The 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 −

Assume integer variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

The Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example

== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

----

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators −

Assume integer variable A holds 60 and variable B holds 13 then −

Show Examples

Operator	Description	Example

& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators

The following table lists the logical operators −

Assume Boolean variables A holds true and variable B holds false, then −

Show Examples

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true

! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

The Assignment Operators

Following are the assignment operators supported by Java language −

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same

		as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Increment and Decrement Operator in Java :

1. It is one of the variation of "Arithmetic Operator".
2. Increment and Decrement Operators are Unary Operators.
3. Unary Operator Operates on One Operand.
4. Increment Operator is Used to Increment Value Stored inside Variable on which it is operating.
5. Decrement Operator is used to decrement value of Variable by 1 (default).

Types of Increment and Decrement Operator :

1. Pre Increment / Pre Decrement Operator
2. Post Increment / Post Decrement Operator

Syntax :

```
++ Increment operator : increments a value by 1
```

```
-- Decrement operator : decrements a value by 1
```

Pre-Increment Operator :

1. "++" is written before Variable name.
2. Value is Incremented First and then incremented value is used in expression.
3. "++" cannot be used over "**Constant**" of "**final Variable**".

Live Example 1 : Post Incrementing Variable

[468×60]

```
class PostIncrement {
  public static void main(String args[]) {
    int num1 = 1;
    int num2 = 1;

    num1++;
    num2++;

    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
  }
}
```

Output :

```
num1 = 2
num2 = 2
```

- **Post Increment :** Increment Value of Variable After Assigning
- **num1++** will increment value inside "**num1**" variable after assigning old value to itself.
- New Value of num1 = 2.

Live Example 2 : Pre Incrementing Variable

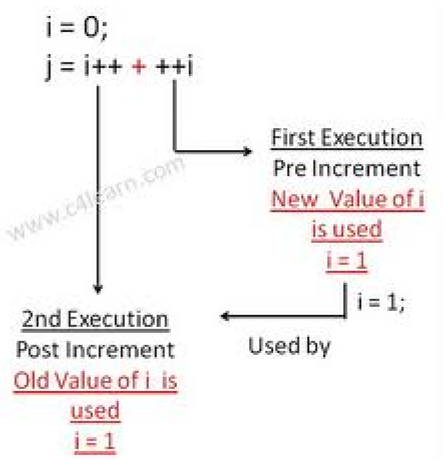
```
class PreIncrement {
  public static void main(String args[]) {
    int num1 = 1;
    int num2 = 1;

    --num1;
    --num2;

    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
  }
}
```

Output :

```
num1 = 0
num2 = 0
```

- **Pre Increment :** First Increment Value and then Assign Value.
- **++num1** will increment value inside "**num1**" variable. New value is assigned to itself.
- New Value of num1 = 0.

Live Example 3 : Post Incrementing Variable



```
class PreIncrement {
  public static void main(String args[]) {
    int num1;
    int num2;
    int num3;

    num1 = 100;
    num2 = ++num1;
    num3 = num2++ + ++num1;

    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
    System.out.println("num3 = " + num3);
  }
}
```

Explanation :

- Initially num1 = 100.

Consider this Expression –

```
num2 = ++num1;
```

- Value is Assigned to Variable Present at Left hand Side, just after incrementing "**num1**" .

```
Step 1 : Increment Value of num1.
Step 2 : New Value of num1 = 101
Step 3 : Assign Value of num1 to num2
Step 4 : New Value of num2 = 101
```

Consider this Expression –

```
num3 = num2++ + ++num1;
```

- During Expression evaluation only pre incremented value is used. (post increment operator is used just after completion of expression evaluation)

```
Step 0 : Initial Values -
         num1 = 101
         num2 = 101
Step 1 : Increment Value of num1 as it is (pre-increment)
Step 2 : Keep Value of num2 as it is (post-increment)
Step 3 : At this stage -
         num1 = 102
         num2 = 101
Step 4 : Add num1 & num2
         num3 = Old(num2) + new(num1)
         num3 = 101 + 102
         num3 = 203
Step 5 : Expression Completed
         num3 = 203
Step 6 : Increment Value of num2 (pending post-increment)
         num2 = 102
Step 7 : Final Values
         num1 = 102
         num2 = 102
         num3 = 203
```

- Conditional operators:

The Java Conditional Operator selects one of two expressions for evaluation, which is based on the value of the first operands. It is also called ternary operator because it takes three arguments.

The conditional operator is used to handling simple situations in a line.

Syntax:

```
expression1 ? expression2:expression3;
```

The above syntax means that if the value given in Expression1 is true, then Expression2 will be evaluated; otherwise, expression3 will be evaluated.

Example:

```
val == 0 ? you are right:you are not right;
```

Program to show Conditional Operator works

```
Example:

public class condiop {

 public static void main(String[] args) {

  String out;

  int  a = 6, b = 12;

  out = a==b ? "Yes":"No";

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

 }

}
```

Output:

```
Ans: No
```

In the above example, the condition given in expression1 is false because the value of a is not equal to the value of b.

Arithmetic expressions

Arithmetic expressions in Java are composed with the usual operators +, –, *, / and the remainder operator %. Multiplication and division operations have higher priority than addition and subtraction. Operations with equal priority are performed from left to right. Parenthesis are used to control the order of operation execution.

```
a + b / c ≡ a + ( b / c )

a * b - c ≡ ( a * b ) - c

a / b / c ≡ ( a / b ) / c
```

It is recommended to always use parenthesis to explicitly define the order of operations, so you do not have to remember which operation has higher priority.

A thing worth mentioning is *integer division*.

The result of division in Java depends on the types of the operands. If both operands are integer, the result will be integer as well. The unintended use of integer division may therefore lead to significant precision loss. To let Java perform real division (and get a real number as a result) at least one of the operands must be of real type.

For example:

```
3 / 2 ≡ 1
```

```
2 / 3 ≡ 0
```

because this is integer division. However,

```
3 / 2. ≡ 1.5

2.0 / 3 ≡ 0.66666666…
```

because 2. and 2.0 are real numbers. If `k` and `n` are variables of type `int`, `k/n` is integer division. To perform a real division over two integer variables or expressions you should force Java to treat at least one of them as real. This is done by *type casting*: you need to write the name of the type you are converting to before the variable in parentheses, for example `(double)k/n` will be real division with result of type `double`.

Integer division is frequently used together with the *remainder operation* to obtain the row and column of the item from its sequential index. Suppose you have a collection of 600 items, say, seats in a theater, and want to arrange them in 20 rows, each row containing 30 seats. The expressions for the seat number in a row and the row would be:

Seat number: `index % 30` (remainder of division of index by 30: 0 - 29)

Row number: `index / 30` (integer division of index by 30: 0 - 19)

where index is between 0 and 599. For example, the seat with index 247 will be in the row 8 with seat number 7.

The power operation in Java does not have an operand (if you write `a^b` this will mean bitwise OR and not power). To perform the power operation you need to call the `pow()` function:

`pow( a, b )` ≡ a^b

Java supports several useful shortcuts for frequent arithmetic operations over numeric variables. They are:

`i++ ≡ i = i+1` (increment i by 1)

`i-- ≡ i = i-1` (decrement i by 1)

`a += 100.0 ≡ a = a + 100.0` (increase a by 100.0)

`b -= 14 ≡ b = b - 14` (decrease b by 14)

Note that, although these shortcuts can be used in expressions, their evaluation has effect, it changes the value of the operands.

Expression Evaluation

Evaluate an expression represented by a String. Expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

Infix Notation: Operators are written between the operands they operate on, e.g. 3 + 4 .
Prefix Notation: Operators are written before the operands, e.g + 3 4
Postfix Notation: Operators are written after operands.

Infix Expressions are harder for Computers to evaluate because of the addional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms. A very well known algorithm for converting an infix notation to a postfix notation is Shunting Yard Algorithm by Edgar Dijkstra. This algorithm takes as input an Infix Expression and produces a queue that has this expression converted to a postfix notation. Same algorithm can be modified so that it outputs result of evaluation of expression instead of a queue. Trick is using two stacks instead of one, one for operands and one for operators. Algorithm was described succinctly on http://www.cis.upenn.edu/matuszek/cit594-2002/Assignments/5-expressions.htm, and is re-produced here.

OPERATOR PRECEDENCE IN JAVA

Java has well-defined rules for specifying the order in which the operators in an expression are evaluated when the expression has several operators. For example, multiplication and division have a higher precedence than addition and subtraction. Precedence rules can be overridden by explicit parentheses.

Precedence order.

When two operators share an operand the operator with the higher *precedence* goes first. For example, 1 + 2 * 3 is treated as 1 + (2 * 3), whereas 1 * 2 + 3 is treated as (1 * 2) + 3 since multiplication has a higher precedence than addition.

Associativity.

When an expression has two operators with the same precedence, the expression is evaluated according to its *associativity*. For example `x = y = z = 17` is treated as `x = (y = (z = 17))`, leaving all three variables with the value 17, since the `=` operator has right-to-left associativity (and an assignment statement evaluates to the value on the right hand side). On the other hand, `72 / 2 / 3` is treated as `(72 / 2) / 3` since the `/` operator has left-to-right associativity. Some operators are not associative: for example, the expressions `(x <= y <= z)` and `x++--` are invalid.

Precedence and associativity of Java operators.

The table below shows all Java operators from highest to lowest precedence, along with their associativity. Most programmers do not memorize them all, and even those that do still use parentheses for clarity.

Level	Operator	Description	Associativity
16	`[]`	access array element	left to right

	`.`
	`()`
		access object member
		parentheses	

15	`++` `--`	unary post-increment unary post-decrement	not associative

14	`++`
	`--`
	`+`
	`-`
	`!`
	`~`
		unary pre-increment
		unary pre-decrement
		unary plus
		unary minus
		unary logical NOT
		unary bitwise NOT
			right to left

13	`()`
	`new`
		cast
		object creation
			right to left

12	`* / %`	multiplicative	left to right

11	`+ -`
	`+`
		additive
		string concatenation
			left to right

10	`<< >>`
	`>>>`
		shift	left to right

9	`< <=`
	`> >=`
	`instanceof`
		relational	not associative

8	`==`
	`!=`
		equality	left to right

7	`&`	bitwise AND	left to right
6	`^`	bitwise XOR	left to right
5	`|`	bitwise OR	left to right
4	`&&`	logical AND	left to right
3	`||`	logical OR	left to right
2	`?:`	ternary	right to left
1	`= += -=`	assignment	right to left

	`*= /= %=`
	`&= ^= |=`
	`<<= >>= >>>=`		

Type conversion in Java with Examples

When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. For example, assigning an int value to a long variable.

Widening or Automatic Type Conversion

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type.

For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.

Byte –> Short –> Int –> Long – > Float –> Double

Widening or Automatic Conversion

```
class Test
{
    public static void main(String[] args)
    {
        int i = 100;

        // automatic type conversion
        long l = i;

        // automatic type conversion
        float f = l;
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

Output:

```
Int value 100

Long value 100

Float value 100.0
```

Narrowing or Explicit Conversion

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

Double –> Float –> Long –> Int –> Short –> Byte

Narrowing or Explicit Conversion

char and number are not compatible with each other. Let's see when we try to convert one into other.

```
//Java program to illustrate incompatible data
// type for explicit type conversion
public class Test
{
  public static void main(String[] argv)
  {
    char ch = 'c';
    int num = 88;
    ch = num;
  }
}
```

Error:

```
7: error: incompatible types: possible lossy conversion from int to
char

    ch = num;

         ^
```

```
1 error
```

How to do Explicit Conversion?

Example:

```
//Java program to illustrate explicit type conversion
class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;

        //explicit type casting
        long l = (long)d;

        //explicit type casting
        int i = (int)l;
        System.out.println("Double value "+d);

        //fractional part lost
        System.out.println("Long value "+l);

        //fractional part lost
        System.out.println("Int value "+i);
    }
}
```

Output:

```
Double value 100.04

Long value 100

Int value 100
```

While assigning value to byte type the fractional part is lost and is reduced to modulo 256(range of byte).
Example:

```
//Java program to illustrate Conversion of int and double to byte
class Test
{
    public static void main(String args[])
    {
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("Conversion of int to byte.");

        //i%256
        b = (byte) i;
        System.out.println("i = " + i + " b = " + b);
        System.out.println("\nConversion of double to byte.");

        //d%256
        b = (byte) d;
        System.out.println("d = " + d + " b= " + b);
    }
}
```

Output:

```
Conversion of int to byte.

i = 257 b = 1


Conversion of double to byte.

d = 323.142 b = 67
```

Type promotion in Expressions

While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:

1. Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
2. If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

Example:

```
//Java program to illustrate Type promotion in Expressions
class Test
{
    public static void main(String args[])
    {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;

        // The Expression
        double result = (f * b) + (i / c) - (d * s);

        //Result after all the promotions are done
        System.out.println("result = " + result);
    }
}
```

Output:

```
Result = 626.7784146484375
```

Explicit type casting in Expressions

While evaluating expressions, the result is automatically updated to larger data type of the operand. But if we store that result in any smaller data type it generates compile time error, due to which we need to type cast the result.

Example:

```
//Java program to illustrate type casting int to byte
class Test
{
    public static void main(String args[])
    {
        byte b = 50;

        //type casting int to byte
        b = (byte)(b * 2);
        System.out.println(b);
    }
}
```

Output

```
 100
```

NOTE- In case of single operands the result gets converted to int and then it is type casted accordingly, as in the above example.

Java Operators Precedence and Associativity

Java operators have two properties those are *precedence*, and *associativity*. Precedence is the priority order of an operator, if there are two or more operators in an expression then the operator of highest priority will be executed first then higher, and then high. For example, in expression 1 + 2 * 5, multiplication (*) operator will be processed first and then addition. It's because multiplication has higher priority or precedence than addition.

Alternatively, you can say that when an operand is shared by two operators (2 in above example is shared by + and *) then higher priority operator picks the shared operand for processing. From above example you would have understood the role of precedence or priority in execution of operators. But, the situation may not be as straightforward every time as it is shown in above example. What if all operators in an expression have same priority? In that case the second property associated with an operator comes into play, which is *associativity*. Associativity tells the direction of execution of operators that can be either *left to right* or *right to left*. For example, in expression `a = b = c = 8` the assignment operator is executed from right to left that means c will be assigned by 8, then b will be assigned by c, and finally a will be assigned by b. You can parenthesize this expression as `(a = (b = (c = 8)))`. Note that, you can change the priority of a Java operator by enclosing the lower order priority operator in parentheses but not the associativity. For example, in expression `(1 + 2) * 3` addition will be done first because parentheses has higher priority than multiplication operator.

Before discussing individual classes of operators, below table presents all Java operators from highest to lowest precedence along with their associativity.

Table 1: Java operators - precedence chart highest to lowest

Precedence	Operator	Description	Associativity
1	[] () .	array index method call member access	Left -> Right
2	++ -- + - ~ !	pre or postfix increment pre or postfix decrement unary plus, minus bitwise NOT logical NOT	Right -> Left
3	(type cast) new	type cast object creation	Right -> Left
4	* / %	multiplication division modulus (remainder)	Left -> Right
5	+ - +	addition, subtraction string concatenation	Left -> Right
6	<< >> >>>	left shift signed right shift unsigned or zero-fill right shift	Left -> Right
7	< <=	less than less than or equal to	Left -> Right

	> >= instanceof	greater than greater than or equal to reference test	
8	== !=	equal to not equal to	Left -> Right
9	&	bitwise AND	Left -> Right
10	^	bitwise XOR	Left -> Right
11	|	bitwise OR	Left -> Right
12	&&	logical AND	Left -> Right
13	||	logical OR	Left -> Right
14	? :	conditional (ternary)	Right -> Left
15	= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment and short hand assignment operators	Right -> Left

2. DECISION MAKING, BRACHING AND LOOPING

Java If-else Statement

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in java.

- if statement
- if-else statement
- if-else-if ladder

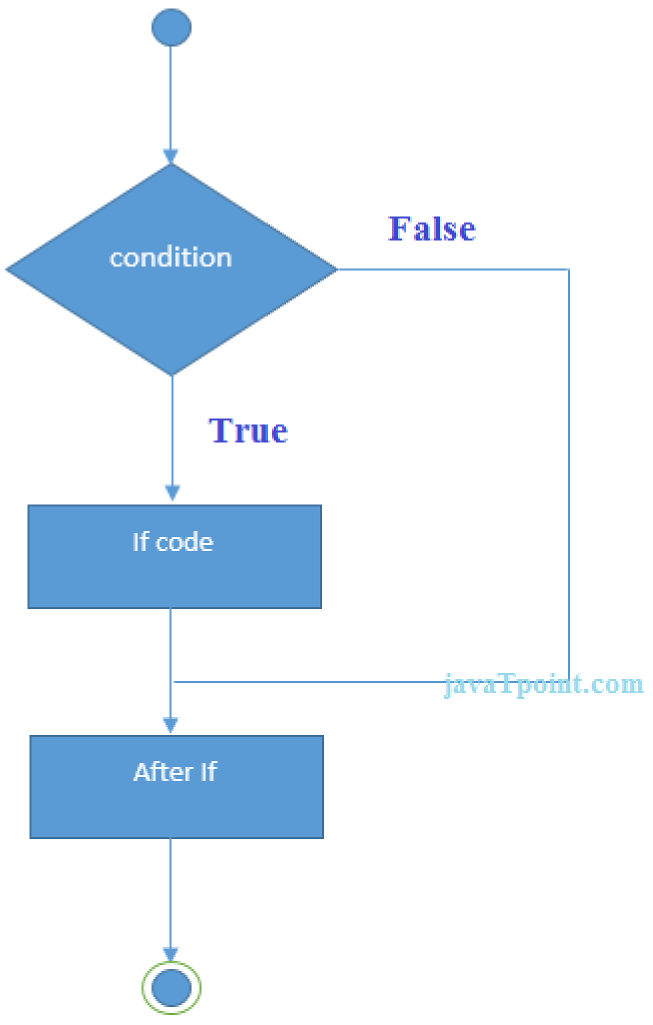
- nested if statement

Java if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

```
if(condition){
//code to be executed
}
```



Example:

```
//Java Program to demonstate the use of if statement.
public class IfExample {
public static void main(String[] args) {
    //defining an 'age' variable
    int age=20;
    //checking the age
    if(age>18){
        System.out.print("Age is greater than 18");
    }
}
}
```

Output:

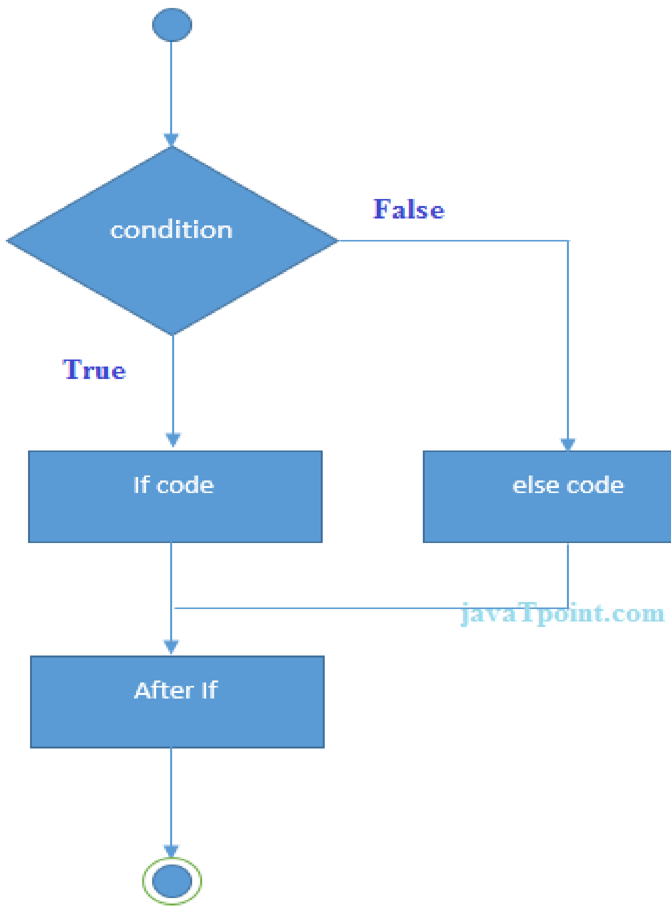
```
Age is greater than 18
```

Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



Example:

```java
//A Java Program to demonstrate the use of if-else statement.
//It is a program of odd and even number.
public class IfElseExample {
public static void main(String[] args) {
    //defining a variable
    int number=13;
    //Check if the number is divisible by 2 or not
    if(number%2==0){
        System.out.println("even number");
    }else{
        System.out.println("odd number");
    }
}
}
```

Output:

```
odd number
```

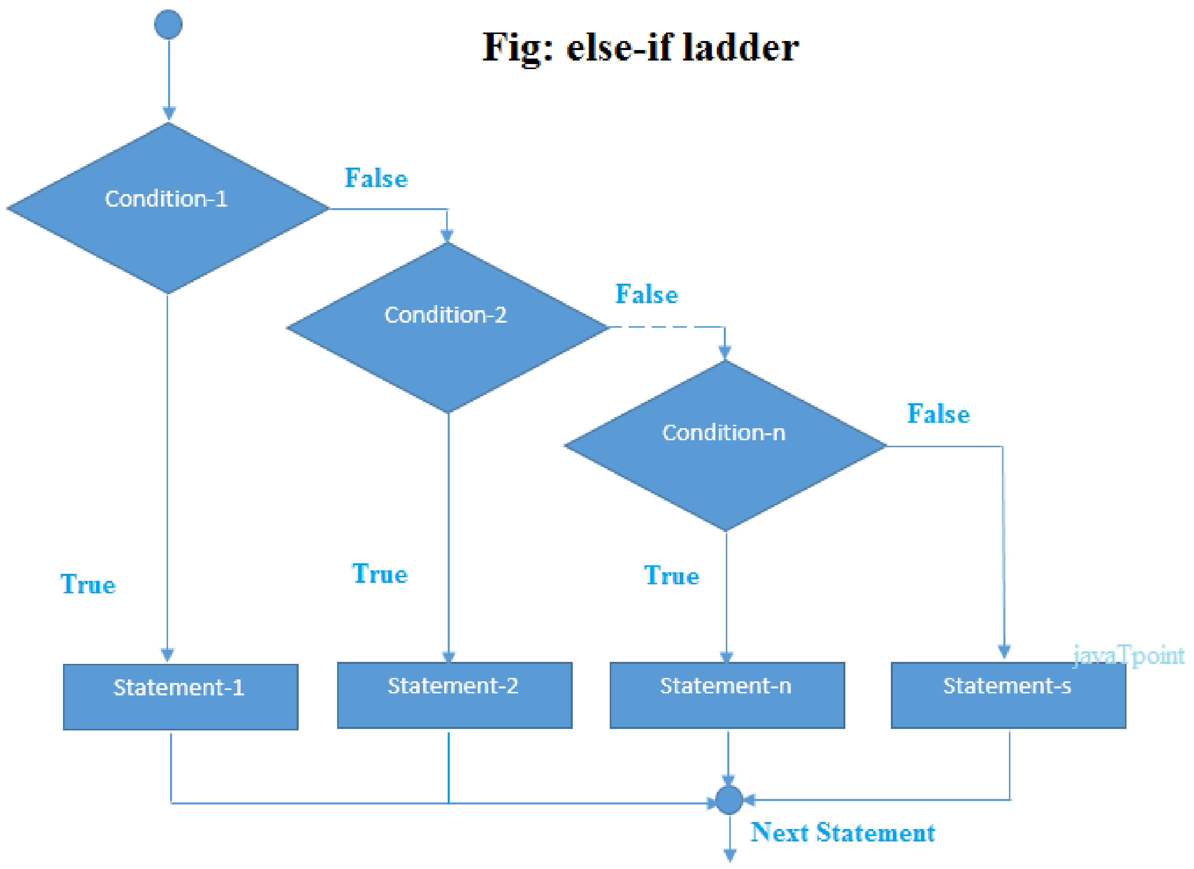
Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```java
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

Fig: else-if ladder



Example:

```
//Java Program to demonstrate the use of If else-if ladder.

//It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.

public class IfElseIfExample {
public static void main(String[] args) {
    int marks=65;

    if(marks<50){
        System.out.println("fail");
    }
    else if(marks>=50 && marks<60){
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
}
}
```

Output:

```
C grade
```

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.

In other words, the switch statement tests the equality of a variable against multiple values.

Points to Remember

- There can be *one or N number of case values* for a switch expression.
- The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.
- The case values must be *unique*. In case of duplicate value, it renders compile-time error.
- The Java switch expression must be of *byte, short, int, long (with its Wrapper type), enums and string*.
- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
- The case value can have a *default label* which is optional.

Syntax:

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......

default:
 code to be executed if all cases are not matched;
}
```

Example:

```
public class SwitchExample {
public static void main(String[] args) {
    //Declaring a variable for switch expression
    int number=20;
    //Switch expression
    switch(number){
    //Case statements
    case 10: System.out.println("10");
    break;
    case 20: System.out.println("20");
    break;
    case 30: System.out.println("30");
    break;
    //Default case statement
    default:System.out.println("Not in 10, 20 or 30");
    }
}

}
```

Output:

```
20
```

conditional operator

The Java Conditional Operator selects one of two expressions for evaluation, which is based on the value of the first operands. It is also called ternary operator because it takes three arguments.

The conditional operator is used to handling simple situations in a line.

Syntax:

```
expression1 ? expression2:expression3;
```

The above syntax means that if the value given in Expression1 is true, then Expression2 will be evaluated; otherwise, expression3 will be evaluated.

Example:

```
val == 0 ? you are right:you are not right;
```

Program to Show Conditional Operator Works

Example:

```
public class condiop {

 public static void main(String[] args) {

  String out;

  int  a = 6, b = 12;

  out = a==b ? "Yes":"No";

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

 }

}
```

Output:

```
Ans: No
```

In the above example, the condition given in expression1 is false because the value of a is not equal to the value of b.

Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

Java While Loop

The `while` loop loops through a block of code as long as a specified condition is `true`:

Syntax

```
while (condition) {

  // code block to be executed

}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
int i = 0;

while (i < 5) {

  System.out.println(i);

  i++;

}
```

The Do/While Loop

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {

  // code block to be executed

}

while (condition);
```

The example below uses a `do/while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
int i = 0;
do {

  System.out.println(i);

  i++;
}
while (i < 5);
```

Java For Loop

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:

Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
 }
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

Example explained

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Java Break and Continue

Java Break

You have already seen the `break` statement used in an earlier chapter of this tutorial. It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a **loop**.

This example jumps out of the loop when i is equal to 4:

Example

```
for (int i = 0; i < 10; i++) {

  if (i == 4) {

    break;

  }

  System.out.println(i);

}
```

Java Continue

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

This example skips the value of 4:

Example

```
for (int i = 0; i < 10; i++) {

  if (i == 4) {

    continue;

  }

  System.out.println(i);

}
```

Break and Continue in While Loop

You can also use `break` and `continue` in while loops:

Break Example

```
int i = 0;
while (i < 10) {
  System.out.println(i);
  i++;
  if (i == 4) {
    break;
  }
}
```

In **Labelled Continue Statement**, we give a *label/name* to a loop.

When this continue statement is encountered with the label/name of the loop, it skips the execution any statement within the loop for the current iteration and *continues* with the next iteration and condition checking in the *labelled loop*.

Labelled continue example

```
//Labelled continue example

class LabelledContinue
{
public static void main(String... ar)
{


loop:
for(int i=0;i<2;i++)
for(int j=0;j<5;j++)
{
        if(j==2)
                continue loop;

        System.out.println("i ="+i);
        System.out.println("j ="+j);
}

System.out.println("Out of the loop");
} //main method ends

} //class ends
```

Output

```
i =0
j =0
i =0
j =1
i =1
j =0
i =1
j =1
Out of the loop
```

As you may see in the example, outer **for-loop** was named **outer**. When the labelled continue statement was encountered during the iteration, *i equals to 0 and j equals to 2*, it skipped executing the two print statements in the loop after it and the program continued with the next iteration i.e. i equals to 1 and j equals to 0.

When the labelled continue statement was encountered during the iteration, *i equals to 1 and j equals to 2*, it skipped executing the two print statements again and continued with the loop(which ended) and finally the print statement, *System.out.println()* is executed.

In **Labelled Break Statement**, we give a *label/name* to a loop.

When this **break statement** is encountered with the *label/name of the loop*, it *skips* the execution any statement after it and takes the control right out of this labelled loop.

And, the control goes to the first statement right after the loop

Labelled break statement example with while loop

```
//Labelled break example with while loop

public class LabelledBreak
{
public static void main(String... ar)
{

int i=7;

loop1:
```

```
while(i<20)
{
        if(i==10)
                break loop1;

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

System.out.println("Out of the loop");

} //main method ends

} //class ends
```

Output

```
7
8
9
Out of the loop
```