History of Java

Java is a **programming language** and a **platform**. Java is a simple, portable, platform independent, high performance, multithreaded , high level, robust, class based, concurrent, general purpose, object-oriented and secure programming language. Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java. JDK 1.0 was released on January 23, 1996. After the first release of Java, there been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application
2) Web Application
3) Enterprise Application
4) Mobile Application

Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)
2) Java EE (Java Enterprise Edition)
3) Java ME (Java Micro Edition)
4) JavaFX

Java Version History

Many java versions have been released till now. The current stable release of Java is Java SE 10.

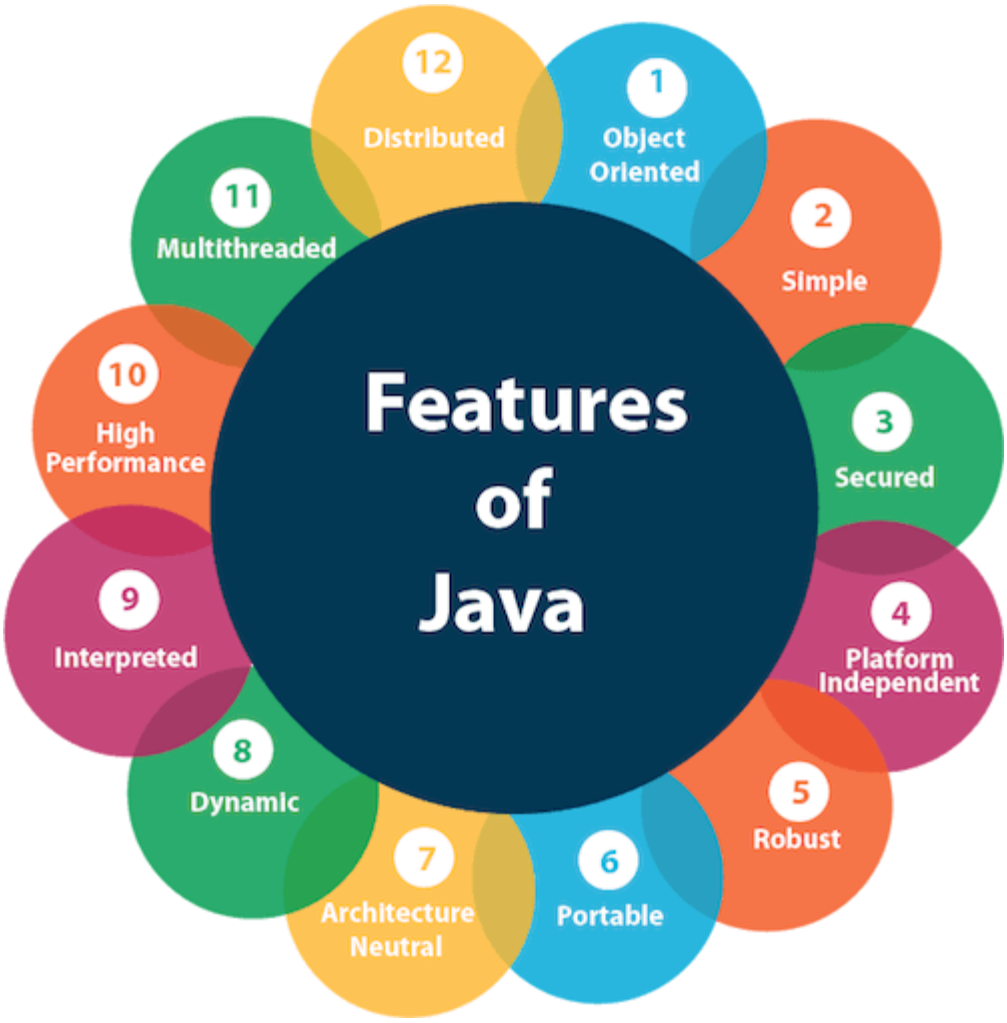
1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan 1996)
3. JDK 1.1 (19th Feb 1997)

4. J2SE 1.2 (8th Dec 1998)
5. J2SE 1.3 (8th May 2000)
6. J2SE 1.4 (6th Feb 2002)
7. J2SE 5.0 (30th Sep 2004)
8. Java SE 6 (11th Dec 2006)
9. Java SE 7 (28th July 2011)
10. Java SE 8 (18th Mar 2014)
11. Java SE 9 (21st Sep 2017)
12. Java SE 10 (20th Mar 2018)
13. Java SE 11 (September 2018)
14. Java SE 12 (March 2019)
15. Java SE 13 (September 2019)
16. Java SE 14 (Mar 2020)
17. Java SE 15 (September 2020)
18. Java SE 16 (Mar 2021)
19. Java SE 17 (September 2021)
20. Java SE 18 (to be released by March 2022)

Features of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as Java buzzwords.

A list of the most important features of the Java language is given below.



Simple:
Java is very easy to learn and easy to understand.

Object-oriented
Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior. Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules. Basic concepts of OOPs are Object, Class, Inheritance, Polymorphism, Abstraction, Encapsulation

Platform Independent
Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

Secured
Java is best known for its security. With Java, we can develop virus-free systems. Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

Robust

- It uses strong memory management.
- Avoids security problems.

- provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral
Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

Portable
Java is portable because the Java bytecode can be carried to any platform. It doesn't require any implementation.

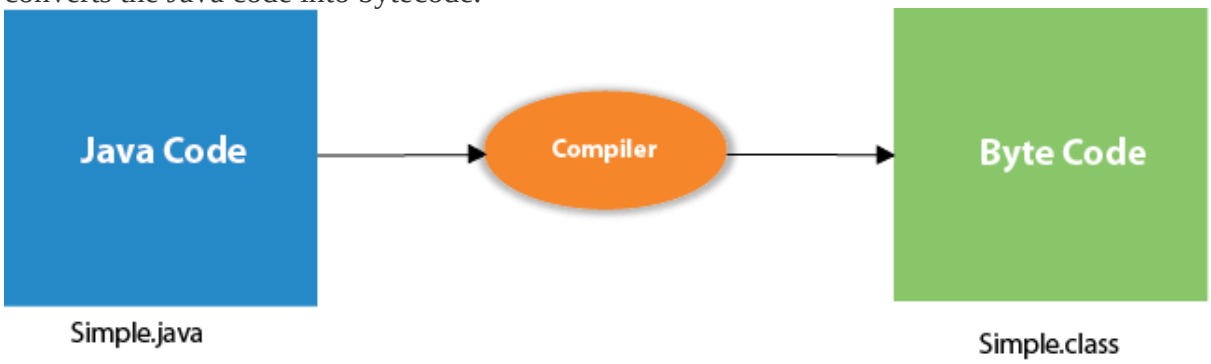
High-performance
Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code.

Distributed
Java is distributed because it facilitates users to create distributed applications in Java. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

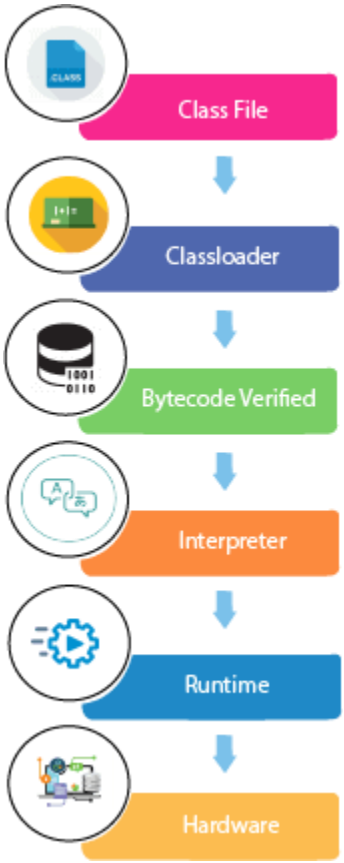
Multi-threaded
A thread is like a separate program, executing concurrently. Java programs handle many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

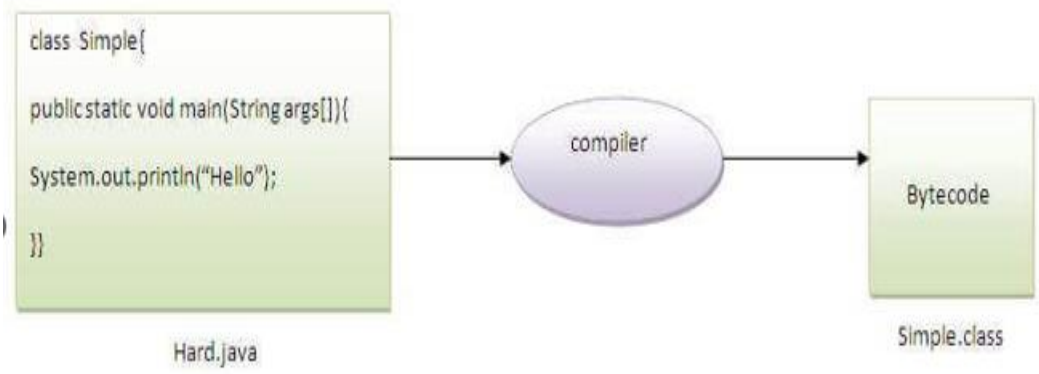
Dynamic
Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand.

What happens at compile time?
At compile time, the Java file is compiled by Java Compiler (It does not interact with OS) and converts the Java code into bytecode.



What happens at runtime?
At runtime, the following steps are performed:





JVM

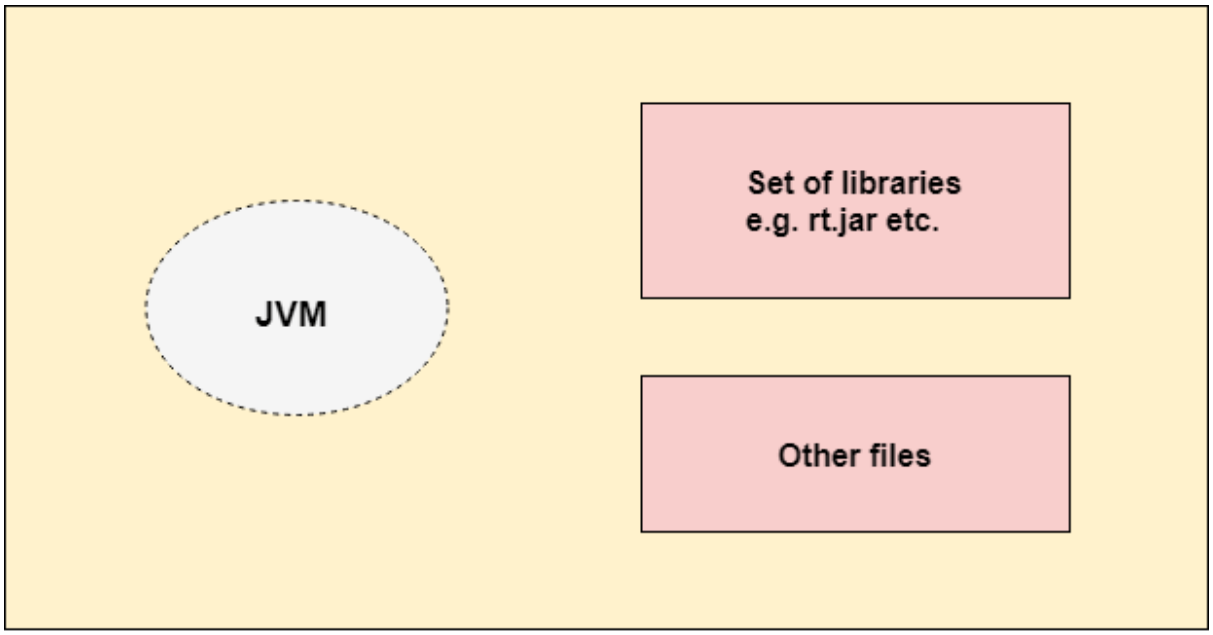
JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode. JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries and other files that JVM uses at runtime. The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



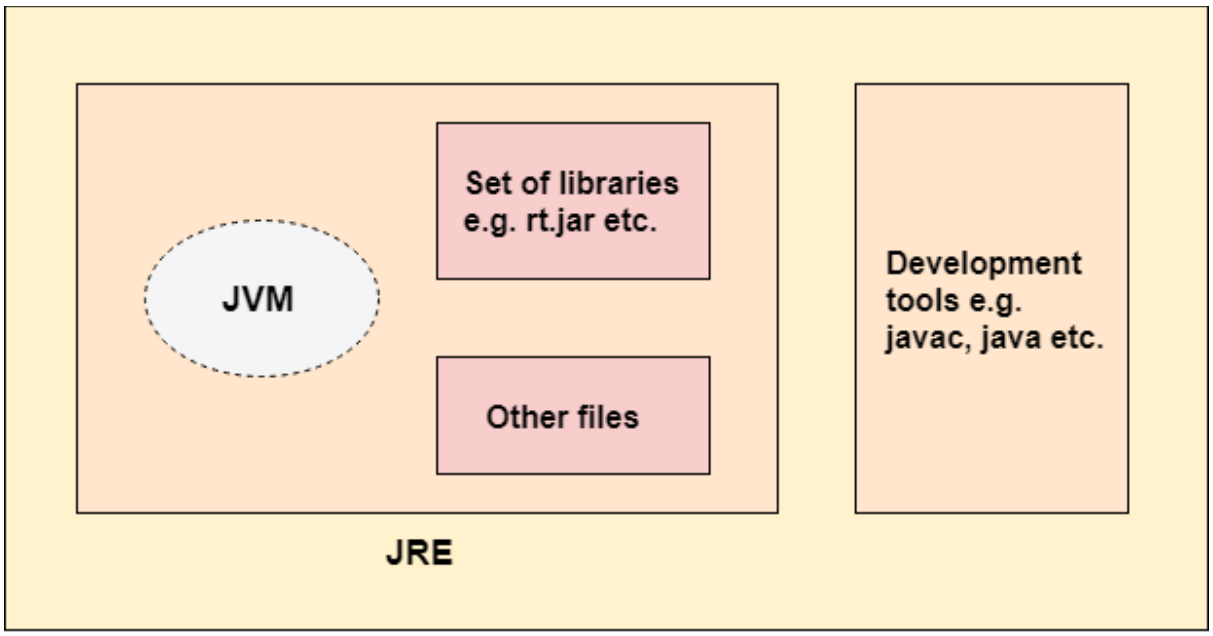
JRE

JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools. JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

JVM is

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

The JVM performs following operation:

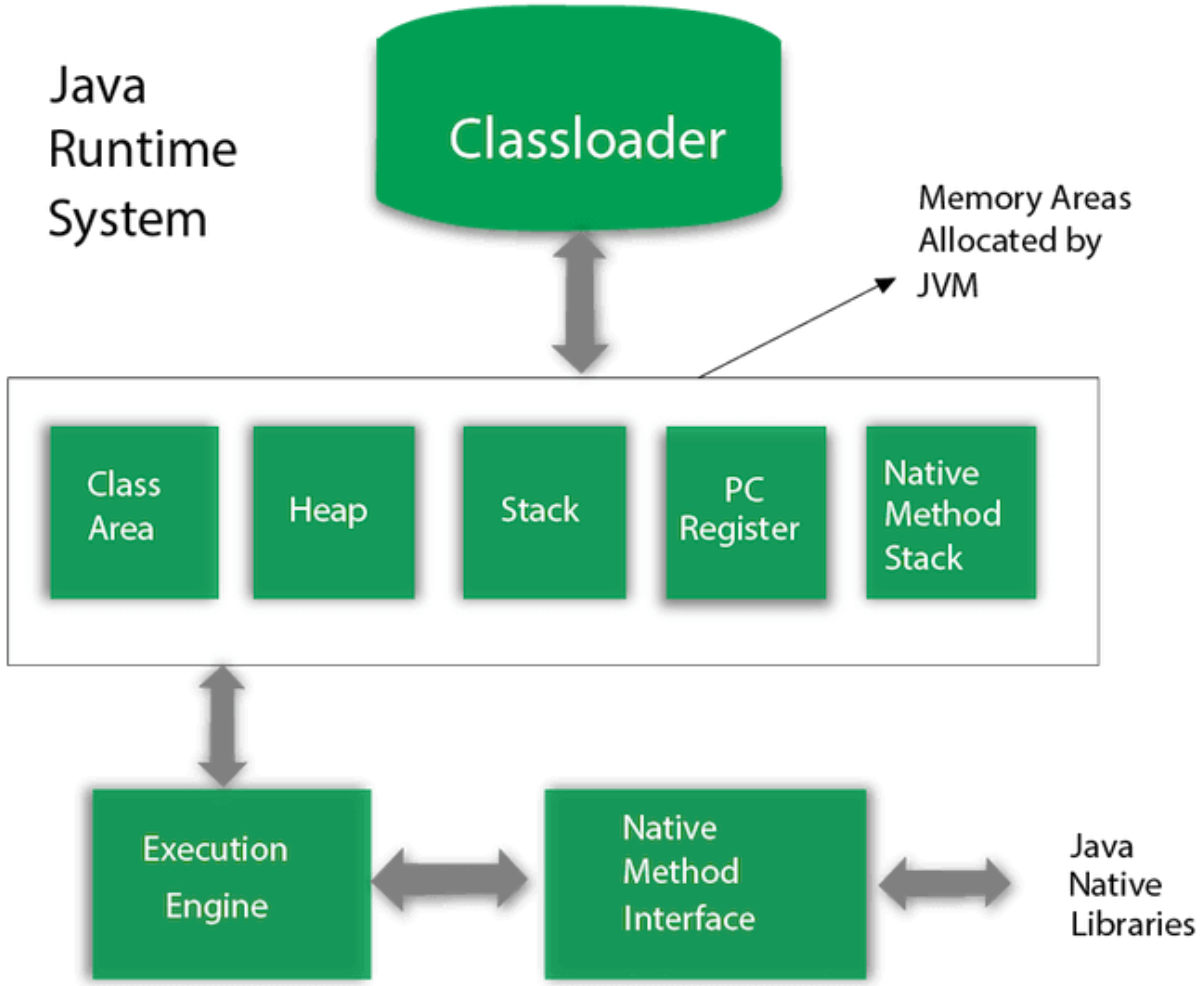
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

JVM Architecture

Let's understand the internal architecture of JVM. It contains class loader, memory area, execution engine etc.



1) Classloader

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader**: This is the first classloader which is the super class of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside *$JAVA_HOME/jre/lib/ext* directory.
3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap

It is the runtime data area in which objects are allocated.

4) Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as thread. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack

It contains all the native methods used in the application.

7) Execution Engine contains

1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces

the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

8) Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

Java Variables

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type. A variable is the name of a reserved area allocated in memory. Variable is a name of memory location. There are three types of variables in java: local, instance and static.

Types of Variables

There are three types of variables in Java.

- local variable
- instance variable
- static variable

1) Local Variable

A variable declared inside the body of the method is called local variable. This variable only within that method and the other methods in the class aren't even aware that the variable exists. A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static. It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local and it can be created a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example to understand the types of variables in java

```
1. public class A
2. {
3.     static int m=100;//static variable
4.     void method()
```

```
    {
        int n=90;//local variable
    }
    public static void main(String args[])
    {
        int data=50;//instance variable
    }
}//end of class
```

Java Variable Example: Add Two Numbers

```
public class Simple{
public static void main(String[] args){
int a=10;
int b=10;
int c=a+b;
System.out.println(c);
}
}
```

Output:

```
20
```

Java Variable Example: Widening

```
public class Simple{
public static void main(String[] args){
int a=10;
float f=a;
System.out.println(a);
System.out.println(f);
}}
```

Output:

```
10
10.0
```

Java Variable Example: Narrowing (Typecasting)

```
public class Simple{
public static void main(String[] args){
float f=10.5f;
//int a=f;//Compile time error
```

```
5. int a=(int)f;
6. System.out.println(f);
7. System.out.println(a);
8. }}
```

Output:

```
10.5
10
```

Java Variable Example: Overflow

```
1. class Simple{
2. public static void main(String[] args){
3. //Overflow
4. int a=130;
5. byte b=(byte)a;
6. System.out.println(a);
7. System.out.println(b);
8. }}
```

Output:

```
130
-126
```

Java Variable Example: Adding Lower Type

```
1. class Simple{
2. public static void main(String[] args){
3. byte a=10;
4. byte b=10;
5. //byte c=a+b;//Compile Time Error: because a+b=20 will be int
6. byte c=(byte)(a+b);
7. System.out.println(c);
8. }}
```

Output:

```
20
```

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

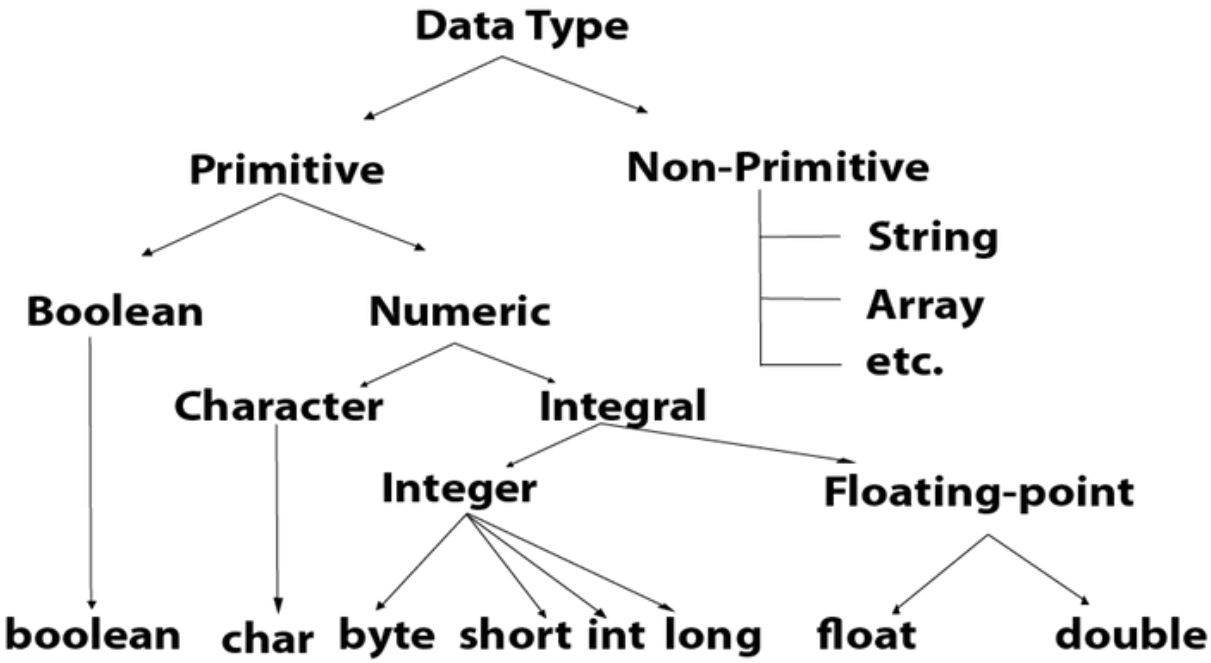
1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.

2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces and Arrays.

Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language. Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:



Data Type	Default Value	Default size
Boolean	False	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte
Short	0	2 byte
Int	0	4 byte
Long	0L	8 byte

Float	0.0f	4 byte
Double	0.0d	8 byte

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions. The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example: Boolean one = false

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0. The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

byte a = 10, byte b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0. The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

short s = 10000, short r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its minimum value is -2,147,483,648and maximum value is 2,147,483,647. Its default value is 0. The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

1. int a = 100000, int b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. The long data type is used when you need a range of values more than those provided by int.

Example:

1. long a = 100000L, long b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example:

```
1. float f1 = 234.5f
```

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

```
1. double d1 = 12.3
```

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example:

```
1. char letterA = 'A'
```

Unicode System

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Operators in Java

Operator in Java is a symbol that is used to perform operations. There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java Operator Precedence

Operator Type	Category	Precedence
Unary	Postfix	*expr*++ *expr*--
	Prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	Multiplicative	* / %
	Additive	+ -
Shift	Shift	<< >> >>>
Relational	Comparison	< > <= >= instanceof
	Equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	Ternary	? :
Assignment	Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Java Unary Operator Example: ++ and --

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int x=10;
4. System.out.println(x++);//10 (11)
5. System.out.println(++x);//12
6. System.out.println(x--);//12 (11)
7. System.out.println(--x);//10
8. }}
```

Java Unary Operator Example: ~ and !

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=-10;
5. boolean c=true;
6. boolean d=false;
7. System.out.println(~a);//-11 (minus of total positive value which starts from 0)
8. System.out.println(~b);//9 (positive of total minus, positive starts from 0)
9. System.out.println(!c);//false (opposite of boolean value)
10. System.out.println(!d);//true
11. }}
```

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations. Java Arithmetic Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. System.out.println(a+b);//15
6. System.out.println(a-b);//5
7. System.out.println(a*b);//50
8. System.out.println(a/b);//2
9. System.out.println(a%b);//0
10. }}
```

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times. Java Left Shift Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10<<2);//10*2^2=10*4=40
4. System.out.println(10<<3);//10*2^3=10*8=80
5. System.out.println(20<<2);//20*2^2=20*4=80
6. System.out.println(15<<4);//15*2^4=15*16=240
7. }}
```

Java Right Shift Operator

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand. Java Right Shift Operator Example.

```
1. public OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10>>2);//10/2^2=10/4=2
4. System.out.println(20>>2);//20/2^2=20/4=5
5. System.out.println(20>>3);//20/2^3=20/8=2
6. }}
```

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true. The bitwise & operator always checks both conditions whether first condition is true or false.

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a<b&&a<c);//false && true = false
7. System.out.println(a<b&a<c);//false & true = false
8. }}
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false. The bitwise | operator always checks both conditions whether first condition is true or false.

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a>b||a<c);//true || true = true
7. System.out.println(a>b|a<c);//true | true = true
8. //|| vs |
9. System.out.println(a>b||a++<c);//true || true = true
10. System.out.println(a);//10 because second condition is not checked
11. System.out.println(a>b|a++<c);//true | true = true
12. System.out.println(a);//11 because second condition is checked
13. }}
```

Java Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands. Java Ternary Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=2;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Output:

```
2
```

Another Example:

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Output:

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left. Java Assignment Operator Example.

```
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
```

Java Assignment Operator Example

```
1. int a=10;
2. a+=3;//10+3
3. a-=4;//13-4
4. a*=2;//9*2
5. a/=2;//18/2
```

Java Assignment Operator Example: Adding short

```
1. short a=10;
2. short b=10;
3. //a+=b;//a=a+b internally so fine
4. a=a+b;//Compile time error because 10+10=20 now int
```

After type cast:

```
1. short a=10;
2. short b=10;
3. a=(short)(a+b);//20 which is int now converted to short
```

Java Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

List of Java Keywords

abstract, boolean,break, byte, case, catch, char, class, continue, default, do, double,if, else,enum,extends, final, finally, float, for, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. Java provides three types of control flow statements.

1. Decision Making statements
    - if statements
    - switch statement
2. Loop statements
    - do while loop
    - while loop
    - for loop
    - for-each loop
3. Jump statements
    - break statement
    - continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

Java If-else Statement

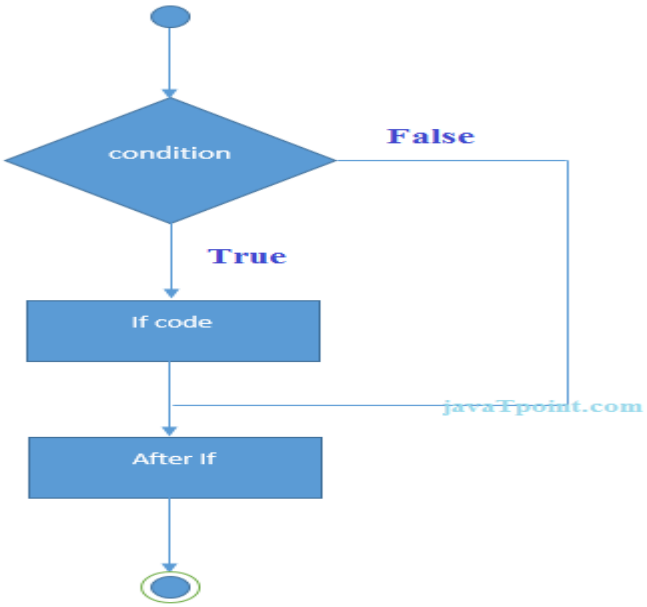
The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in Java.

- **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:**

```
1. if(condition)
2. {
3. //code to be executed
4. }
```



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:

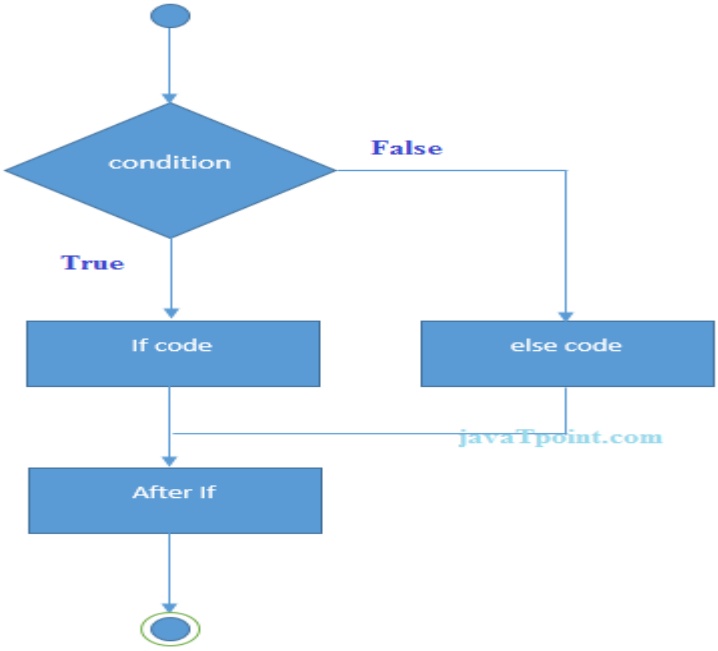
```
Age is greater than 18
```

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
```

5. }



Example:

```
1. //A Java Program to demonstrate the use of if-else statement.
2. //It is a program of odd and even number.
3. public class IfElseExample {
4. public static void main(String[] args) {
5.     //defining a variable
6.     int number=13;
7.     //Check if the number is divisible by 2 or not
8.     if(number%2==0){
9.         System.out.println("even number");
10.    }else{
11.        System.out.println("odd number");
12.    }
13. }
14. }
```

Output:

```
odd number
```

Using Ternary Operator

We can also use ternary operator (? :) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned. **Example:**

```
1. public class IfElseTernaryExample {
2. public static void main(String[] args) {
3.     int number=13;
4.     //Using ternary operator
5.     String output=(number%2==0)?"even number":"odd number";
6.     System.out.println(output);
7. }
8. }
```

Output:

```
odd number
```

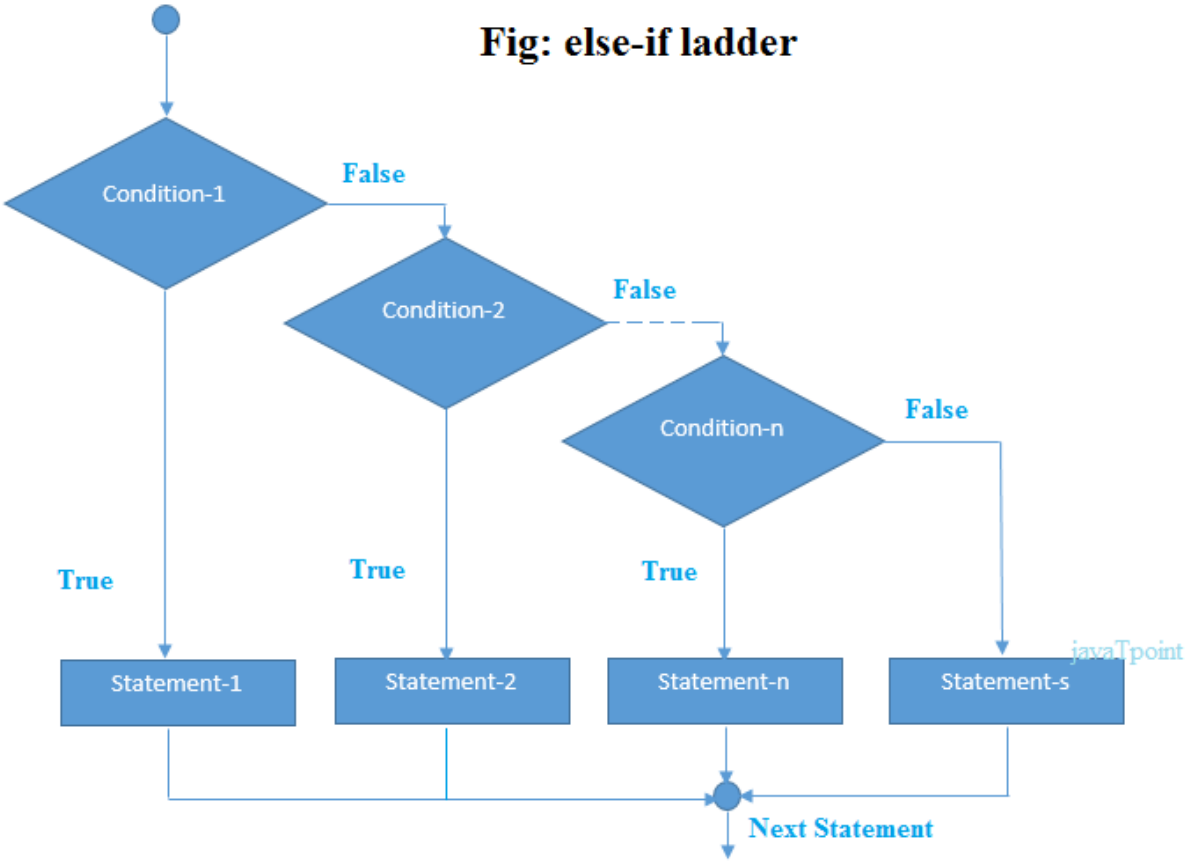
Java if-else-if ladder Statement

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

Syntax:

```
1. if(condition1){
2. //code to be executed if condition1 is true
3. }else if(condition2){
4. //code to be executed if condition2 is true
5. }
6. else if(condition3){
7. //code to be executed if condition3 is true
8. }
9. ...
10. else{
11. //code to be executed if all the conditions are false
12. }
```

Fig: else-if ladder



Example:

```
1. //Java Program to demonstrate the use of If else-if ladder.
2. //It is a program of grading system for fail, D grade, C grade, B grade, A grade and A
   +.
3. public class IfElseIfExample {
4. public static void main(String[] args) {
5.     int marks=65;
6.
7.     if(marks<50){
8.         System.out.println("fail");
9.     }
10.    else if(marks>=50 && marks<60){
11.        System.out.println("D grade");
12.    }
13.    else if(marks>=60 && marks<70){
14.        System.out.println("C grade");
15.    }
16.    else if(marks>=70 && marks<80){
```

```
17.         System.out.println("B grade");
18.     }
19.     else if(marks>=80 && marks<90){
20.         System.out.println("A grade");
21.     }else if(marks>=90 && marks<100){
22.         System.out.println("A+ grade");
23.     }else{
24.         System.out.println("Invalid!");
25.     }
26. }
27. }
```

Output:

```
C grade
```

Program to check POSITIVE, NEGATIVE or ZERO:

```
1.  public class PositiveNegativeExample {
2.  public static void main(String[] args) {
3.      int number=-13;
4.      if(number>0){
5.      System.out.println("POSITIVE");
6.      }else if(number<0){
7.      System.out.println("NEGATIVE");
8.      }else{
9.      System.out.println("ZERO");
10.    }
11. }
12. }
```

Output:

```
NEGATIVE
```

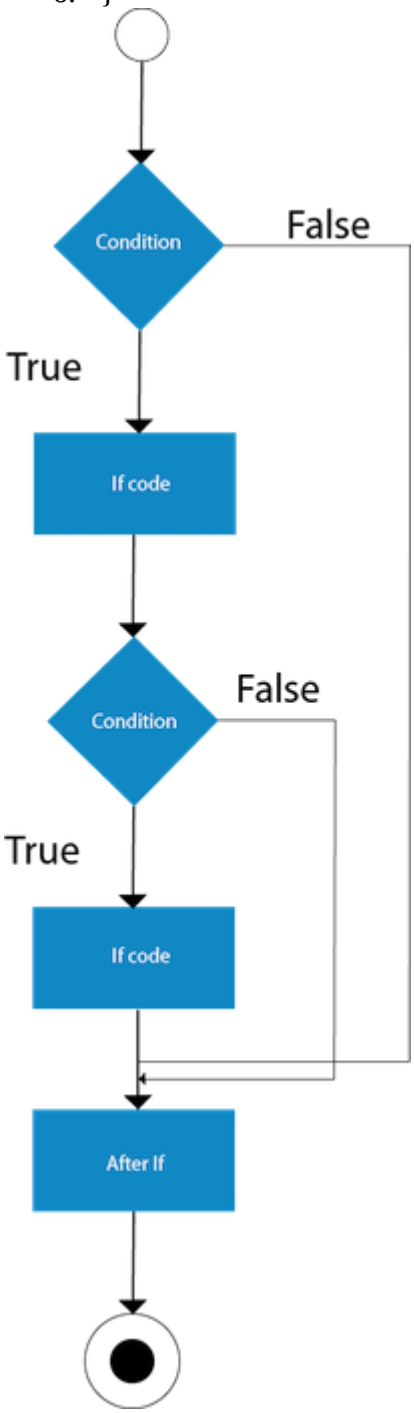
Java Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

Syntax:

```
1.  if(condition){
2.      //code to be executed
```

```
3.          if(condition){
4.            //code to be executed
5.     }
6.  }
```



Example:

```
1.  //Java Program to demonstrate the use of Nested If Statement.
2.  public class JavaNestedIfExample {
3.  public static void main(String[] args) {
```

```
4.      //Creating two variables for age and weight
5.      int age=20;
6.      int weight=80;
7.      //applying condition on age and weight
8.      if(age>=18){
9.         if(weight>50){
10.            System.out.println("You are eligible to donate blood");
11.        }
12.    }
13. }}
```

Output:

```
You are eligible to donate blood
```

Example 2:

```
1.  //Java Program to demonstrate the use of Nested If Statement.
2.  public class JavaNestedIfExample2 {
3.  public static void main(String[] args) {
4.     //Creating two variables for age and weight
5.     int age=25;
6.     int weight=48;
7.     //applying condition on age and weight
8.     if(age>=18){
9.        if(weight>50){
10.          System.out.println("You are eligible to donate blood");
11.       } else{
12.          System.out.println("You are not eligible to donate blood");
13.       }
14.    } else{
15.      System.out.println("Age must be greater than 18");
16.    }
17. }  }
```

Output:

```
You are not eligible to donate blood
```

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

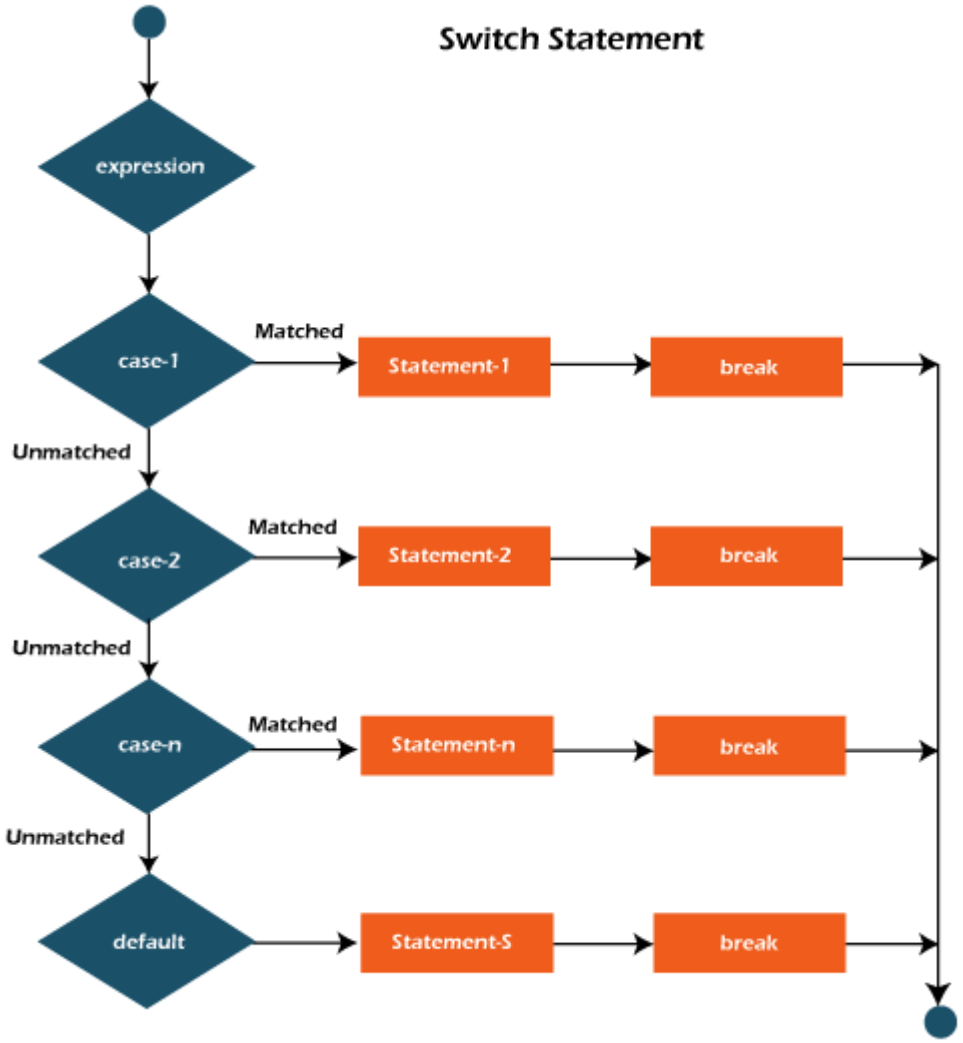
- There can be *one or N number of case values* for a switch expression.
- The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.
- The case values must be *unique*. In case of duplicate value, it renders compile-time error.
- 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;
}
```

Flowchart of Switch Statement



Example:

SwitchExample.java

```
1.  public class SwitchExample {
2.  public static void main(String[] args) {
3.      //Declaring a variable for switch expression
4.      int number=20;
5.      //Switch expression
6.      switch(number){
7.      //Case statements
8.      case 10: System.out.println("10");
9.      break;
10.     case 20: System.out.println("20");
11.     break;
```

```
12.     case 30: System.out.println("30");
13.     break;
14.     //Default case statement
15.     default:System.out.println("Not in 10, 20 or 30");
16.     }
17. }
18. }
```

Output:

```
20
```

Java Switch Statement with String

Java allows us to use strings in switch expression since Java SE 7. The case statement should be string literal.

Example:

```
1.  //Java Program to demonstrate the use of Java Switch
2.  //statement with String
3.  public class SwitchStringExample {
4.  public static void main(String[] args) {
5.      //Declaring String variable
6.      String levelString="Expert";
7.      int level=0;
8.      //Using String in Switch expression
9.      switch(levelString){
10.     //Using String Literal in Switch case
11.     case "Beginner": level=1;
12.     break;
13.     case "Intermediate": level=2;
14.     break;
15.     case "Expert": level=3;
16.     break;
17.     default: level=0;
18.     break;
19.     }
20.     System.out.println("Your Level is: "+level);
21. }
22. }
```

Test it Now

Output:

```
Your Level is: 3
```

Java Wrapper in Switch Statement

Java allows us to use four wrapper classes: Byte, Short, Integer and Long in switch statement.

Example:

```
1.  //Java Program to demonstrate the use of Wrapper class
2.  //in switch statement
3.  public class WrapperInSwitchCaseExample {
4.      public static void main(String args[])
5.      {
6.          Integer age = 18;
7.          switch (age)
8.          {
9.              case (16):
10.                 System.out.println("You are under 18.");
11.                 break;
12.             case (18):
13.                 System.out.println("You are eligible for vote.");
14.                 break;
15.             case (65):
16.                 System.out.println("You are senior citizen.");
17.                 break;
18.             default:
19.                 System.out.println("Please give the valid age.");
20.                 break;
21.         }
22.     }
23. }
```

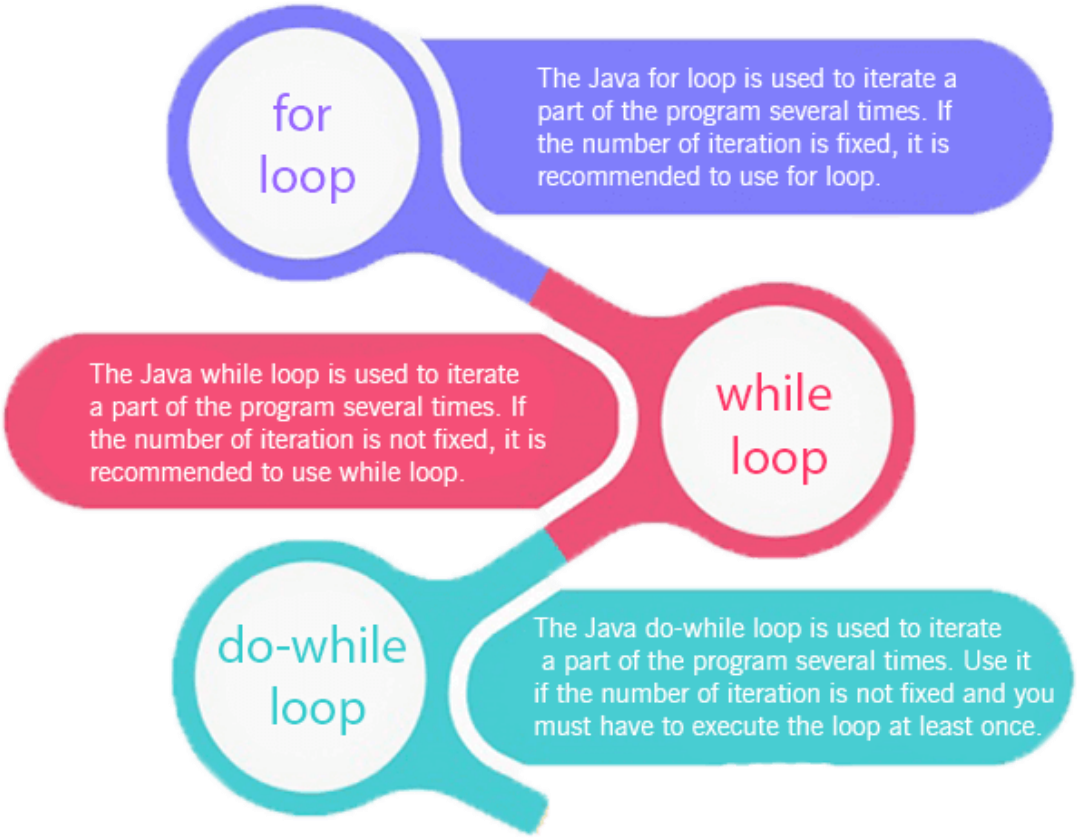
Test it Now

Output:

```
You are eligible for vote.
```

Loops in Java

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is **fixed**, it is recommended to use for loop. There are three types of for loops in Java.



- Simple for Loop
- For-each or Enhanced for Loop
- Labeled for Loop

Java Simple for Loop

A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

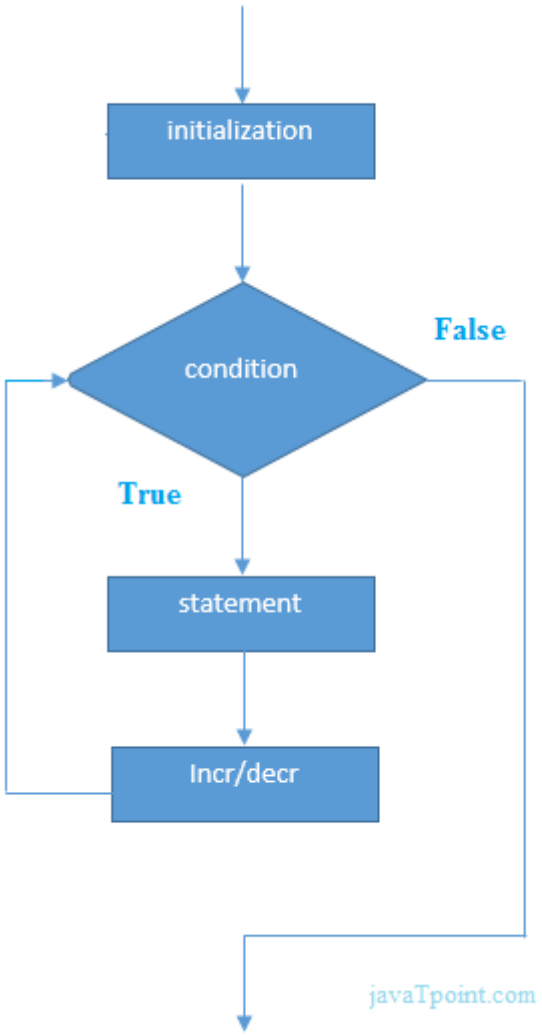
1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.

3. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.
4. **Statement**: The statement of the loop is executed each time until the second condition is false.

Syntax:

```
1. for(initialization; condition; increment/decrement){
2. //statement or code to be executed
3. }
```

Flowchart:



Example:

```
1. //Java Program to demonstrate the example of for loop
2. //which prints table of 1
```

```
3. public class ForExample {
4. public static void main(String[] args) {
5.     //Code of Java for loop
6.     for(int i=1;i<=10;i++){
7.         System.out.println(i);
8.     }
9. }
10. }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Java Nested for Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

```
1. public class NestedForExample {
2. public static void main(String[] args) {
3. //loop of i
4. for(int i=1;i<=3;i++){
5. //loop of j
6. for(int j=1;j<=3;j++){
7.         System.out.println(i+" "+j);
8. }//end of i
9. }//end of j
10. }
11. }
```

Output:

```
1 1
1 2
1 3
2 1
2 2
```

```
2 3
3 1
3 2
3 3
```

Java for-each Loop

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. It works on the basis of elements and not the index. It returns element one by one in the defined variable.

Syntax:

```
for(data_type variable : array_name)
{
//code to be executed
}
```

```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample {
public static void main(String[] args) {
    //Declaring an array
    int arr[]={12,23,44,56,78};
    //Printing array using for-each loop
    for(int i:arr){
        System.out.println(i);
    }
}
}
```

Output:

```
12
23
44
56
78
```

Java Labeled For Loop

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful while using the nested for loop as we can break/continue specific for loop. **Note: The break and continue keywords breaks or continues the innermost for loop respectively.**

Syntax:

```
labelname: for(initialization; condition; increment/decrement)
{
//code to be executed
}
```

```
//A Java program to demonstrate the use of labeled for loop
public class LabeledForExample {
public static void main(String[] args) {
    //Using Label for outer and for loop
aa: for(int i=1;i<=3;i++)
{
bb: for(int j=1;j<=3;j++)
{
        if(i==2&&j==2)
        {
            break aa;
        }
        System.out.println(i+" "+j);
        }
    }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
3 3
```

Java for Loop vs while Loop vs do-while Loop

Comparison	for loop	while loop	do-while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.

Syntax	for(init;condition;incr/decr){
	// code to be executed
	}
		while(condition){
		//code to be executed
		}
			do{
			//code to be executed
			}while(condition);

Example	//for loop
	for(int i=1;i<=10;i++){
	System.out.println(i);
	}
		//while loop
		int i=1;
		while(i<=10){
		System.out.println(i);
		i++;
		}
			//do-while loop
			int i=1;
			do{
			System.out.println(i);
			i++;
			}while(i<=10);

Syntax for infinitive loop	for(;;){
	//code to be executed
	}
		while(true){
		//code to be executed
		}
			do{
			//code to be executed
			}while(true);

Java While Loop

The Java *while loop* is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops. The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while loop.

Syntax:

```
1. while (condition)
2. {
3. //code to be executed
4. Increment / decrement statement
5. }
```

The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. When the condition becomes false, we exit the while loop.

Example:

i <=100

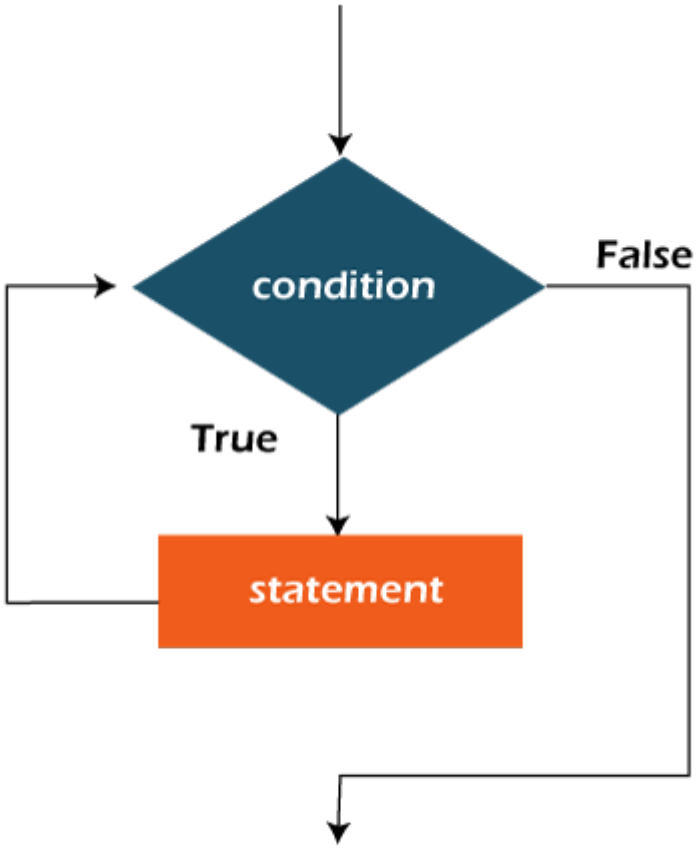
2. Update expression: Every time the loop body is executed, this expression increments or decrements loop variable.

Example:

i++;

Flowchart of Java While Loop

Here, the important thing about while loop is that, sometimes it may not even execute. If the condition to be tested results into false, the loop body is skipped and first statement after the while loop will be executed.



Example:

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

```
1. public class WhileExample {
2. public static void main(String[] args) {
3.     int i=1;
4.     while(i<=10){
5.         System.out.println(i);
6.     i++;
7.     }
8. }
9. }
```

Output:

```
1
2
3
4
5
```

```
6
7
8
9
10
```

Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop. Java do-while loop is called an **exit control loop**. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
1. Do
2. {
3. //code to be executed / loop body
4. //update statement
5. }while (condition);
```

The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. As soon as the condition becomes false, loop breaks automatically.

Example:

i <=100

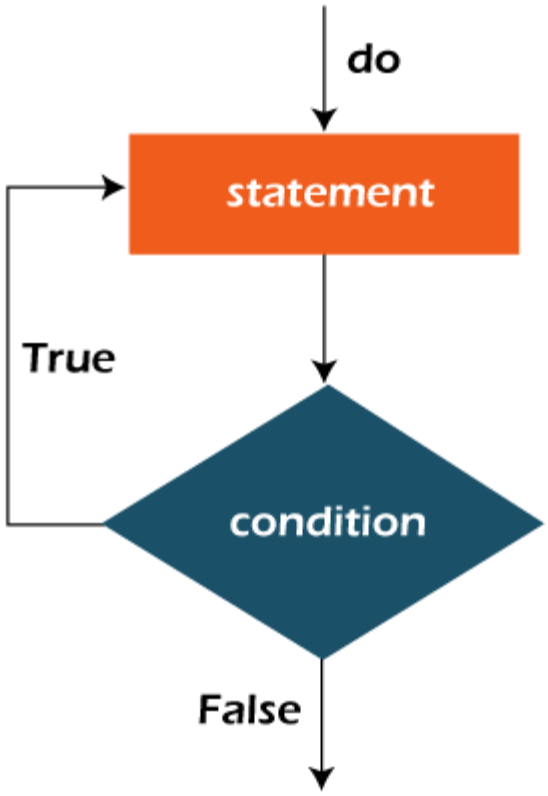
2. Update expression: Every time the loop body is executed, the this expression increments or decrements loop variable.

Example:

i++;

Note: The do block is executed at least once, even if the condition is false.

Flowchart of do-while loop:



Example:

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

DoWhileExample.java

```
1. public class DoWhileExample {
2. public static void main(String[] args) {
3.     int i=1;
4.     do{
5.         System.out.println(i);
6.     i++;
7.     }while(i<=10);
8. }
9. }
```

Output:

```
1
2
```

```
3
4
5
6
7
8
9
10
```

Java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop. The Java *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only. We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

```
1. jump-statement;
2. continue;
```

Java Continue Statement Example

```
1. //Java Program to demonstrate the use of continue statement
2. //inside the for loop.
3. public class ContinueExample {
4. public static void main(String[] args) {
5.     //for loop
6.     for(int i=1;i<=10;i++){
7.         if(i==5){
8.             //using continue statement
9.             continue;//it will skip the rest statement
10.        }
11.        System.out.println(i);
12.    }
13. }
14. }
```

Output:

```
1
2
3
4
6
7
```

```
8
9
10
```

As you can see in the above output, 5 is not printed on the console. It is because the loop is continued when it reaches to 5.

Java Continue Statement with Inner Loop

It continues inner loop only if you use the continue statement inside the inner loop.

ContinueExample2.java

```
1. //Java Program to illustrate the use of continue statement
2. //inside an inner loop
3. public class ContinueExample2 {
4. public static void main(String[] args) {
5.         //outer loop
6.         for(int i=1;i<=3;i++){
7.             //inner loop
8.             for(int j=1;j<=3;j++){
9.                 if(i==2&&j==2){
10.                     //using continue statement inside inner loop
11.                     continue;
12.                 }
13.                 System.out.println(i+" "+j);
14.             }
15.         }
16. }
17. }
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

Java Comments

The Java comments are the statements in a program that are not executed by the compiler and interpreter.

Why do we use comments in a code?

- Comments are used to make the program more readable by adding the details of the code.
- It makes easy to maintain the code and to find the errors easily.
- The comments can be used to provide information or explanation about the variable, method, class, or any statement.
- It can also be used to prevent the execution of program code while testing the alternative code.

Types of Java Comments

There are three types of comments in Java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

1) Java Single Line Comment

The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements. Single line comments starts with two forward slashes **(//)**. Any text in front of // is not executed by Java.

Syntax:

```
1. //This is single line comment
```

2) Java Multi Line Comment

The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time (as it will be difficult to use single-line comments there). Multi-line comments are placed between /* and */. Any text between /* and */ is not executed by Java.

Syntax:

```
1. /*
2. This
3. is
4. multi line
5. comment
6. */
```

3) Java Documentation Comment

Documentation comments are usually used to write large programs for a project or software application as it helps to create documentation API. These APIs are needed for reference, i.e., which classes, methods, arguments, etc., are used in the code. To create documentation API, we need to use the **javadoc tool**. The documentation comments are placed between /** and */.

Syntax:

```
/**
*
*We can use various tags to depict the parameter
*or heading or author name
*We can also use HTML tags
*
*/
```

Are Java comments executable?

Ans: As we know, Java comments are not executed by the compiler or interpreter, however, before the lexical transformation of code in compiler, contents of the code are encoded into ASCII in order to make the processing easy.