand by one. The decrement operator decreases its operand by one.
- For example, this statement:
  - **x = x + 1;**
- can be rewritten like this by use of the increment operator:
  - **x++;**
- Similarly, this statement:
  - **x = x - 1;**
- is equivalent to
  - **x--;**

Prefix and Postfix

- In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.
- In postfix form, the previous value is obtained for use in the expression, and then the operand is modified.
- For example:
  - x = 42;
  - y = ++x;
- In this case, y is set to 43 as you would expect, because the increment occurs before x is assigned to y.
- Thus, the line **y = ++x;** is the equivalent of these two statements:
  - x = x + 1;
  - y = x;
- However, when written like this,
  - x = 42;
  - y = x++;
- the value of x is obtained before the increment operator is executed, so the value of y is 42. Of course, in both cases x is set to 43.
- Here, the line y = x++; is the equivalent of these two statements:
  - y = x;
  - x = x + 1;

Program demonstrates the increment operator


```
class IncDec {
public static void main(String args[]) {
    int a = 1,  b = 2, c, d;
    c = ++b;
    d = a++;
    c++;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
   }
}
```

The output of this program follows:

```
a = 2
b = 3
c = 4
d = 1
```

Relational Operators

- These operators determine the relationship that one operand has to the other.
- The outcome of these operations is a boolean value.
- These operators are most frequently used in the expressions that control the if statement and the various loop statements.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Example

- The result produced by a relational operator is a boolean value.
- 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.

Continue …

- Note: In C/C++, the following statements are very common:
  - int done;
  - if(!done)
  - if(done)
- In Java, above two statements must be written like this:
  - if(done == 0)
  - if(done != 0)
- The reason is that Java does not define true and false in the same way as C/C++. In C/C++, true is any nonzero value and false is zero.
- In Java, **true** and **false** are nonnumeric values that do not relate to zero or nonzero.

Boolean Logical Operators

- operate only on boolean operands. Combine two boolean values to form 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

Boolean Logical Operators ..

- The following table shows the effect 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

Demonstrate the 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);
         }
}
```

OUTPUT:

```
          a = true
          b = false
        a|b = true
        a&b = false
        a^b = true
a&b|a&!b = true
         !a = false
```

Short-Circuit Logical Operators && and ||

- As you can see from the table, the OR operator results in true when A is true, no matter what B is.
- Similarly, the AND operator results in false when A is false, no matter what B is.
- If you use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.

The Assignment Operator

- The assignment operator is the single equal sign, =.
- This 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.
- 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.

The ?: Operator

- Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operator is the ?:.
- The ?: has this general form:
  - **expression1 ? expression2 : expression3**
- Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated.
- The result of the ?: operation is that of the expression evaluated.
- Both expression2 and expression3 are required to return the same type, which can't be void.
- Here is an example of the way that the ?: is:
  - **Num1=5, num2=4;**
  - **large= num1 > num2 ? num1 : num2;**

Demonstrate ?:


```
class Ternary {
    public static void main(String args[]) {
    int i, k;
    i = 10;
    k = i < 0 ? –i : i;
    System.out.println(i + " is " + k);
    i = -10;
    k = i < 0 ? -i : i;
    System.out.println(i + " is " + k);
    }
}
```

The Bitwise Operators

- These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	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

The Bitwise Logical Operators

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

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

Note : the bitwise operators are applied to each individual bit within each operand.

The Bitwise NOT

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

```
35 = 00100011 (In Binary)

// using bitwise complement operator
~ 00100011

_____________
  11011100
```

```
class Main {
  public static void main(String[] args) {
    int number = 35, result;
     result = ~number;
    System.out.println(result);
  }
}
```

The Bitwise AND

- The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases.
- Here is an example:

```
  00101010      42
& 00001111      15
----------
  00001010      10
```

Example 12 & 25


```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

// Bitwise AND Operation of 12 and 25
   00001100
 & 00011001
_____________
   00001000 = 8 (In Decimal)
```

```
class Main {
    public static void main(String[] args) {
    int number1 = 12, number2 = 25, result;
   result = number1 & number2;
   System.out.println(result);    // prints 8
  }
}
```

The Bitwise OR

- The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
  00101010      42
| 00001111      15
----------------
  00101111      47
```

bitwise OR operation of two integers 12 and 25.


```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
   00001100
 | 00011001
_____________
   00011101 = 29 (In Decimal)
```

```
class Main {
  public static void main(String[] args) {
    int number1 = 12, number2 = 25, result;
    result = number1 | number2;
    System.out.println(result);    // prints 29
  }
}
```

The Bitwise XOR

- The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.

```
  00101010      42
^ 00001111      15
----------
  00100101      37
```

bitwise XOR operation of two integers 12 and 25

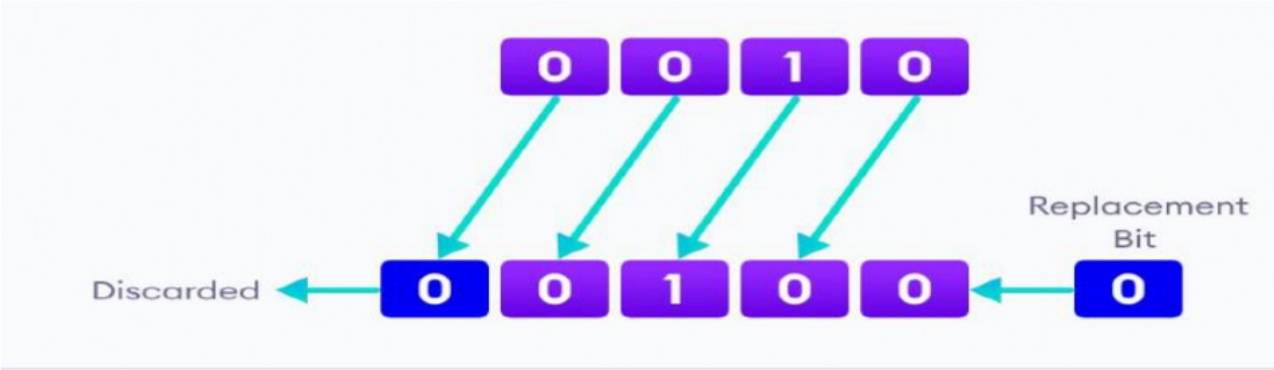

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

// Bitwise XOR Operation of 12 and 25
   00001100
 ^ 00011001
_____________
   00010101 = 21 (In Decimal)
```

```
class Main {
    public static void main(String[] args) {
        int number1 = 12, number2 = 25, result;
        result = number1 ^ number2;
       System.out.println(result);    // prints 21
    }
}
```

Bitwise Left Shift Operator (<<)

- Left shift operator shifts the bits of the number towards **left** a specified number of positions.
- **x<<n**; shift the bits of **x** towards left **n** specified positions. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.
- Example: x=2; x<<1;



Bitwise Left Shift Operator (<<)

- Example : **x=10; x<<2;**
- Shifting the value of x towards the left two positions will make the leftmost 2 bits to be lost.
- Example : **00001010** : 10 (in decimal).
- Becomes **00101000** : 40 (in decimal).

```
class Main {
  public static void main(String[] args) {
      int number = 2;
       // 2 bit left shift operation
      int result = number << 2;
    System.out.println(result);    // prints 8
  }
}
```

Bitwise Right Shift Operator(>>)

- Shifts the bits of the number towards **right ,** specified number of positions.
- **x>>n;** the meaning is to shift the bits of **x** towards the right **n** specified positions.
- **>>** shifts the bits towards the right and also **preserve the sign bit**, which is the leftmost bit.
- The leftmost bit represents the **sign** of the number. The sign bit **0** represents a **positive number**, and **1** represents a **negative number**.
- After shift **>>** on a positive number, we get a **positive value** in the result also. When we perform **>>** on a **negative number**, again we get a **negative value**.

Bitwise Right Shift Operator(>>)

- Example
  - **If x=10; x>>2;**
- Shifting the value of **x** towards the right two positions will make the rightmost 2 bits to be lost.
- Observe the above example, after shifting the bits to the right the binary number

  00001010 (in decimal 10)

  becomes **00000010** (in decimal 2).

```
// right shift of 8
8 = 1000 (In Binary)

// perform 2 bit right shift
8 >> 2:
1000 >> 2 = 0010 (equivalent to 2)
```

```
class Main {
    public static void main(String[] args) {
    int number1 = 8;
   System.out.println(number1 >> 2);    // prints 2
 }
}
```

Bitwise Zero Fill Right Shift Operator (>>>) (Unsigned Right Shift)

- Shifts the bits of the number towards the **right** a specified ***n*** number of positions. The sign bit filled with 0's.
- When we apply **>>>** on **a positive number**, it gives the same output as that of >>. It gives a positive number when we apply >>> on a negative number. MSB is replaced by a 0.
- Example, x=32; x>>>3;

  00100000 (in decimal 32)

  becomes **00000100** (in decimal 4).

The last three bits shifted out and lost.

Difference between >> and >>> operator

- Both >> and >>> are used to shift the bits towards the right.
- The difference is that the >> preserve the sign bit while the operator >>> does not preserve the sign bit.
- To preserve the sign bit, you need to add 0 in the MSB.

Control Statements

- **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.

The **if Statement**

- The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths.
- Here is the general form of the if statement:

```
if (condition)
        statement1;
else
        statement2;
```

The **if Statement …**

- Here, each statement may be a single statement or a compound statement enclosed in **curly braces** (that is, a block).
- The condition is any expression that returns a boolean value. The **else** clause is optional.
- The **if** works like this:
  - If the condition is **true**, then statement1 is executed.
  - Otherwise, statement2 (if it exists) is executed.

Example


```
class Test {
 public static void main(String args[]){
      int x = 10;
      if( x < 20 )
            System.out.print("This is if statement");
      System.out.print(“End of main");
   }
}
```

if - else


```
if (condition) {
  // block of code to be executed if the condition is true
}
else {
  // block of code to be executed if the condition is false
}
```

Example


```
class Test {
 public static void main(String args[]){
 int x = 30;
        if( x < 20 )
                System.out.print("This is if statement");
        else
                System.out.print("This is else statement");
    }
}
```

Nested if

- It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

```
if(Condition 1){
        //Executes when the Boolean Condition 1 is true
        if(Condition 2){
            //Executes when the Boolean Condition 2 is true
        }
}
```

Example


```
public class Test {
 public static void main(String args[]){
 int x = 30;
 int y = 10;
 if( x == 30 ){
                    if( y == 10 ){
                                System.out.print("X = 30 and Y = 10");
                            }
                    }
            }
  }
```

The if-else-if Ladder


```
if(condition1)
        statement1;
else if(condition2)
        statement2;
else if(condition3)
        statement3;
.
.
else
        statement+1;
```

Example


```
class Test {
public static void main(String args[]){
int x = 30;
if( x == 10 ){
                    System.out.print("Value of X is 10");
                    }
else if( x == 20 ){
                    System.out.print("Value of X is 20");
 }
else if( x == 30 ){
                    System.out.print("Value of X is 30");
 }
else{
          System.out.print("This is else statement");
 }
```

Switch - Java's multiway branch statement


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

Switch …

- **The following rules apply to a switch statement:**

- The variable used in a switch statement can only be a byte, short, int, or char.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break statement is reached.*

- When a *break statement is reached, the switch terminates, and the flow of control jumps to the next line* following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through to subsequent* cases until a break is reached.
- A *switch statement can have an optional default case, which must appear at the end of the switch. The default* case can be used for performing a task when none of the cases is true. No break is needed in the default case.

```
class SampleSwitch {
public static void main(String args[]) {
for(int i=0; i<6; i++)
switch(i) {
                case 0: System.out.println("i is zero.");
                        break;
                case 1: System.out.println("i is one.");
                        break;
                case 2: System.out.println("i is two.");
                        break;
                case 3: System.out.println("i is three.");
                        break;
                default:System.out.println("i is greater than 3.");
        }
    }
}
```

Output:

- The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Nested switch Statements

- You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested* **switch**.

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

Iteration Statements

- Java's iteration statements are **for**, **while**, and **do-while**.
- There may be a situation when we need to execute a block of code several number of times and is often referred to as a loop.

while

- The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {
        // body of 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. The curly braces are unnecessary if only a single statement is being repeated.

Example


```
class While1 {
        public static void main(String args[]) {
        int n = 10;
        while(n > 0) {
                System.out.println("n=" + n);
                n--;
            }
        }
}
```

Example 2


```
class While2{
        public static void main(String args[]){
        int x =10;
        while( x <20){
                System.out.print("value of x : "+ x );
                x++;
                System.out.print("\n");
        }
    }
}
```

do-while

- A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

The syntax of a do...while loop is:

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

- Notice that the **condition** appears at the end of the loop, so the statements in the loop execute once before the **condition** is tested.
- If the **condition** is true, the flow of control jumps back up to do, and the statements in the loop execute again.
- This process repeats until the **condition** is false.

Example1


```
class DoWhile1 {
public static void main(String args[]){
    int n = 10;
    do {
        System.out.println("n=" + n);
        n--;
      } while(n > 0);
   }
}
```

Example1


```
class DoWhile2{
public static void main(String args[]){
int x =10;
do{
        System.out.print("value of x : "+ x );
        x++;
        System.out.print("\n");
   }while( x <20);
}
}
```

for

- There are two forms of the for loop.

1. The first is the **traditional** form.
2. The second is the new "**for-each**" form.

Traditional for Loop


```
for(initialization; condition; update)
{
    // body
}
```

- The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
- Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps out of the for loop.
- After the body of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update loop control variable.
- Now Condition is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then condition). After the condition is false, the for loop terminates.

Example1


```
class Test {
 public static void main(String args[]) {
        for(int x = 1; x < 10; x ++) {
                System.out.println("value of x : " + x );
     }
 }
```

Example2


```
class Test {
    public static void main(String args[]) {
     for(int x = 1; x < 10; x =x+2) {
         System.out.println("value of x : " + x );
    }
 }
```

foreach loop or enhanced for loop

- The general form of the for-each version of the **for** is:

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

- Here, *type* specifies the type and ***itr-var*** specifies the name of an *iteration variable* that will receive the elements from the *array*, one at a time, from beginning to end.
- With each iteration of the loop, the next element in the ***array*** is retrieved and stored in *itr-var*. The loop repeats until all elements in the array have been obtained.

Example1

- The following code displays all the elements of the array ***nums***:

```
public class TestArray {
 public static void main(String[] args) {
        int nums[] = {1,2,3,4,5,6,7,8,9,10};
        for (int x: nums) {
            System.out.println("x=" + x);
        }
    }
}
```

Example2

- Program to display and find the sum of elements in the array ***nums***:

```
public class TestArray {
 public static void main(String[] args) {
        int nums[] = {1,2,3,4,5,6,7,8,9,10};
        int sum=0;
        for (int x: nums) {
            System.out.println("x=" + x);
           sum+=x;
       }
      System.out.println("sum=" + sum);
   }
}
```

Nested Loops

- One loop may be inside another. For example, here is a program that nests for loops:

```
class Nested {
        public static void main(String args[]) {
        int i, j;
        for(i=0; i<10; i++) {
                for(j=i; j<10; j++)
                        System.out.print("$");
                System.out.println();
        }
    }
}
```

Jump Statements

- Java supports three jump statements:

  break,

  continue, and

  return.

- These statements transfer control to another part of your program.

Java break

- In Java, the break statement has three uses.

1. It terminates a statement sequence in a switch statement.
2. It can be used to exit a loop.
3. Third, it can be used as a “civilized” form of goto.

Using break to Exit a Loop

- By using break, you can force immediate termination of a loop, by passing the conditional expression and any remaining code in the body of the loop.
- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
class BreakLoop {
public static void main(String args[]) {
for(int i=1; i<10; i++) {
        if(i == 5)
                break;                  // terminate loop if i is 5
        System.out.println("i: " + i);
    }
  System.out.println("Loop complete.");
  }
}
```

Using break as a Form of Goto

- This form of **break** works with a label. The Labeled **break** statement is shown here:

  break *label;*

- ***label*** is the name of a label that identifies a block of code.
- When this form of **break** executes, control is transferred out of the named block.
- To name a block, put a label at the start of it.
- A *label* is any valid Java identifier followed by a colon.
- Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the *end* of the labeled block.

Example


```
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("The Third Block");
                            }
                        System.out.println("The Second Block");
                    }
                System.out.println("This is after second block.");
            }
        }
}
```

Output

Running this program generates the following output:

Before the break.

This is after second block.

Using continue

- The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
class Continue{
        public static void main(String args[]) {
            int [] numbers = {10, 20, 30, 40, 50};
            for(int x : numbers ) {
                                        if( x == 30 ) {
                                            continue;
                                        }
                                System.out.println( x );
                        }
                }
}
```

return

- The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.
- The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main( )**.

Example


```
class Return {
        public static void main(String args[]) {
        boolean t = true;
        System.out.println("Before the return.");
        if(t)
            return;        // return to caller
        System.out.println("This won't execute.");
    }
}
```

Output:

Before the return.

THANK YOU