

Java Operators | Arithmetic Operators Example

-Siva Yannam

Java Operators | A programmer generally makes a program to perform some operation.

For example, we can perform the addition of two numbers just by using + symbol in a program.

Here, + is a symbol that performs an operation called addition. Such kind of symbols is called operators in java.

So, let us understand the different types of operators and expressions supported by java in this tutorial and learn how to use them with examples.

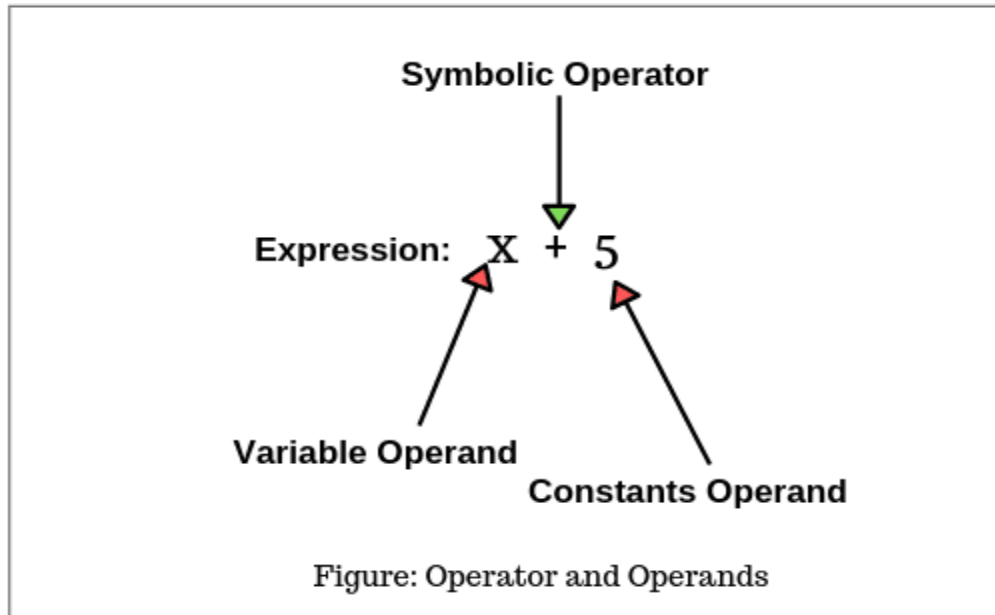
Java Operators

In Java, an expression has two parts: operand and operator. The variables or constants that operators act upon are called **operands**.

An operator in java is a symbol or a keyword that tells the compiler to perform a specific mathematical or logical operations.

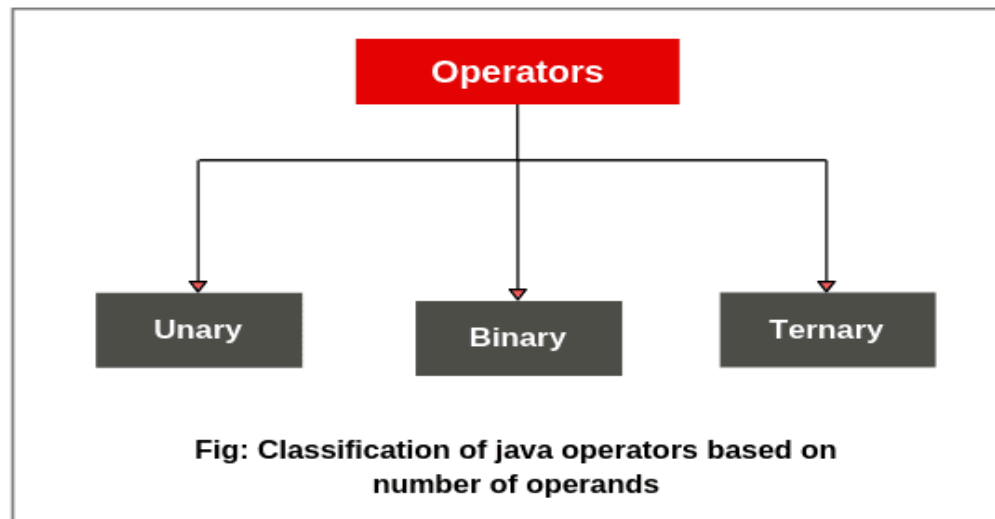
They are used in programs to manipulate data and variables. They generally form mathematical or logical expressions. An expression is a combination of variables, constants, and operators.

For example, an expression is $x+5$. Here, the operand x is a variable, operand 5 is a constant, and + is an operator that acts on these two operands and produces the desired result.



Types of Operators in Java

There are three types of operators in java based on the number of operands used to perform an operation. These operators are shown in the below figure.



1. Unary operator: The operator that acts on a single operand is called unary operator. A unary operator uses a single variable.

2. Binary operator: The operator that acts on two operands is called binary operator. A binary operator uses two variables.

3. Ternary operator: The operator that acts on three operands is called ternary operator. A ternary operator uses three variables.

Based on the notation used, Java operator has been divided into two categories: Symbolic and named.

If a symbol like +, -, *, etc is used as an operator, it is called symbolic operator. If a keyword is used as an operator, it is called named operator.

Classification of Operators based on Symbols and Named in Java

The Java operators can be classified on the basis of symbols and named. They are as follows:

1. Symbolic operator in Java

- Arithmetic operators: +, -, *, /, etc.
- Relational operators: <, >, <=, >=, =, !=.
- Logical operators: &&, ||, !.
- Assignment operators: =,
- Increment and decrement operators: ++, --
- Conditional operators: ?:
- Bitwise operators: &, !, ^, ~, <<, >>, >>>
- Shift operators: <<, >>, >>>.

2. Named operators in Java

- instanceof operator

Let us understand various types of operators in detail.

Arithmetic Operators in Java

Java Arithmetic operators are used to performing fundamental arithmetic operations such as addition, subtraction, multiplication, and division on numeric data types.

The numeric data types can be char, byte, short, int, long, float, and double. Java provides five arithmetic operators. They are listed in the below table.

Table: Java Arithmetic Operators

Operators	Meaning	Description
1. +	Addition or unary plus	Performs addition operation.
2. -	Subtraction or unary minus	Performs subtraction operation.
3. *	Multiplication	Performs multiplication operation.
4. /	Division	Performs division operation.
5. %	Modulo division (Remainder)	Performs remainder after division operation.

Integer Arithmetic

If both operands in a single arithmetic expression are integers, the expression is called arithmetic expression, and the operation is called integer arithmetic. Integer arithmetic always takes integer values.

Let's create a program where we will implement all the above arithmetic operators shown in the table.

Program source code 1:

```
package arithmeticPrograms;

public class IntegerTest {

    public static void main(String[] args)

    {

        // Declaration of instance variables with initialization.

        int a = 20, b = 10;

        System.out.println("a: " +a);

        System.out.println("b: " +b);

        System.out.println("a + b = " +(a + b));

        System.out.println("a - b = " +(a - b));

        System.out.println("a * b = " +(a * b));

        System.out.println("a / b = " +(a / b));

        System.out.println("a % b = " +(a % b));

    }

}
```

Output:

a: 20

b: 10

a + b = 30

```
a - b = 10
```

```
a * b = 200
```

```
a / b = 2
```

```
a % b = 0 (Remainder of integer division)
```

Let's take one more example program based on integer arithmetic operators.

Program source code 2:

```
package arithmeticPrograms;

public class IntegerTest2
{
    // Declaration of instance variables.

    int a = 50, b = 30;

    // Declaration of instance methods.

    void m1()
    {
        int p = b - a;

        System.out.println("p = " + p);
    }

    void m2()
    {
```

```
int q = -a*b;

System.out.println("q = " +q);

}

void m3()

{

    int r = a/b;

    System.out.println("r = " +r);

}

void m4()

{

    int s = -a%b;

    int u = a%-b;

    System.out.println("s = " +s);

    System.out.println("t = " +t);

    System.out.println("u = " +u);

}

public static void main(String[] args)

{
```

```
// Creating an object of class.
```

```
IntegerTest2 i = new IntegerTest2();
```

```
i.m1(); // Calling of m1() method.
```

```
i.m2(); // Calling of m2.
```

```
i.m3(); // Calling of m3.
```

```
i.m4(); // Calling of m4.
```

```
}
```

```
}
```

Output:

```
p = -20
```

```
q = -1500
```

```
r = 1
```

```
s = -20
```

```
t = -20 (Decimal part truncated)
```

```
u = 20 (Remainder of integer division)
```

1. In the preceding example program, a and b are integer types. Therefore, the result of a/b is 1 because the decimal part (divisor) has been truncated. This operation is called integer division.
2. In the case of modulus division, the sign of result is always sign of the dividend (i.e, first operand).
For example :

$-a \% b = -20$ because the sign of a (dividend) is minus.

$-a \% -b = -20$ because the sign of a is minus.

$a \% -b = 20$ because sign of a is plus.

Real Arithmetic

An arithmetic operation that involves only real operands is called real arithmetic. In real operand, values are assumed either in decimal or exponential notation.

Let's make a program where we will assume values of variables in decimal form.

Program source code 3:

```
package arithmeticPrograms;

public class RealTest
{
    public static void main(String[] args)
    {
        double a = 15.5, b = 20.2;

        System.out.println("a + b = " + (a + b));

        System.out.println("a - b = " + (a - b));

        System.out.println("a * b = " + (a * b));

        System.out.println("a / b = " + (a / b));

        System.out.println("a % b = " + (a % b));
    }
}
```

```
}
```

Output:

```
a + b = 35.7
```

```
a - b = -4.699999999999999
```

```
a * b = 313.09999999999997
```

```
a / b = 0.7673267326732673
```

```
a % b = 15.5
```

Mixed-mode Arithmetic

When one operand is integer and other operands are real, this kind of expression is called mixed-mode arithmetic operation. First, integer operand is converted into real operand and then real arithmetic operation is performed.

Let's make a program where we will take one value as integer and another value real.

Program source code 4:

```
package arithmeticPrograms;
```

```
public class MixedTest
```

```
{
```

```
    int x = 20; // Integer
```

```
    double y = 12.5; // Real
```

```
    void div()
```

```
{  
  
double z = x/y;  
  
System.out.println("z = " +z);  
  
}  
  
public static void main(String[] args)  
  
{  
  
MixedTest mt = new MixedTest();  
  
mt.div();  
  
}  
}
```

Output:

z = 1.6

In the above example program, you can see that the result is in decimal point because integer operand has been first converted into real operand.

Precedence of Arithmetic Operators

The combination of variables, constants, and operators as per the syntax of the language is called arithmetic expression in java.

It is evaluated from left to right using the rules of precedence of operators if an arithmetic expression has no parentheses.

There are two priority levels of arithmetic operation in java. They are as follows:

1. High priority: * / %
2. Low priority: + -

If parentheses are used in the expression, the expression with parentheses will be assumed with the highest priority.

If two or more sets of parentheses occur into the expression one after another, the order of evaluation will be done from left set towards the right set.

Let' take an example program based on all the above concepts.

Program source code 5:

```
package arithmeticPrograms;

public class MyTest
{
    int x = 9;
    int y = 12;
    int z = 3;
    void m1()
    {
        int exp1 = x - y / 3 + z * 2 - 1;

        System.out.println("Evaluation1 = " +exp1);
    }
}
```

```
}  
  
void m2()  
{  
  
    int exp2 = (x - y)/3 + ((z * 2) - 1);  
  
    System.out.println("Evaluation2 = " +exp2);  
  
}  
  
public static void main(String[] args)  
{  
  
    MyTest t = new MyTest();  
  
    t.m1(); // Calling of m1 method.  
  
    t.m2(); // Calling of m2 method.  
  
}  
}
```

Output:

Evaluation1 = 10

Evaluation2 = 4

Explanation of exp1: exp1 has been evaluated by the following steps.

They are as follows:

exp1 = $9 - 12/3 + 3 * 2 - 1$

Step 1: $\text{exp1} = 9 - 4 + 3 * 2 - 1$ (12/3 evaluated)

Step 2: $\text{exp1} = 9 - 4 + 6 - 1$ ($3 * 2$ evaluated)

Step 3: $\text{exp1} = 5 + 6 - 1$ ($9 - 3$ evaluated)

Step 4: $\text{exp1} = 11 - 1$ ($6 + 5$ evaluated)

Step 5: $\text{exp1} = 10$ ($11 - 1$ evaluated)

Explanation of exp2 : exp2 has been evaluated by the following steps.

They are as follows:

$\text{exp2} = (9 - 12) / 3 + ((3 * 2) - 1)$

Step 1: $\text{exp2} = -3 / 3 + ((3 * 2) - 1)$ ($9 - 12$ evaluated)

Step 2: $\text{exp2} = -3 / 3 + (6 - 1)$ ($3 * 2$ evaluated)

Step 3: $\text{exp2} = -3 / 3 + 5$ ($6 - 1$ evaluated)

Step 4: $\text{exp2} = -1 + 5$ ($-3 / 3$ evaluated)

Step 5; $\text{exp2} = 4$ ($-1 + 5$ evaluated)

Key points: Java does not have an operator for exponentiation.

Relational Operators in Java | Example Program

Relational operators in Java are those operators that are used to perform the comparison between two numeric values or two quantities. These operators determine the relationship between them by comparing operands. For example, to know which is a bigger number, comparing the age of two persons, comparing the price of two items, etc.

These comparisons can be performed with the help of relational operators. Relational operators require two operands.

Java supports six types of relational operators. They are listed in the below table.

Java Six Relational Operators

Operators	Meaning
1. <	Less than
2. <=	Less than or equal to
3. >	Greater than
4. >=	Greater than or equal to
5. ==	Equal to
6. !=	Not equal to

Key points:

1. The result of all relational operators is always of a boolean type. It returns always true or false.

2. Relational operators are mainly used to create conditions in decision statements, like this:

if(condition_is_true) statement to be executed.

This statement can be implemented in the program like this:

```
if(x > y)
```

```
System.out.println(x);
```

Relational Operators Example Programs

Let's create a simple program based on the java relational operators' concept.

Program source code 1:

```
package relationalPrograms;

public class RelationalOperatorsDemo
{
    public static void main(String[] args)
    {
        int x = 10;
        int y = 30;

        System.out.println("x = " +x+ "y = " +y);

        System.out.println("x is greater than y: " +(x>y));

        System.out.println("x is less than y: " +(x<y));

        System.out.println("x is less than equal to y: " +(x<=y));

        System.out.println("x is equal to y: " +(x==y));

        System.out.println("(x+20 < y+10): " +(x+20 < y+10));

    }
}
```

Output:

```
x = 10 y = 30
```



```
x is greater than y: false
```

```
x is less than y: true
```

```
y is greater than equal to x: true
```

```
x is less than equal to y: true
```

```
x is equal to y: false
```

```
(x+20 < y+10): true
```

In the preceding example program, the value of the relational expressions is either true or false. If the condition is true, it returns true otherwise return false statement.

The arithmetic expression $(x + 20)$ and $(y + 10)$ are evaluated first and then the result is compared between them because arithmetic operators have a higher priority over relational operators.

Let's create another program where we will use the if-else statement to compare among three numeric values.

Program source code 2:

```
package relationalPrograms;
```

```
public class Test
```

```
{
```

```
    int x = 30;
```

```
    float y = 50.5F;
```

```
    int z = 60;
```

```
void compare()

{

    if(y > x)

    {

        System.out.println("y is greater than x");

    }

    else

    {

        System.out.println("y is less than x");

    }

    if(y < z)

    {

        System.out.println("y is less than z");

    }

    else

    {

        System.out.println("y is greater than z");

    }

}
```

```
public static void main(String[] args)
{
    // Create an object of class.

    Test t = new Test();

    t.compare(); // Calling compare method using reference variable t.
}
}
```

Output:

y is greater than x

y is less than z

In the above example program, if you have difficulty understanding the concept of the if-else statement, you can skip it. When you will learn this topic in the decision-making chapter, you can easily understand this program.

Hope that this tutorial has covered important basic points related to **java relational operators** with example programs. Remember the following key points.

Key Points:

1. Relational operators in Java are the most frequently used operators in the expressions that control the if statement and different loop statements.

2. These operators are mainly used to determine equality and order.
3. The six relational operators are `<` , `>` , `<=` , `>=` , `==` , and `!=` .
4. Equality in Java is represented with two equal signs, not one. Note that a single equal sign is an assignment operator.
5. In Java, integers, floating-point numbers, characters, and booleans can be compared using the equality test, `==`, and inequality test, `!=`.

Logical Operators in Java | Example Program

Logical operators in Java are those operators which are used to form compound conditions by combining two or more conditions or relations. Sometimes in Java, these operators are also called Boolean operators because they return a boolean value.

An example of logical operators is given below.

```
x > y && x > z
```

This kind of expression that combines two or more relational expressions is called **logical expression** or compound relational expression. The result of logical expression is also a value of true or false.

Types of Logical Operators in Java

In Java, there are three types of logical operators. They are listed in the below table.

Table: Logical Operators

Operators	Meaning
-----------	---------

1. &&	AND operator
2.	OR operator
3. !	NOT operator

AND Operator in Java

In AND operator, two expressions (conditions) are combined by && operator. If both conditions are true, the operator returns true.

If one expression is false or both expressions are false then the operator returns false.

For example:

```
if(x > y && y < z)
    System.out.println("Hello Java");
```

In the above statement, there are two conditions: $x > y$ and $y < z$. Since both conditions are joined by the && operator. So if both conditions are true, "Hello Java" will be displayed.

Note that both expressions are evaluated separately and then && operator compares the result of both.

Let's create a program where we will implement && operator to combine two conditions.

Program source code 1:

```
package logicalOperatorPrograms;

public class LogicalOperatorDemo {

    public static void main(String[] args)

    {

        int x = 200;
```

```
int y = 50;

int z = 100;

if(x > y && y > z)
{
    System.out.println("Hello");
}

if(z > y && z < x)
{
    System.out.println("Java");
}

if((y+200) < x && (y+150) < z)
{
    System.out.println("Hello Java");
}
}
}
```

Output:

Java

Explanation:

1. In the first if statement, there are two conditions: $x > y$ and $y > z$. The condition $x > y$ is true but $y > z$ is not true. Therefore, the statement "Hello" is not displayed.
2. In the second if statement, both conditions $z > y$ and $z < x$ are true. Therefore, the statement "Java" is displayed.
3. In the third if statement, both conditions are false. Therefore, the statement "Hello Java" is not displayed.

OR Operator in Java

In OR operator, two or more expressions (conditions) are combined by `||` (OR) operator. If either one of the conditions is true, the operator returns true.

For example:

```
if(x = 1 || y = 1 || z = 1)

    System.out.println("Hello");
```

In the above example, there are three conditions: $x = 1$, $y = 1$, and $z = 1$ which are combined by `||` (or operator).

If either of x or y or z value becomes equal to 1 then the next statement "Hello" will be displayed. If any of the three conditions are not equal to 1, the message will not be displayed.

Let's see an example program based on the OR operator in java.

Program source code 2:

```
package logicalOperatorPrograms;

public class OROperatorExample {

    public static void main(String[] args)

    {
```

```
int x = 1;

int y = 2;

int z = 5;

System.out.println("x: " +(x==1));

System.out.println("y: " +(y==z));

System.out.println("z>x: " +(z>x));

if(x==1 || x>y || x>z)
{
    System.out.println("One");
}

if(x==y || y==2 || z==5)
{
    System.out.println("Two");
}

if(x==y || y==z || z==x)
{
    System.out.println("Three");
}

} }
```


Output:

x: true

y: false

z>x: true

One

Two

Explanation:

1. In the first if statement, first expression ($x == 1$) is true. Therefore, the statement "One" is displayed.
2. In the second if statement, two conditions ($y == 2$) and ($z == 5$) are correct. Therefore, the output "Two" is displayed.
3. In the third if statement, all conditions are not correct. Therefore, the statement "Three" is not displayed.

Not Operator in Java

The NOT operator is used to reverse the logic state of its operand. If the condition is correct, the logical NOT operator returns false. If the condition is false, the operator returns true.

For example:

```
if(!( x > y ))
```

```
System.out.println("Hello Java");
```

In the above example, if the condition ($x > y$) is true, the statement "Hello Java" will not be displayed. If the ($x > y$) is not true, the statement "Hello Java" will be displayed.

Let's take an example program related to NOT operator.

Program source code 3:

```
package logicalOperatorPrograms;

public class NotOperatorExample {

    public static void main(String[] args)

    {

        int x = 1;

        int y = 2;

        int z = 5;

        System.out.println("x: " + (!((x+2)==(1+2))));

        System.out.println("y: " + (!(y==z)));

        System.out.println("z>x: " + !(z > x));


        if(!(x==y) && ((y+5) > z) && (!((z-3)==0)))

        {

            System.out.println("Hello");

        }

    }

}
```

Output:

```
x: false  
  
y: true  
  
z>x: false  
  
Hello
```

Explanation:

1. In the expression $((x + 2) == (1 + 2))$, $(x + 2)$ is equal to $(1 + 2)$. Therefore, NOT operator returns false.
2. In the expression $(y == z)$, y is not equal to z . Therefore, the output is true.
3. In the expression $(z > x)$, z is greater than x . Therefore, the output is false.
4. In the if statement, there are three conditions: $!(x == y)$, $((y + 5) > z)$, and $!((z - 3) == 0)$.
 - In the first condition, x is not equal to y . Therefore, the NOT operator returns true.
 - In the second condition, $(y + 5)$ is greater than z . So, its return type is true.
 - In the third condition, $(z - 3)$ is not equal to 0. Therefore, the NOT operator returns true.

Since all three conditions are true for AND operator. Therefore, the statement "Hello" is displayed.

Hope that this tutorial has covered all important points related to **logical operators in java**.

Assignment Operator in Java | Example Program

An operator which is used to store a value into a particular variable is called **assignment operator in java**.

In any programming language, an assignment operator is the most commonly used to assign a value in a variable.

There are three categories of assignment operations in java programming. They are as follows:

1. Simple assignment
2. Compound assignment
3. Assignment as expression

Simple Assignment

A simple assignment can be used in two ways:

- To store or assign a value into a variable.
- To store a value of a variable into another variable.

It has the following general format to represent a simple assignment.

```
v = expression;
```

where,

- v: It is a variable name that represents a memory location where a value may be stored.
- expression: It may be a constant, variable, or a combination of constants, variables, and operators.
- = is an assignment operator.

For example:

```
1. int x = 10;  
2. int y = x; // Here, the value of variable x is stored into y.
```

Compound Assignment

The general form of compound assignment is as follows:

```
v op = expression;
```

where,

- op: It is a Java binary operator. It may be + - * / % << >> etc. v and expression are the same as explained in the simple assignment.
- The operator op = is known as a shorthand assignment operator in Java because
- v op = expression is shorthand notation for v = v operator expression.

For example:

```
1. x += 5; // It is equivalent to int x = x + 5;  
2. x -= 10; // It is equivalent to int x = x - 10;  
3. a *= 100; // Equivalent to int a = a * 100;  
4. a /= (b + c);
```

Let's take an example program based on compound assignment operator.

Program source code 1:

```
package assignmentOperatorPrograms;  
  
public class CompoundTest  
{
```

```
public static void main(String[] args)
{
    int x = 20, y = 30, z = 50;

    x += y;

    y -= x + z;

    z *= x * y;

    System.out.println("x = " +x );

    System.out.println("y = " +y );

    System.out.println("z = " +z );

}
}
```

Output:

x = 50

y = -70

z = -175000

Explanation:

1. `x += y;` will be evaluated in the following steps:

`x += y;` is equivalent to `x = x + y;`

`x = 20 + 30;`

`x = 50;`

2. $y -= x + z$; is equivalent to $y = y - (x + z)$;

$y = y - (50 + 50)$;

$y = 30 - 100$;

$y = -70$;

3. $z *= x * y$; is equivalent to $z = z * (x * y)$;

$z = z * (50 * (-70))$;

$z = 50 * (-3500)$;

$z = -175000$;

Assignment as Expression

In Java, an assignment operation is also considered an expression because the operation has a result. The result of expression is a value that is stored in a variable. It is mainly used in more than one assignment.

For example:

1. `int x = y - z + 4;` // Here, the expression $y - z + 4$ is evaluated first and then its result is stored into the variable x.

Let's take an example program related to this concept.

Program source code 3:

```
package assignmentOperatorPrograms;

public class Expression
{
    public static void main(String[] args)
    {
```

```
int a = 19, b = 31, c = 50;

a += 1;

b -= 1;

c *= 2;

int x = (10 + a);

int y = x + 100;

int z = x + y + c;

System.out.println("Value of a: " +a);

System.out.println("Value of b: " +b);

System.out.println("Value of c: " +c);

System.out.println("Value of x: " +x);

System.out.println("Value of y: " +y);

System.out.println("Value of z: " +z);

}

}
```

Output:

Value of a: 20

Value of b: 30

Value of c: 100

Value of x: 30

Value of y: 130

Value of z: 260

Explanation:

The following steps have been performed to evaluate the above expressions of the program.

1. `a += 1;`

`a = a + 1;`

`a = 19 + 1;`

`a = 20;`

2. `b -= 1;`

`b = b - 1;`

`b = 31 - 1;`

`b = 30;`

3. `c *= 2;`

`c = c * 2;`

`c = 50 * 2;`

`c = 100;`

4. `x = (10 + a);`

`x = 10 + 20; // Here, the value of x will be 20, not 19.`

`x = 30;`

5. `y = x + 100;`

`y = 30 + 100; // Here, value of x will be 30.`

`y = 130;`

6. `z = x + y + c;`

`z = 30 + 130 + 100; // Here, value of c will be 100.`

`z = 260;`

Note:

1. You cannot use more than one variable on the left-hand side of = operator.

For example:

`x + y = 20;` // It is invalid because there will be doubt to Java compiler regarding for storing the value 20.

2. You cannot use a literal or constant value on the left-hand side of = operator.

For example:

`20 = a;` // It is also invalid because how can we store value of x in a number.

Hope that this tutorial has covered all important points related to **assignment**

Unary Operator in Java | Pre, Post Increment

The operator that acts on a single operand is called **unary operator in Java**.

A unary operator uses a single variable. There are three types of unary operators in java. They are as follows:

1. Unary minus operator (`-`)
2. Increment operator (`++`)
3. Decrement operator (`--`)

Unary minus operator in Java

The operator which is used to negate a given value is called unary minus operator in java. Let's understand it with an example program.

Program source code 1:

```
package unaryOperatorPrograms;

public class MinusExample
{
    public static void main(String [] args)
    {
        int x = 5;

        x = -5;

        System.out.println(x);
    }
}
```

Output:

-5

In this code snippet, the value of variable x is 5 in the beginning. When we applied the unary minus (-) operator on it then it became -5 because the unary minus operator has negated its value.

Negation means the conversion of negative value into positive value and vice-versa.

Increment (++) Operator in Java

The operator which is used to increase the value of a variable (operand) by one is called increment operator in java. It is represented by (++).

The operator ++ adds 1 to the operand. It can have the following forms:

```
++x; or x++;
```

++x is equivalent to $x = x + 1$; (or $x += 1$;).

Consider the following examples:

```
1. int a = 5;
```

```
    ++a;
```

```
2. int a = 5;
```

```
    a++
```

Here, the value of variable a is incremented by 1 when ++ operator is used before or after x. In Java, both are valid expressions.

Let's take one more example to understand it better.

```
a = a + 1;
```

In this expression, if we a is equal to 5, then $a + 1$ value will be 6. This value will be stored again left-hand side of variable x. Thus, the value of x now becomes 6.

The same thing can be performed by using ++ operator also. ++ operator can be used in two forms:

1. Pre incrementation (Prefix):

When we write ++ operator before a variable, it is called pre incrementation or prefix. In pre incrementation, the increment is done first and then any other operation will be performed.

In other words, the prefix operator first adds 1 to an operand, and then the result is stored in the left-hand side variable.

2. Post incrementation (Postfix):

When we write ++ operator after a variable, it is called post incrementation or postfix. In post-increment, all the other operations are performed first, and then at the end increment is done only.

In other words, the postfix operator first stores the value in the left-side variable and then the operator adds 1 to an operand.

Let's take an example program related to this concept to understand better.

Program source code 2:

```
package operatorPrograms;

public class IncOperatorTest1
{
    public static void main(String[] args)
    {
        int i = 0;

        System.out.println(i); // Output 0

        System.out.println(++i); // Output 1, first increment the value of i and then displays it as 1.

        System.out.println(i); // Output 1, displays the value of i because i is already incremented, i.e. 1 at right-hand side.
```

```
System.out.println(i++); // Output 1, first displays the value of i as 1 and then increment it.
```

```
System.out.println(i); // Output 2, displays the incremented value of i, i.e. 2.
```

```
}
```

```
}
```

Output:

0

1

1

1

2

Explanation:

1. The first pre-increment operator increases the value of i by 1 and outputs the new value of 1.
2. The next post-increment operator also increases the value of i by 1 but outputs the value of i as 1 before increment occurs.
3. After the post-increment, the output is displayed as 2.

Program source code 3:

```
package operatorPrograms;

public class IncrOperatorTest2
{

public static void main(String[] args)
```

```
{  
  
int x = 50;  
  
int y = 100;  
  
int z = 200;  
  
  
int a, b, c;  
  
a = ++x;  
  
b = y++;  
  
c = x + y++ + ++z;  
  
System.out.println("x = " +x);  
  
System.out.println("y = " +y);  
  
System.out.println("z = " +z);  
  
System.out.println("a = " +a);  
  
System.out.println("b = " +b);  
  
System.out.println("c = " +c);  
  
}  
  
}
```

Output:

x = 51

```
y = 102
```

```
z = 201
```

```
a = 51
```

```
b = 100
```

```
c = 353
```

Explanation:

In the above example program, the initial values of x, y, z are 50, 100, and 200 respectively.

1. When the statement `a = ++x` is executed, the value of x is incremented first by 1 and then the value of x is displayed as 51.

After incrementing the value of x is stored into the variable a. Therefore, the value of a is also displayed as 51 (same as that value of x). Thus, this operation happened in the below form:

```
x = x + 1;
```

```
a = x;
```

2. When the statement `b = y++` is executed, the current value of y is stored into the variable b because the increment operator is in postfix form. Therefore, the value of b is displayed as 100.

After storing the current value of y into b, y's value is incremented by 2 because y is modified two times in the program and then it is assigned to y. Therefore, the value of y is displayed as 102.

Thus, this operation occurs as follows:

```
b = y;
```

```
y = (y + 1) + 1;
```


3. When the statement `c = x + y++ + z++` is executed, the value of `x` is assigned as 51 because the value of `x` is now 51 after increment. The value of `y` is incremented by 1 and then it is assigned as 101 into `y` because `y++` is in postfix form.

Similarly, the value of `z` is also incremented by 1 and then it is assigned 201 into `z`. The sum of three values `51 + 101 + 201` will be stored into the variable `c`. Thus, the output is 353.

Let's find the value of following expression `++x*x++`, given that the value of `x` is 10.

Program source code 4:

```
package operatorPrograms;

public class IncrOperatorTest3 {

    public static void main(String[] args)

    {

        int x = 10;

        int m = ++x * x++;

        System.out.println("m = " +m);

    }

}
```

Output:

```
m = 121
```

Explanation:

1. In the above program, the value of variable x is 10. When ++x is executed, the operator will increment the value of x by 1, and then it will assign the value of x as 11.

2. When x++ is executed, the value of x will not be incremented in the statement because this is post incrementation. Hence, the value of x will stay the same i.e. 11. Thus, the output will be $11 * 11 = 121$.

Find the value of x and y in the following expression, given that the value of x is equal to 20.

Program source code 5:

```
package operatorPrograms;

public class IncrOpertaorTest4
{
    public static void main(String[] arg)
    {
        int x = 20;

        int y = ++x * 10 / x++ + ++x;


        System.out.println("x = " + x);

        System.out.println("y = " + y);

    }
}
```

Output:

x = 23

 `y = 33`**Explanation:**

This question is more complicated than the previous example. So, let's understand the explanation of the above program.

1. When `++x` will be executed in the numerator of expression, first, value's `x` is incremented by 1 and then returned to the expression as 21, which is multiplied by 10.

So, the operation will occur like this: `int y = 21 * 10 / x++ + ++x; // x assigned value of 21.`

2. Next, when `x++` in the denominator of expression will be executed, the value of `x` is again incremented by 1 but the original value 21 of `x` will use in the expression because it is post-increment.

So, the next operation will be like this: `int y = 21 * 10 / 21 + ++x; // x assigned value of 21.`

3. The final assignment of `x` increments the value of `x` by 1 because it is pre-increment. So, the value of `x` is now 23 because, after post-increment, the value of `x` returned as 22 to the `++x`.

We can simplify this: `int y = 21 * 10 / 21 + 23; // x assigned value of 23.`

4. Finally, we can easily evaluate multiply and division from left to right and perform simple addition. Thus, the final value of `x` will be printed as 23, and the value of `y` is 33.

Decrement (—) Operator in Java

The operator which is used to decrement the value of a variable (operand) by one is called decrement operator in java. It is represented by (—).

The operator `--` subtracts 1 to the operand. It can have the following forms:

```
--X; or X--;
```

```
--X is equivalent to X=X - 1; (or X -=1; ).
```

Like `++` operator, `--` operator can also be used in two forms:

1. Pre decrementation (Prefix):

When we write `--` operator before a variable, it is called pre decrementation or prefix. In pre decrementation, the decrement is done first and then any other operation will be performed.

In other words, the prefix operator first subtracts 1 to an operand, and then the result is stored in the left-hand-side variable.

2. Post decrementation (Postfix):

When we write `--` operator after a variable, it is called post decrementation or postfix. In post-decrement, all the other operations are performed first, and then at the end increment is done only.

In other words, the postfix operator first stores the value in the left-side variable and then the operator subtracts 1 to an operand. Post decrement is performed after all other operations are carried out.

Consider the following examples:

```
1. int a = 5;
```

```
int p = --a; // pre-decrement.
```

Here, the value of variable `a` is decremented by 1 and then assigned to `a`. In both cases, the value of `a` and `p` would be 4. The operation will occur like this;

```
a = a - 1;
```

```
a = 5 - 1;
```

```
a = 4;

p = 4;

2. int a = 5;

int q = a--; // post-decrement.
```

Here, first, the original value of a will be assigned to variable q and then it is decremented by 1. In this case, the value of a is 4, and q is 5. The operation will be like this:

```
q = 5;

a = a - 1;

a = 5 - 1;

a = 4;
```

Let's take a simple example program based on the decrement operator. Program source code 6:

```
package operatorPrograms;

public class DecrOperatorTest1

{

public static void main(String[] args)

{

int a = 1;

System.out.println(a); // Output 1

System.out.println(--a); // Output 0
```

```
System.out.println(a); // Output 0

System.out.println(a--); // Output 0

System.out.println(a); // Output -1

}

}
```

Output:

```
1
0
0
0
-1
```

Let's take a complex example program based on all the above concepts.

Program source code 7:

```
package operatorPrograms;

public class DecrOperatorTest2 {

    public static void main(String[] args)

    {

        int a = 1;

        ++a;

        int b = a++;
```

```
a--;  
  
int c= --a;  
  
int x = a * 10 / (b - c);  
  
System.out.println("a = " +a);  
  
System.out.println("x = " +x);  
  
}  
  
}
```

Output:

```
a = 1
```

```
x = 10
```

Hope that this tutorial has covered all the important points related to the **unary operator in java**.

Conditional (Ternary) Operator in Java Example

The character pair `? :` is called **ternary operator in Java** because it acts on three variables.

This operator is also known as conditional operator in Java. It is used to make a conditional expression.

It has the following syntax in java:

Syntax:

```
variable = exp1 ? exp2 : exp3;
```

where `exp1`, `exp2`, and `exp3` are expressions.

Operator? works as follows:

First of all, the expression `exp1` is evaluated. If it is true, the expression `exp2` will be evaluated and the value of `exp2` will store in the variable. If `exp1` is false, `exp3` will be evaluated and its value will store in the variable.

Consider the following example:

```
int a = 40;

int b = 30;

int x = (a > b) ? a : b;
```

Here, first `(a > b)` is evaluated. If it is true, the value of `a` is stored in the variable `x`. If it is false, the value of `b` is stored in the variable `x`.

Here, `(a > b)` is true because the value of `a` is greater than the value of `b`. Therefore, the value of `a` will assign in the variable `x`. That is, `x = 40`.

In the above example, we have used 3 variables `a`, `b`, and `x`. That's why it is called ternary operator in java. This example can also be achieved by using the if-else statement as follows:

```
if(a > b)

{

    x = a;

}

else {
```



```
x = b;  
  
}
```

Let's create a program where we will find out the greatest number between the two numbers.

Program source code 1:

```
package operatorPrograms;  
  
public class Test  
{  
  
    public static void main(String[] args)  
    {  
  
        int x = 20;  
  
        int y = 10;  
  
        int z = (x > y) ? x : y;  
  
        System.out.println("Greatest number: " +z);  
  
    }  
  
}
```

Output:

```
Greatest number: 20
```

Hope that this tutorial has covered all important points related to the **conditional (ternary) operator in java**

Bitwise Operator in Java | Bitwise AND, OR, XOR

An operator that acts on individual bits (0 or 1) of the operands is called **bitwise operator in java**.

It acts only integer data types such as byte, short, int, and long. Bitwise operators in java cannot be applied to float and double data types.

The internal representation of numbers in the case of bitwise operators is represented by the binary number system. Binary number is represented by two digits 0 or 1.

Therefore, these operators are mainly used to modify bit patterns (binary representation).

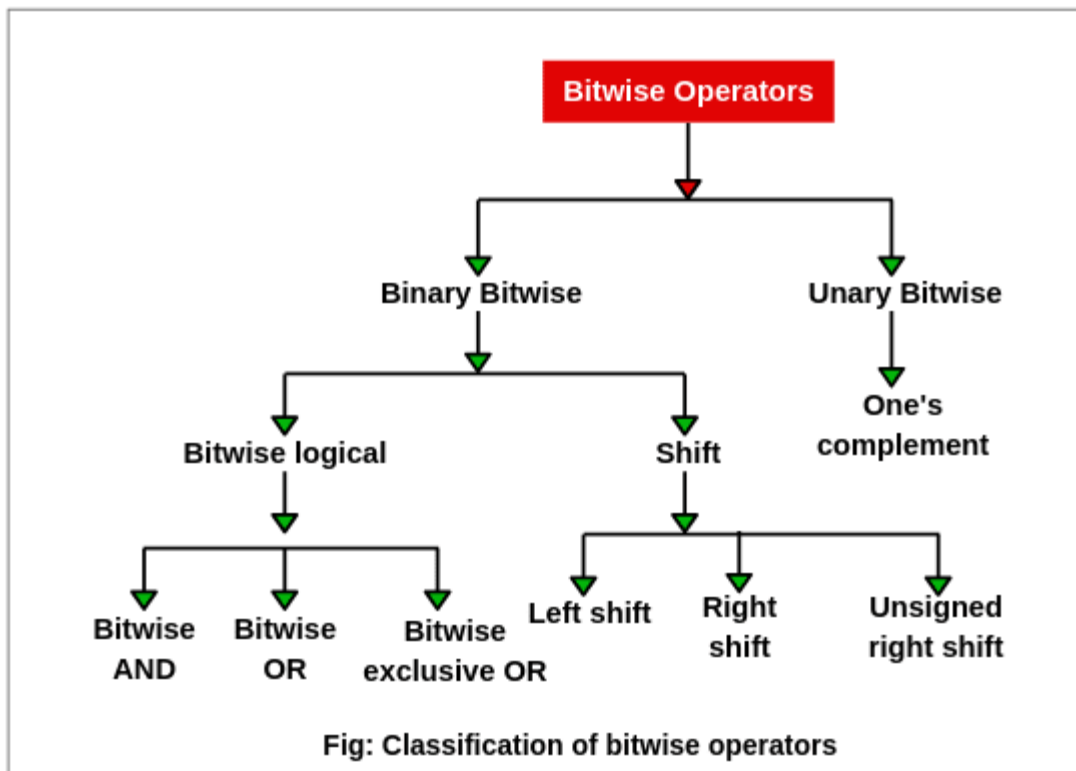
Types of Bitwise Operators in Java

Operator	Meaning
1. &	bitwise AND
2.	bitwise OR
3. ^	bitwise exclusive OR
4. ~	one's complement (unary)
5. <<	shift left
6. >>	shift right
7. >>>	unsigned right shift

In Java, there are seven types of bitwise operators. They are listed in the below table form.

Table: Bitwise Operators

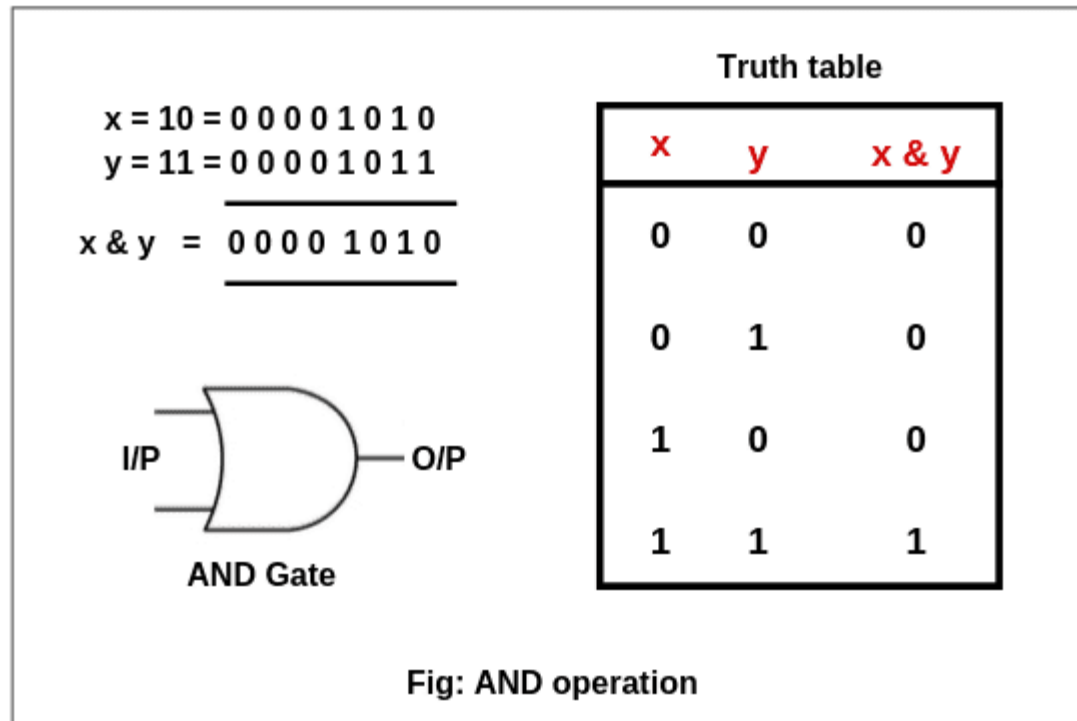
The classification of bitwise operators in java can also be seen in the below figure.



Bitwise AND Operator (&) in Java

This operator is used to perform bitwise AND operation between two integral operands. The AND operator is represented by a symbol & which is called ampersand. It compares each bit of the left operand with the corresponding bit of right operand.

Let's consider the truth table given in the figure to understand the operation of AND operator.



Truth table is a table that gives the relationship between inputs and output. In AND operation, on multiplying two or more input bits, we get output bit.

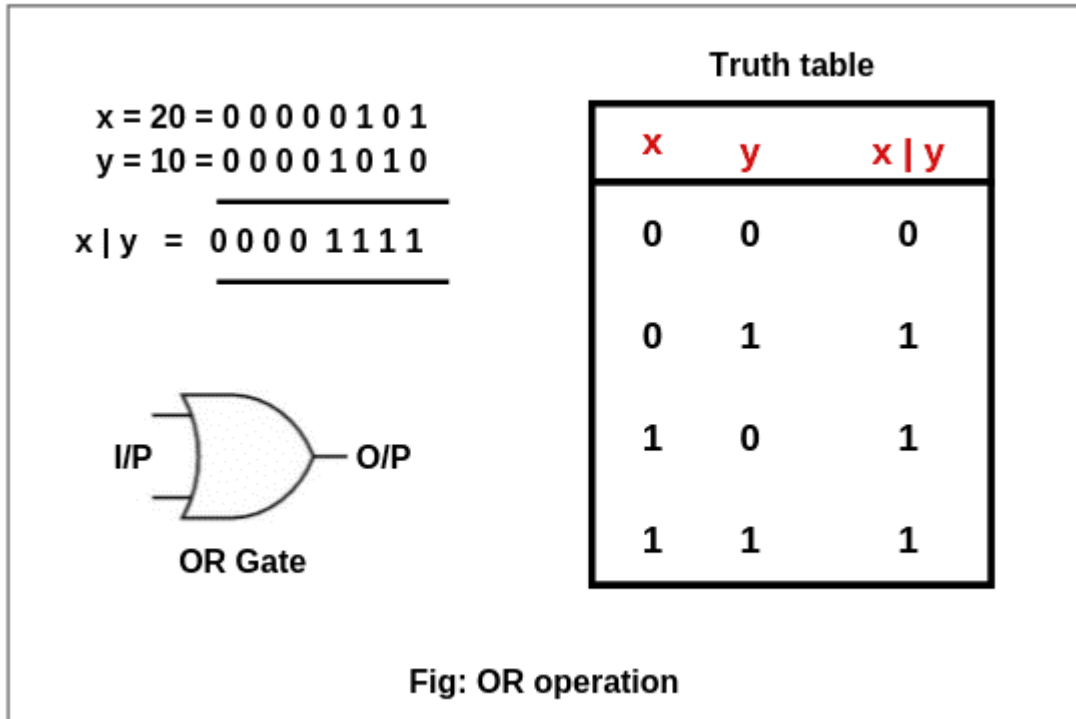
From the truth table shown in figure, if both compared input bits are 1, we get output 1. Otherwise, we get output 0.

From the truth table, On multiplying the individual bits of x and y, we get $x \& y = 00001010$. It is nothing but 10 in decimal form.

Bitwise OR Operator (|) in Java

This operator is used to perform OR operation on bits of numbers. It is represented by a symbol | called pipe symbol.

In OR operation, each bit of first operand (number) is compared with the corresponding bit of the second operand.



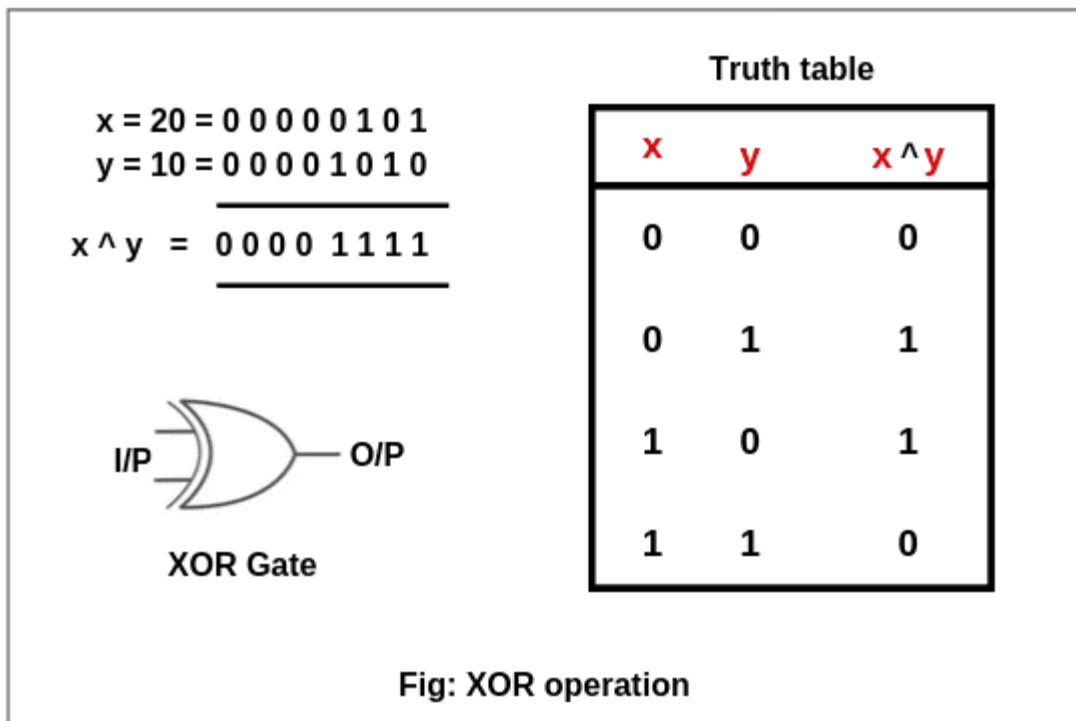
To understand OR operation, consider truth table given in the above figure. If both compared bits are 0, the output is 0. If any one bit is 1 in both bits, the output is 1.

On adding input bits of ($x = 20$) and ($y = 10$), we get output bits 0 0 0 0 1 1 1. This is nothing but 15 in decimal form.

Bitwise Exclusive OR (XOR) Operator (^)

The operator ^ performs exclusive OR (XOR) operation on the bits of numbers. This operator compares each bit first operand (number) with the corresponding bit of the second operand.

Exclusive OR operator is represented by a symbol \wedge called cap. Let us consider the truth table of the XOR operator to understand exclusive OR operation.



From the truth table, when we get odd number of 1's then notice that the output is 1 otherwise the output is 0. Hence, $x \wedge y = 00001111$ is nothing but 15 in decimal form.

Hope that this tutorial has covered almost all important points related to the **bitwise operator in java** with suitable examples.

Shift Operator in Java | Left, Signed, Unsigned Right

The operator which is used to shift the bit patterns right or left is called **shift operator in java**.

The general form of shift expression is as follows:

```
left_operand op n
```

where,

left_operand → left operand that can be of integral type.

op → left shift or right shift operator.

n → number of bit positions to be shifted. It must be of type int only.

For example:

```
10 << 2
```

Here, the value 10 is operand, << is left shift operator, 2 is the number of bit positions to be shifted.

Types of Shift Operator in Java

There are two types of shift operators in Java. They are:

1. Left shift operator
2. Right shift operator

Left Shift Operator in Java

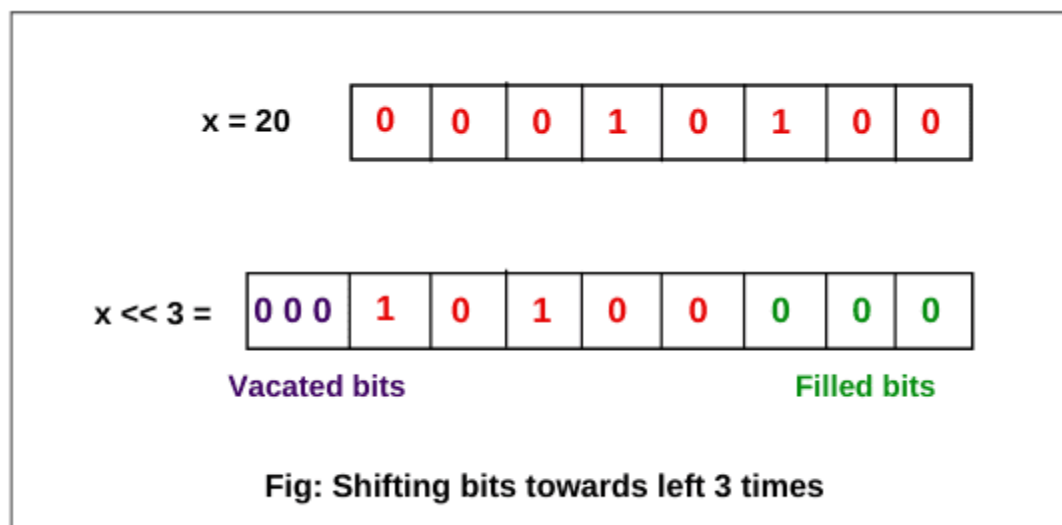
The operator that shifts the bits of number towards left by n number of bit positions is called **left shift operator in Java**.

This operator is represented by a symbol `<<`, read as double less than. If we write `x << n`, it means that the bits of x will be shifted towards left by n positions.

Let us take an example to understand the concept of the Java left shift operator.

1. If `int x = 20`. Calculate x value if `x << 3`.

The value of x is `20 = 0 0 0 1 0 1 0 0` (binary format). Now `x << 3` will shift the bits of x towards left by 3 positions. Due to which leftmost 3 bits will be lost.



Key point: Shifting a value to the left, n bits, is equivalent to multiplying that value by 2^n .

For example: In the above expression, $n = 3$. So, $20 * 2^3 = 20 * 8 = 160$.

Let's take another example based on the left shift operator.

2. If `a = 45`, calculate a value if `a << 1`.

The value 45 is represented in binary format as 0 0 1 0 1 1 0 1. If this value is shifted towards left by one position, we will get 0 1 0 1 1 0 1 0. This number in decimal form is nothing but 90 that is twice of 45. i.e. $45 * 2^1 = 90$.

Right Shift Operator in Java

The operator that shifts the bits of number towards the right by n number of bit positions is called right shift operator in Java. The right shift operator in java is represented by a symbol `>>`, read as double greater than.

If we write `x >> n`, it means that the bits of x will be shifted towards right by n positions. There are two types of right shift operators in java:

1. Signed right shift operator (`>>`)
2. Unsigned right shift operator (`>>>`)

Both of these operators shifts bits of number towards right by n number of bit positions.

Signed Right Shift Operator

The signed right shift operator `>>` shifts bits of the number towards the right and also reserves the sign bit, which is leftmost bit. A sign bit represents the sign of a number.

If the sign bit is 0 then it represents a positive number. If the sign bit is 1, it represents a negative number.

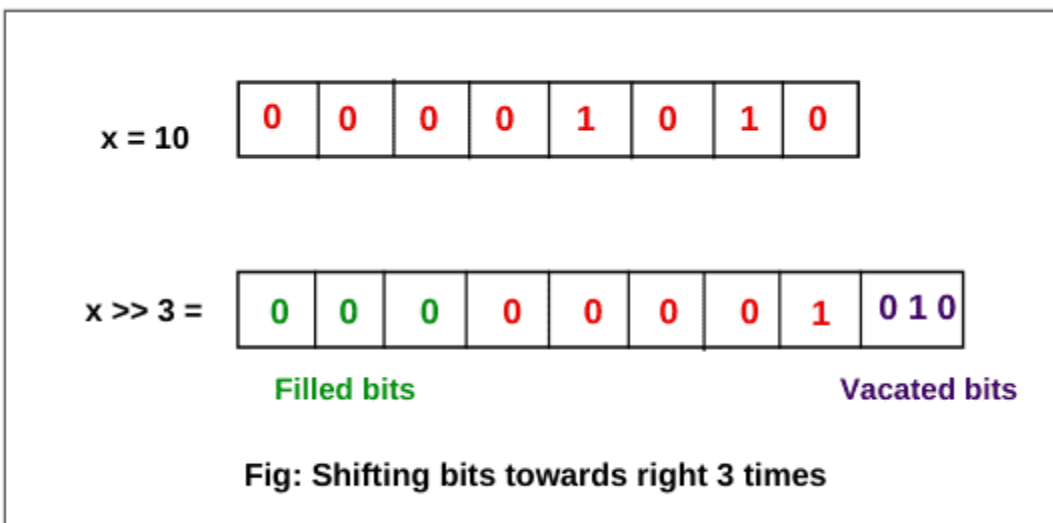
If the number is positive, the leftmost position is filled with 0. If the number is negative, the leftmost position is filled with 1. The signed shift operator uses the same sign as used in the number before shifting of bits.

Let's understand the concept of right shift operator with the help of an example.

1. If `int x = 10` then calculate `x >> 3` value.

The value of `x` is `10 = 0 0 0 0 1 0 1 0`. Since the number is positive, the leftmost bit position will be filled with 0. Now `x >> 3` will shift the bits of `x` towards the right by 3 positions. The rightmost 3 bits will be lost due to shifting.

Hence, after shifting, bits of `x` is `0 0 0 0 0 0 0 1` that is 1 in decimal form. The bit pattern after the right shift is shown in the below figure:



You will notice in this example that after performing right shift operation on a positive number 10, we get a positive value 1 as a result.

Similarly, if we perform right shift operation on a negative number, again we will get a negative value only.

Key point: Shifting a value to the right is equivalent to dividing the number by 2^n .

Let's take some examples.

```
1. int i = 10;
```

```
int result = i >> 3; // result = 1
```

In this expression, $10 / 2^3 = 10 / 8 = 1$.

2. `int i = 20;`

```
int result = i >> 2; // result = 5.
```

In this expression, $20 / 2^2 = 5$.

3. `int i = -20`

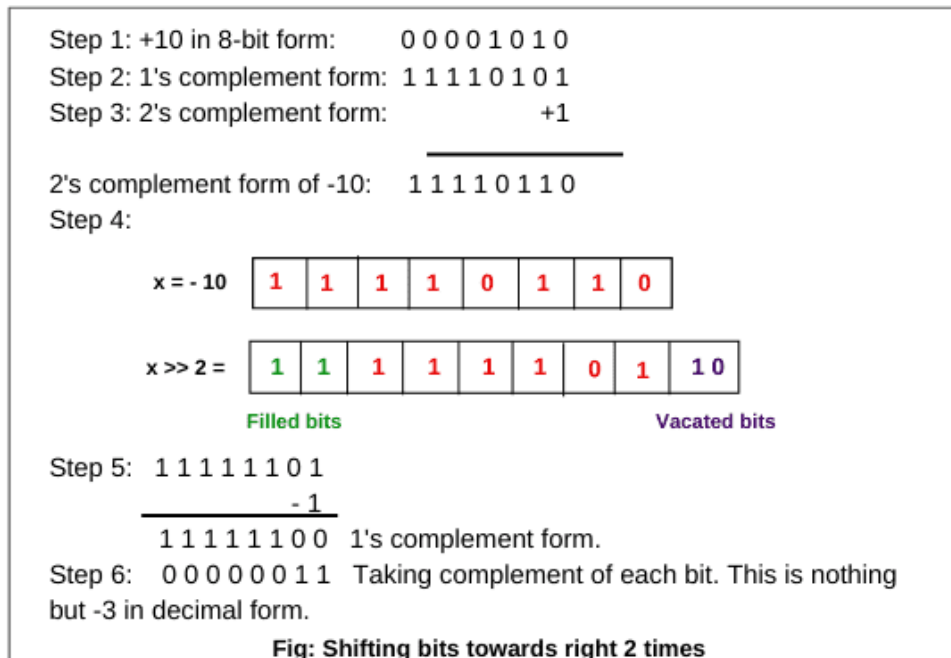
```
int result = i >> 2; // result = -5.
```

In this expression, $-20 / 2^2 = -5$.

Let's take an example that is based on right shifting on a negative number.

2. If `int x = -10` then calculate `x >> 2` value.

The value of `x` is `-10` that is a negative number. So, we will use 2's complement system to represent the negative numbers. Follow all the



1. +10 in 8-bit form is 0 0 0 0 1 0 1 0.

2. Obtain the 1's complement of 0 0 0 0 1 0 1 0. The 1's complement can be obtained by simply complementing each bit of the number, that is, by changing all 0s to 1s and all 1s to 0s.

After taking complement of 0 0 0 0 1 0 1 0, the result in 1's complement form is 1 1 1 1 0 1 0 1.

3. Now add 1 to get 2's complement form. After adding 1 in LSB (Least Significant Bit), the result is 1 1 1 1 0 1 1 0. This is in 2's complement form.

Here, 1 in MSB (Most Significant Bit) represents negative number. Thus, the 2's complement form of -10 is 1 1 1 1 0 1 1 0.

4. Since the number is negative, the leftmost position is filled with two 1s. Now $x \gg 2$ will shift the bits of x towards the right by 2 positions.

The rightmost 2 bits will be lost due to shifting. Hence, after shifting, bits of x in 2's complement form is 1 1 1 1 1 1 0 1.

5. Now convert 2's complement bits into 1's complement bits. Subtract 1 in LSB to convert 2's complement form into 1's complement. The result in 1's complement form is 1 1 1 1 1 1 0 0.

6. Take the complement of each bit to get original form of number after shifting. So, the result in original form is 0 0 0 0 0 0 1 1. This is nothing but -3 in decimal form.

Let's verify the above result by creating a program in java.

Program source code 1:

```
package shiftOperator;

public class SignedRightShiftExample
{
    public static void main(String[] args)
    {
        int x = -10,
        c = 0;

        c = x >> 2;

        System.out.println("x >> 2 = " + c);
    }
}
```

Output:

```
x >> 2 = -3
```

Unsigned Right Shift Operator

The unsigned right shift operator in java performs nearly the same operation as the signed right shift operator in java. The unsigned right shift operator is represented by a symbol `>>>`, read as triple greater than.

The unsigned right shift operator always fills the leftmost position with 0s because the value is not signed. Since it always stores 0 in the sign bit, it is also called zero fill right shift operator in java.

If you will apply the unsigned right shift operator `>>>` on a positive number, it will give the same result as that of `>>`. But in the case of negative numbers, the result will be positive because the signed bit is replaced by 0.

Key points: Both signed right shift operator `>>` and unsigned right shift operator `>>>` are used to shift bits towards the right.

The only difference between them is that `>>` preserve sign bits whereas `>>>` does not preserve sign bit. It always fills 0 in the sign bit whether number is positive or negative.

Let us create a program to observe the effect of various shift operators.

Program source code 2:

```
package shiftOperator;

public class UnsignedRightShiftExample {

    public static void main(String[] args)

    {

        int x = 10;

        int y = -10;
```

```
System.out.println("x >> 1 = " + (x >> 1));  
  
System.out.println("x >>> 1 = " + (x >>> 1));  
  
System.out.println("y >> 2 = " + (y >> 2));  
  
System.out.println("y >>> 2 = " + (y >>> 2));  
  
}  
  
}
```

Output:

```
x >> 1 = 5
```

```
x >>> 1 = 5
```

```
y >> 2 = -3
```

```
y >>> 2 = 1073741821
```

Hope that this tutorial has covered almost all topics related to **shift operator in Java**.