***# OPERATORS***

# In Java, = is the **assignment operator**. It is the most common and important operator in any programming language.

# In Java, we have 5 **arithmetic operators**:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | Used for addition and string concatenation |
| - | Used for subtraction |
| \* | Used for multiplication |
| / | Used for division |
| % | Remainder/Modulus operator for finding remainder |

**Note:**When any of the above arithmetic operators is used in an expression, if at least one operand is of type double, float or long then the operation is carried out by automatically widening the other operand also to that former's type (if it is not already of the former's type) and the result is also of the former's type. And for all other numeric types (like byte, short, char) the operands are automatically widened to int and the result is also an int.

/////////////////////////////////////////////////////////

# In Java, below are the **comparison and relational operators**:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | Checks if two values are equal |
| != | Checks if two values are not equal |
| > | Checks for greater-than condition |
| >= | Checks for greater-than-or-equals condition |
| < | Checks for less-than condition |
| <= | Checks for less-than-or-equals condition. |

////////////////////////////////////////////////////////

# In Java, we have 5 **Unary operators**:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | Unary Plus, used for indicating a positive value |
| - | Unary Minus, subtracts from zero. |
| ++ | Increment operator. Increments the value by one. |
| -- | Decrement operator. Decrements the value by one. |
| ! | Negation operator. Negates a boolean value. |

**Note:**When Increment or Decrement operator is applied before an operand (e.g. ++i or --i), the value of the operand is changed first and the changed value is substituted in the expression.  
When Increment or Decrement operator is applied after an operand (e.g. i++ or i--), the original value of the operand is substituted in the expression first and the value is changed later.

//////////////////////////////////////////////////////////

# In Java, && represents conditional-and operator (also called **logical-and**) and || represents conditional-or operator (also called logical-or).

Both conditional-and operator(&&) and conditional-or operator(||) operators evaluate from left to right.

In case of a &&, if the left expression evaluates to false, it will skip evaluating the expression on the right side of the operator.  
In case of a ||, if the left expression evaluates to true, it will skip evaluating the expression on the right side of the operator.

The result of the expression involving any of the above mentioned conditional operators is always a boolean value (true or false).

//////////////////////////////////////////////////////////

# There is yet another conditional operator called the **ternary operator** ?:. Its usage is

condition ? expression1 : expression2

The condition should always evaluate to a boolean. If the condition evaluates to true, expression1 is evaluated and its value returned, else expression2 is evaluated and its value returned

/////////////////////////////////////////////////////////

**# Bitwise and Bit Shift operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| ~ | is the **unary bitwise complement** operator. It flips the bits represented by a number. Its usage is ~operand, the operand should be of primitive integral type. For example if a binary number 110 is flipped we get 001 |
| & | is the **bitwise AND** operator. Its usage is operand1 & operand2, where operand1 and operand2 should be of primitive integral type. It sets a bit to 1 if and only if both of the corresponding bits in its operands are 1, and to 0 if the bits differ or both are 0. E.g. 1010 & 1100 will result in 1000. |
| | | is the **bitwise OR** operator. Its usage is operand1 | operand2, where operand1 and operand2 should be of primitive integral type. It sets a bit to 1 if one or both of the corresponding bits in its operands are 1, and to 0 if both of the corresponding bits are 0. E.g. 1010 | 1100 will result in 1110. |
| ^ | is the **bitwise XOR** or exclusive OR operator. Its usage is operand1 ^ operand2, where operand1 and operand2 should be of primitive integral type. It sets the bit to 1 where the corresponding bits in its operands are different, and to 0 if they are the same. E.g. 1010 ^ 1100 will result in 0110. |
| << | is the **binary left shift** operator. Its usage is operand1 << operand2, where operand1 should be of primitive integral type and operand2 should be a number which indicates the number of bits that should be shifted left in operand1.  The value of number << bitsToBeShifted is number left-shifted by bitsToBeShifted bit positions. This is equal to multiplication by 2bitsToBeShifted. E.g. Decimal 3 in binary is 11. Hence, 11 << 1 will result in 110, i.e, 3 << 1 results in 6 11 << 2 will result in 1100, i.e, 3 << 2 results in 12  **Note:** If the number's value after the operation results in overflow (i.e becomes greater than the MAX\_VALUE of number's type), then positive numbers become negative and negative numbers become zero. |
| >> | is the **binary right shift** operator. Its usage is operand1 >> operand2, where operand1 should be of primitive integral type and operand2 should be a number which indicates the number of bits that should be shifted right in operand1.  The value of number >> bitsToBeShifted is number right-shifted by bitsToBeShifted bit positions while preserving the sign. The resulting value is equal to the rounded up value of number / 2bitsToBeShifted. For non-negative values of number, this is equivalent to truncating integer division, as computed by the integer division operator /, i.e. number / 2bitsToBeShifted.  E.g. 110 >> 1 will result in 11, i.e, 6 >> 1 results in 3 1111 >> 2 will result in 11, i.e, 15 >> 2 results in 3 -15 >> 2 results in -4 |
| >>> | is the **unsigned binary right shift** operator. Its usage is operand1 >>> operand2, where operand1 should be of primitive integral type and operand2 should be a number which indicates the number of bits that should be shifted right in operand1.  For all positive numbers it is the same as >> operator. For negative numbers the leftmost bit used for sign is filled with 0, thereby changing the value and sign. |

//////////////////////////////////////////////////////////

# **Instance of Operator**

In Java, there is a keyword instanceof, which is used to verify if an object is an instance of a particular class.  
Its usage is

reference instanceof ClassName

For example, the expression (args instanceof String[]) evaluates to true

The result of an expression using instanceof is always a boolean value (true or false).

#ex

String name1 = "Ganga";

String name2 = "Amazon";

boolean statement1 = name1 instanceof String; // output: TRUE

////////////////////////////////////////////////////////////

# + operator is also used to **concatenate strings**

|  |  |
| --- | --- |
| **Expression** | **Result** |
| "Ganga" + "River" | "GangaRiver" |
| "Ganga" + (2 + 3) | "Ganga5" |
| "Ganga" + 2 + 3 | "Ganga23" |
| 2 + 3 + "Ganga" | "5Ganga" |
| 2.3f + "Ganga" | "2.3Ganga" |
| "Ganga" + 'R' + 3 + true | "GangaR3true" |

Below are the rules involved in String concatenation:

1. The + operator is syntactically left-associative, when used in string concatenation or numeric addition
2. If one operand in the concatenation expression is of type String, the other operand is also converted to String.
3. The result of a String concatenation expression is a reference to a String object containing characters in the left-side operand followed by the characters in the right-side operand.

////////////////////////////////////////////////////////////////

**#  compound assignment operators**

+= -= \*= /= %= &= ^= |= <<= >>= >>>=

**Usage:**

int x = 3, y = 4, z = 0;

z += x; // is same as writing z = z + x;

z -= y; // is same as writing z = z - y;

////////////////////////////////////////////////////////////////////

**# Operator Precedence**

In an expression when two or more operators are used there are certain rules followed to evaluate the expression. For example:

int result1 = 3 + 4 \* 3; // evaluates to 15 and not 21

The below table shows the precedence applied to the operators during their evaluation.

|  |  |
| --- | --- |
| **Precedence** | **Operators** |
| *expr*++ *expr*-- | postfix |
| ++*expr* --*expr* +*expr* -*expr* ~ ! | unary |
| \* / % | multiplicative |
| + - | additive |
| << >> >>> | shift |
| < > <= >= instanceof | relational |
| == != | equality |
| & | bitwise AND |
| ^ | bitwise exclusive OR |
| | | bitwise inclusive OR |
| && | logical AND |
| || | logical OR |
| ? : | ternary |
| = += -= \*= /= %= &= ^= |= <<= >>= >>>= | assignment |