Java Tutorial- by Javatpoint

Java Tutorial

Our core Java programming tutorial is designed for students and working professionals. Java is an object-oriented

, class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.

What is Java?

Java is a **programming language** and a **platform**. Java is a high level, robust, 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.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

Java Example

Let's have a quick look at Java programming example. A detailed description of Hello Java example is available in next page.

Simple.java

```java
class Simple{
    public static void main(String args[]){
     System.out.println("Hello Java");
    }
}
```

Test it Now

Application

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics

8. Games, etc.

Types of Java Applications

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

1) Standalone Application

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet

, JSP
, Struts
, Spring
, Hibernate
, JSF
, etc. technologies are used for creating web applications in Java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB

is used for creating enterprise applications.

4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, String

, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2) Java EE (Java Enterprise Edition)

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA

, etc.

3) Java ME (Java Micro Edition)

It is a micro platform that is dedicated to mobile applications.

4) JavaFX

It is used to develop rich internet applications. It uses a lightweight user interface API.

Prerequisite

To learn Java, you must have the basic knowledge of C/C++ programming language.

Audience

Our Java programming tutorial is designed to help beginners and professionals.

Problem

We assure that you will not find any problem in this Java tutorial. However, if there is any mistake, please post the problem in the contact form.

Do You Know?

- What is the difference between JRE and JVM?
- What is the purpose of JIT compiler?
- Can we save the java source file without any name?
- Why java uses the concept of Unicode system?

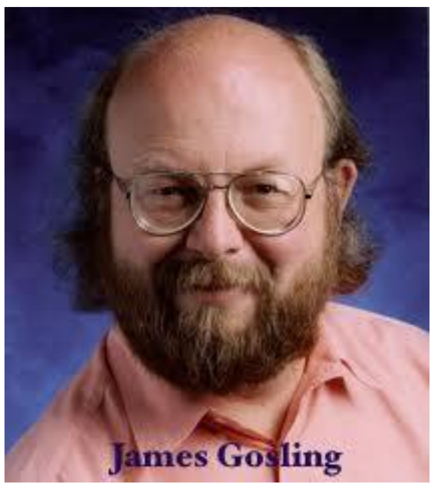
What will we learn in Basics of Java?

- History of Java
- Features of Java
- Hello Java Program
- Program Internal
- How to set path?
- Difference between JDK,JRE and JVM
- Internal Details of JVM
- Variable and Data Type
- Unicode System
- Operators

History of Java

The history of Java is very interesting. 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. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.



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.

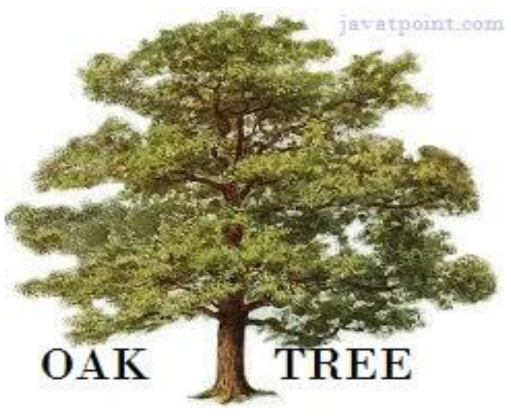
1) **James Gosling, Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.

Why Java was named as "Oak"?



5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.

6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java Programming named "Java"?

7) Why had they chose the name Java for Java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell, and fun to say.

According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.

8) Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have 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.

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)

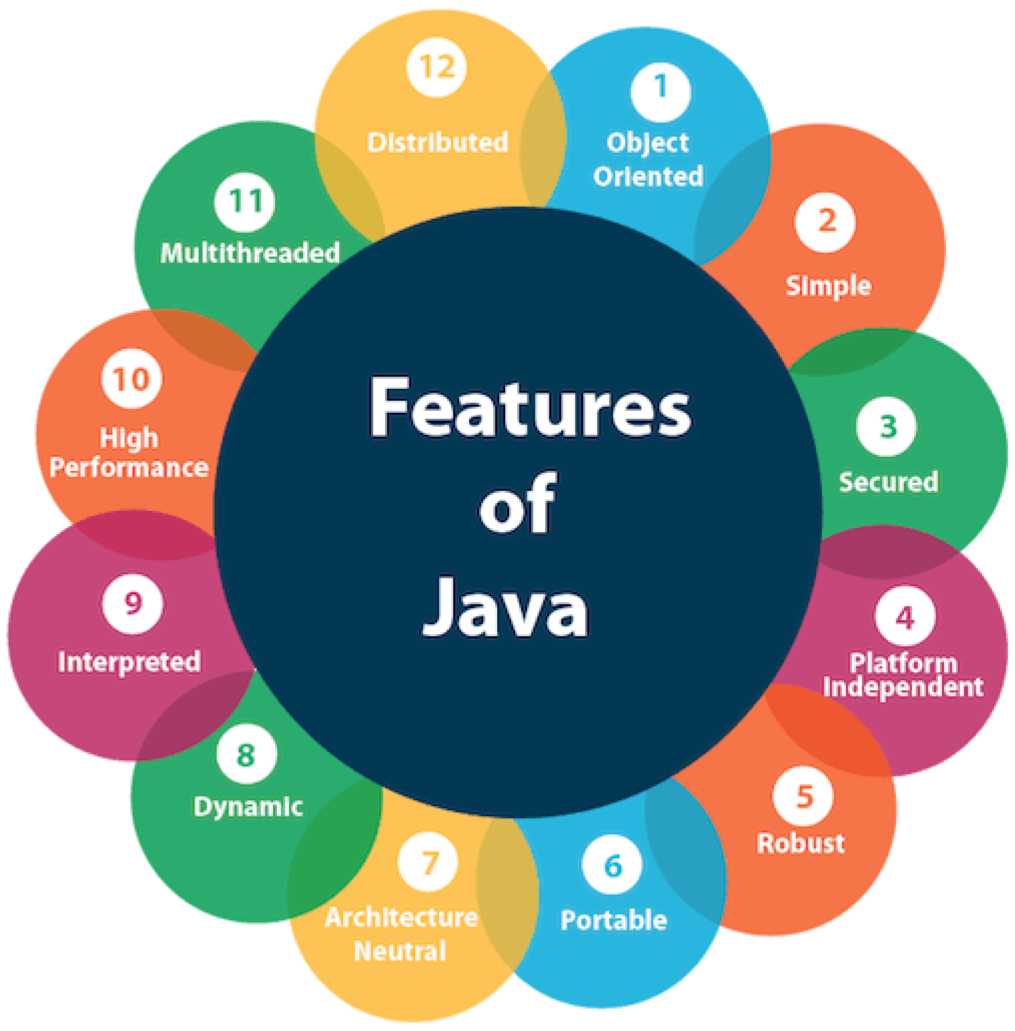
Since Java SE 8 release, the Oracle corporation follows a pattern in which every even version is release in March month and an odd version released in September month.

More Details on Java Versions.

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.



1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic

Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

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.

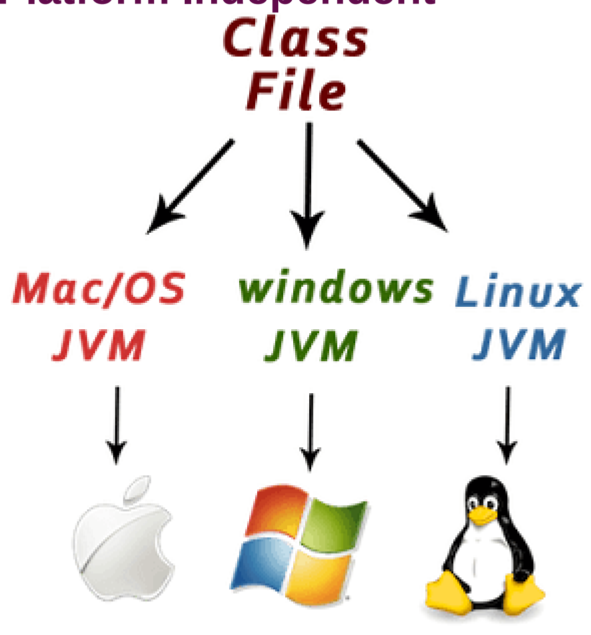
Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object
2. Class

3. Inheritance
4. Polymorphism
5. Abstraction
6. 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.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms. It has two components:

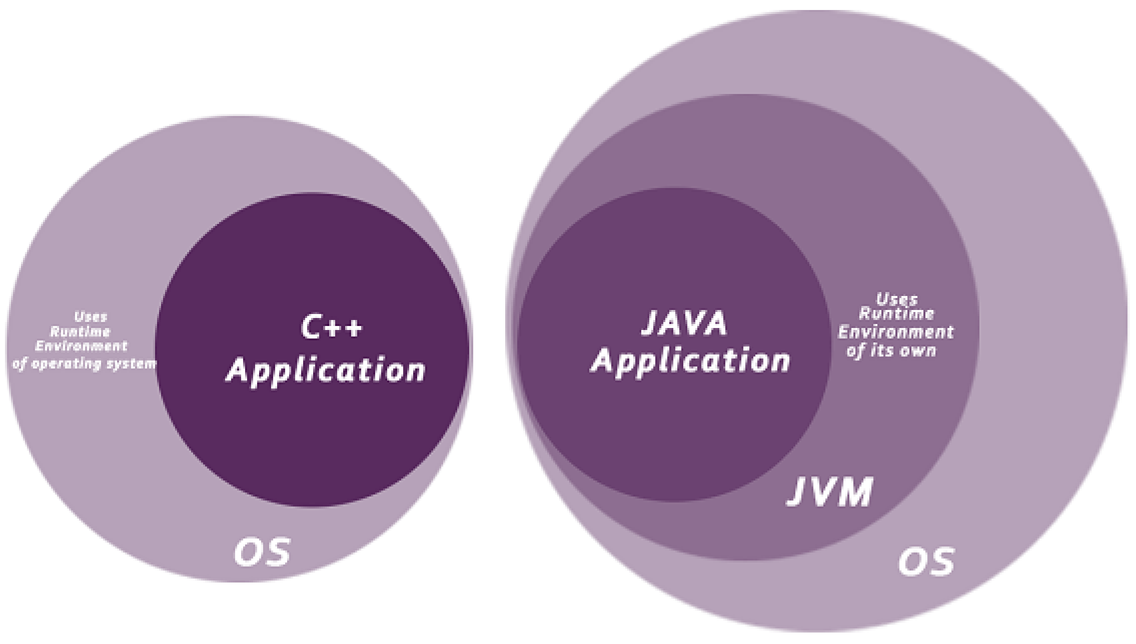
1. Runtime Environment
2. API(Application Programming Interface)

Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**


Uses Runtime Environment of operating system
C++ Application
OS
JAVA Application
Uses Runtime Environment of its own
JVM
OS

- **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access rights to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

Robust

The English mining of Robust is strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- Java 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.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable

Java is portable because it facilitates you to carry the Java bytecode 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. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. 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. We can write Java programs that deal with 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. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

C++ vs Java

There are many differences and similarities between the C++ programming language and Java. A list of top differences between C++ and Java are given below:

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write a pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comments.	Java supports documentation comment (/** ... */) to create

		documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not to override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
Inheritance Tree	C++ always creates a new inheritance tree.	Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.
Object-oriented	C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.

Note

- Java doesn't support default arguments like C++.
- Java does not support header files like C++. Java uses the import keyword to include different classes and methods.

C++ Program Example

File: main.cpp

```
#include <iostream>
using namespace std;
int main() {
   cout << "Hello C++ Programming";
   return 0;
}
```

Output:

```
Hello C++ Programming
```

Java Program Example

File: Simple.java

```
class Simple{
    public static void main(String args[]){
     System.out.println("Hello Java");
    }
}
```

Output:

```
Hello Java
```

First Java Program | Hello World Example

In this section, we will learn how to write the simple program of Java. We can write a simple hello Java program easily after installing the JDK.

To create a simple Java program, you need to create a class that contains the main method. Let's understand the requirement first.

The requirement for Java Hello World Example

For executing any Java program, the following software or application must be properly installed.

- Install the JDK if you don't have installed it, download the JDK and install it.
- Set path of the jdk/bin directory. http://www.javatpoint.com/how-to-set-path-in-java
- Create the Java program
- Compile and run the Java program

Creating Hello World Example

Let's create the hello java program:

```java
class Simple{
    public static void main(String args[]){
     System.out.println("Hello Java");
    }
}
```

Test it Now

Save the above file as Simple.java.

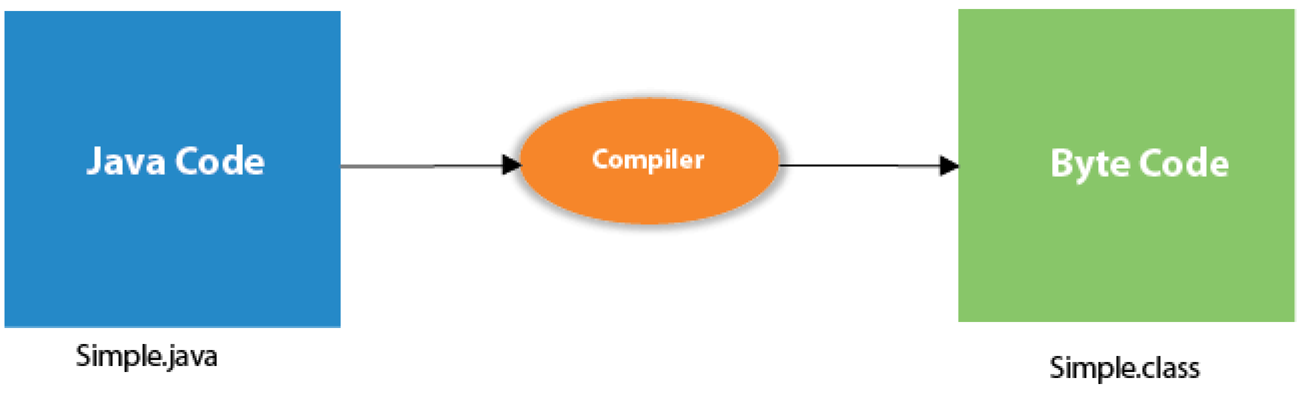
To compile:	javac Simple.java
To execute:	java Simple

Output:

```
Hello Java
```

Compilation Flow:

When we compile Java program using javac tool, the Java compiler converts the source code into byte code.



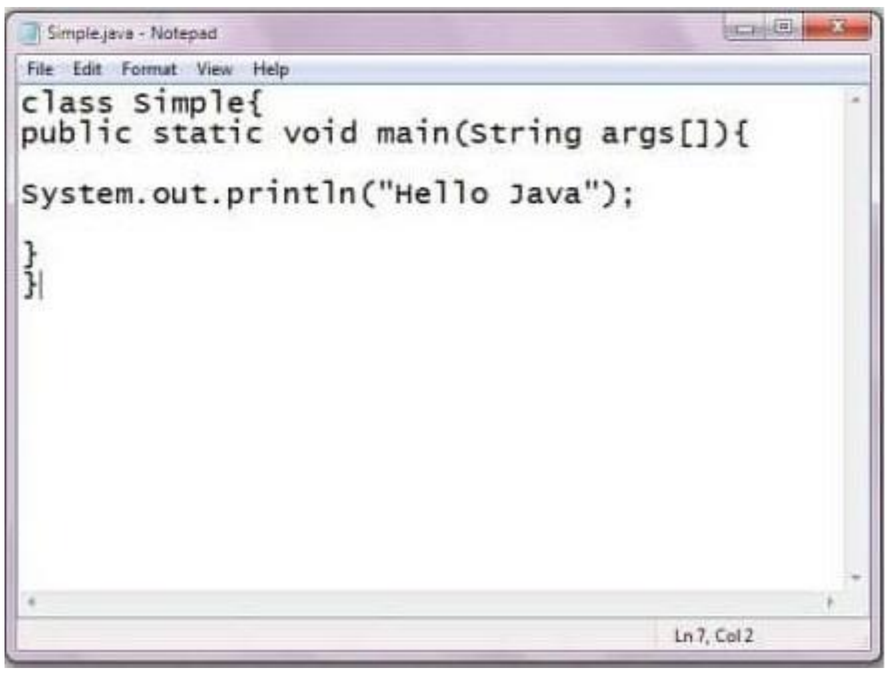
Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in Java.

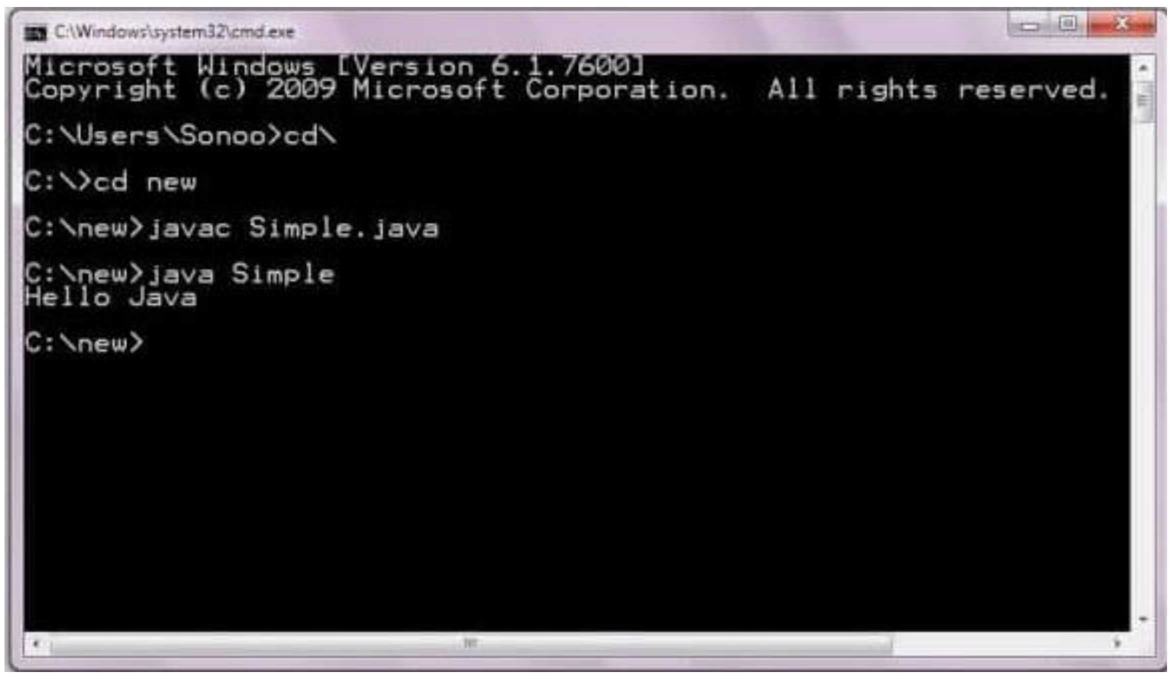
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** or **String args[]** is used for command line argument. We will discuss it in coming section.
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class. We will discuss the internal working of System.out.println() statement in the coming section.

To write the simple program, you need to open notepad by **start menu -> All Programs -> Accessories -> Notepad** and write a simple program as we have shownbelow:



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

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

}
}
```

As displayed in the above diagram, write the simple program of Java in notepad and saved it as Simple.java. In order to compile and run the above program, you need to open the command prompt by **start menu -> All Programs -> Accessories -> command prompt**. When we have done with all the steps properly, it shows the following output:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Sonoo>cd\

C:\>cd new

C:\new>javac Simple.java

C:\new>java Simple
Hello Java

C:\new>
```

To compile and run the above program, go to your current directory first; my current directory is c:\new. Write here:

To compile:	javac Simple.java
To execute:	java Simple

In how many ways we can write a Java program?

There are many ways to write a Java program. The modifications that can be done in a Java program are given below:

1) By changing the sequence of the modifiers, method prototype is not changed in Java.

Let's see the simple code of the main method.

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

2) The subscript notation in the Java array can be used after type, before the variable or after the variable.

Let's see the different codes to write the main method.

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

3) You can provide var-args support to the main() method by passing 3 ellipses (dots)

Let's see the simple code of using var-args in the main() method. We will learn about var-args later in the Java New Features chapter.

```
public static void main(String... args)
```

4) Having a semicolon at the end of class is optional in Java.

Let's see the simple code.

```
class A{
static public void main(String... args){
System.out.println("hello java4");
}
};
```

Valid Java main() method signature

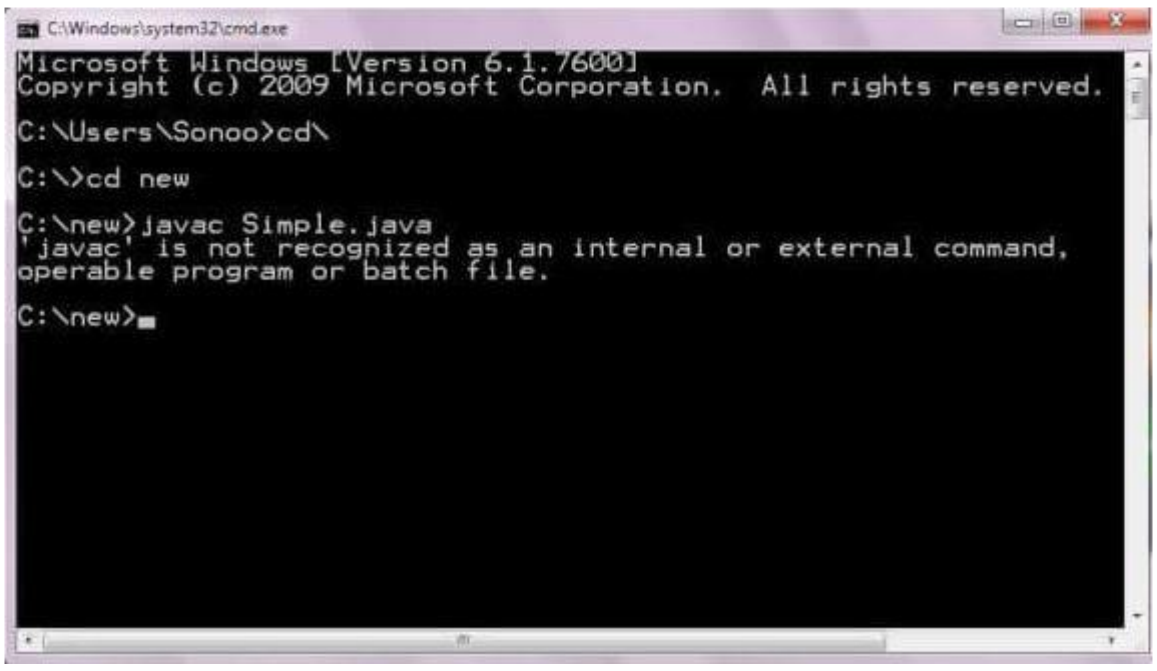
```
public static void main(String[] args)
public static void main(String []args)
public static void main(String args[])
public static void main(String... args)
static public void main(String[] args)
public static final void main(String[] args)
final public static void main(String[] args)
final strictfp public static void main(String[] args)
```

Invalid Java main() method signature

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

Resolving an error "javac is not recognized as an internal or external command"?

If there occurs a problem like displayed in the below figure, you need to set a path. Since DOS doesn't recognize javac and java as internal or external command. To overcome this problem, we need to set a path. The path is not required in a case where you save your program inside the JDK/bin directory. However, it is an excellent approach to set the path. Click here for How to set path in java.



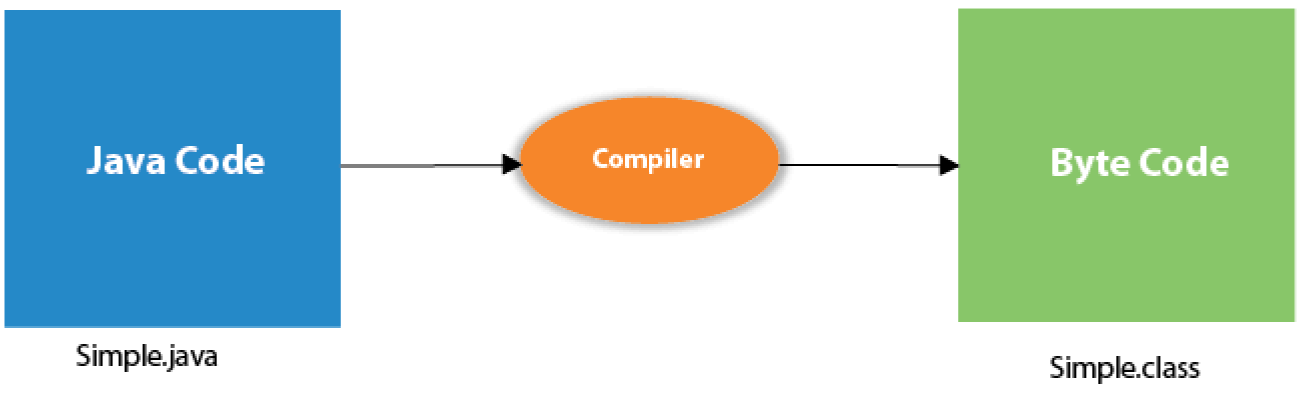
```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Sonoo>cd\

C:\>cd new

C:\new>javac Simple.java
'javac' is not recognized as an internal or external command,
operable program or batch file.

C:\new>
```

Internal Details of Hello Java Program

In the previous section, we have created Java Hello World program and learn how to compile and run a Java program. In this section, we are going to learn, what happens while we compile and run the Java program. Moreover, we will see some questions based on the first program.

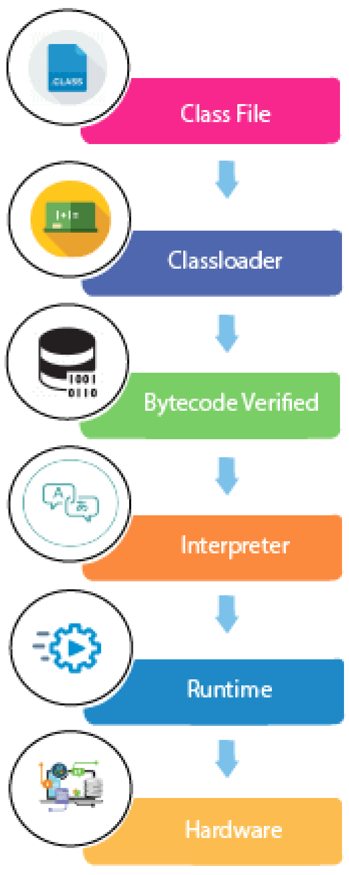
What happens at compile time?

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:



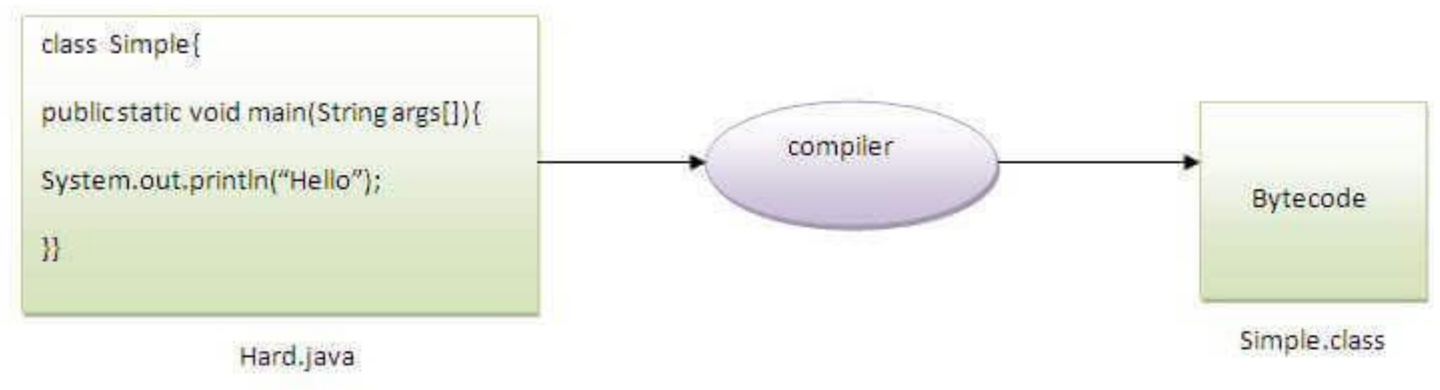
Classloader: It is the subsystem of JVM that is used to load class files.

Bytecode Verifier: Checks the code fragments for illegal code that can violate access rights to objects.

Interpreter: Read bytecode stream then execute the instructions.

Q) Can you save a Java source file by another name than the class name?

Yes, if the class is not public. It is explained in the figure given below:

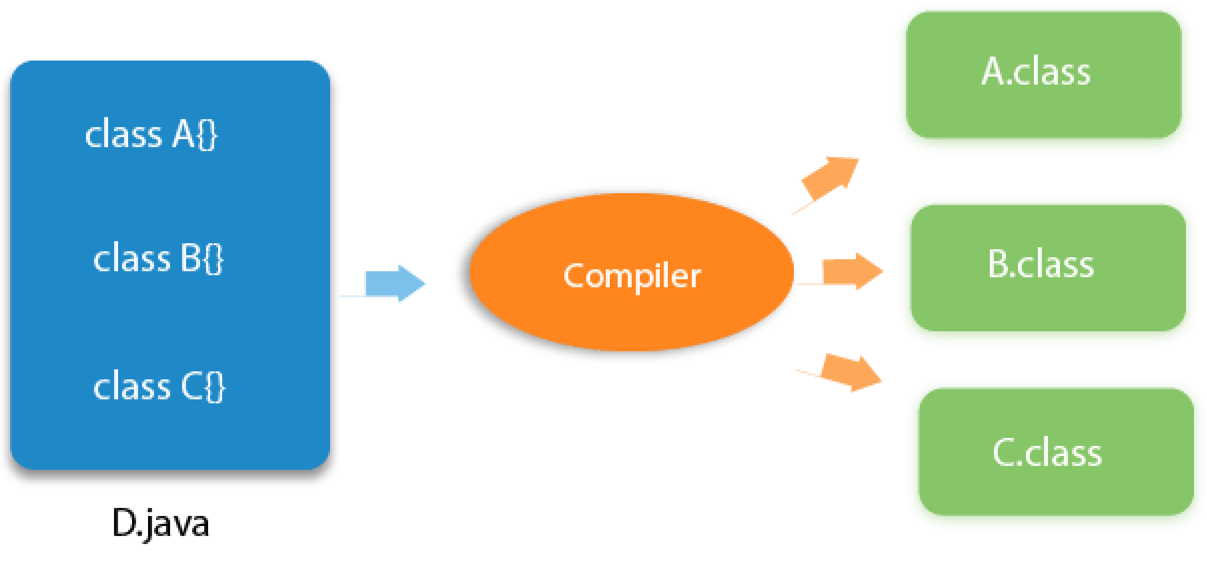

class Simple{
public static void main(String args[]){
System.out.println("Hello");
}}
Hard.java
compiler
Bytecode
Simple.class

To compile:	javac Hard.java
To execute:	java Simple

Observe that, we have compiled the code with file name but running the program with class name. Therefore, we can save a Java program other than class name.

Q) Can you have multiple classes in a java source file?

Yes, like the figure given below illustrates:



How to set path in Java

1. How to set the path of JDK in Windows OS
    1. Setting Temporary Path of JDK
    2. Setting Permanent Path of JDK
2. How to set the path of JDK in Linux OS

The path is required to be set for using tools such as javac, java, etc.

If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.

However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are two ways to set the path in Java:

1. Temporary
2. Permanent

1) How to set the Temporary Path of JDK in Windows

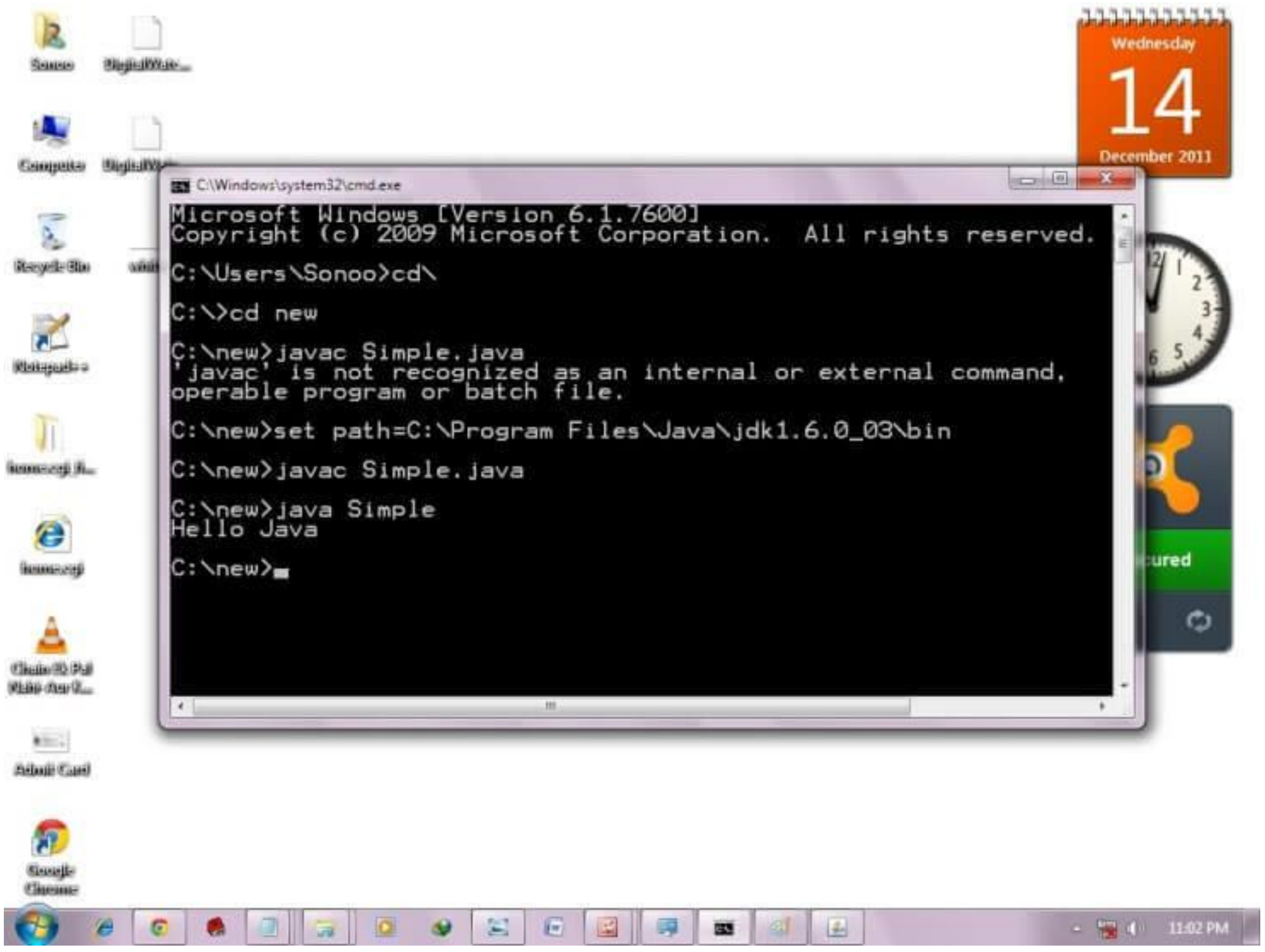
To set the temporary path of JDK, you need to follow the following steps:

- Open the command prompt
- Copy the path of the JDK/bin directory
- Write in command prompt: set path=copied_path

For Example:

```
set path=C:\Program Files\Java\jdk1.6.0_23\bin
```

Let's see it in the figure given below:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Sonoo>cd\

C:\>cd new

C:\new>javac Simple.java
'javac' is not recognized as an internal or external command,
operable program or batch file.

C:\new>set path=C:\Program Files\Java\jdk1.6.0_03\bin

C:\new>javac Simple.java

C:\new>java Simple
Hello Java

C:\new>
```

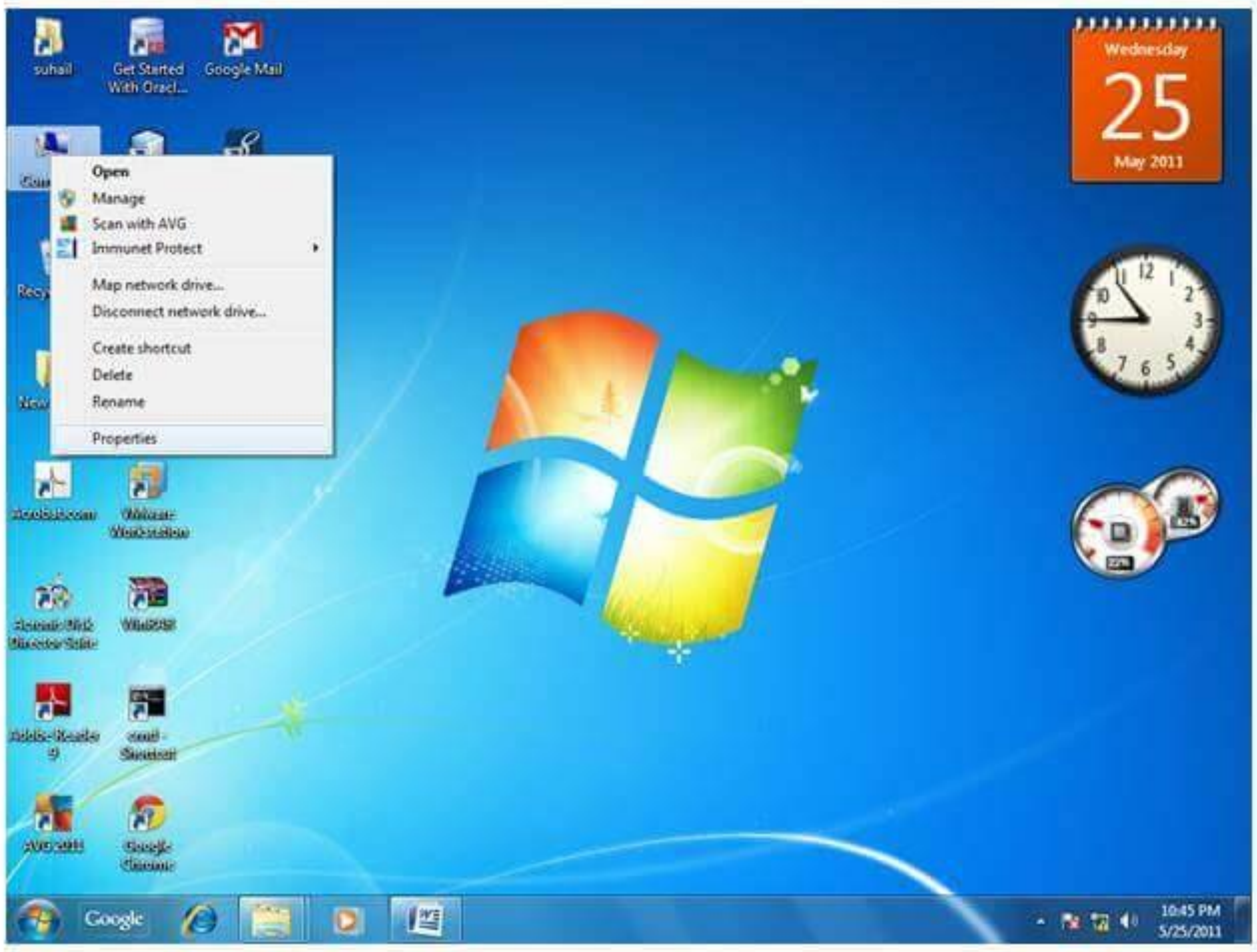
2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

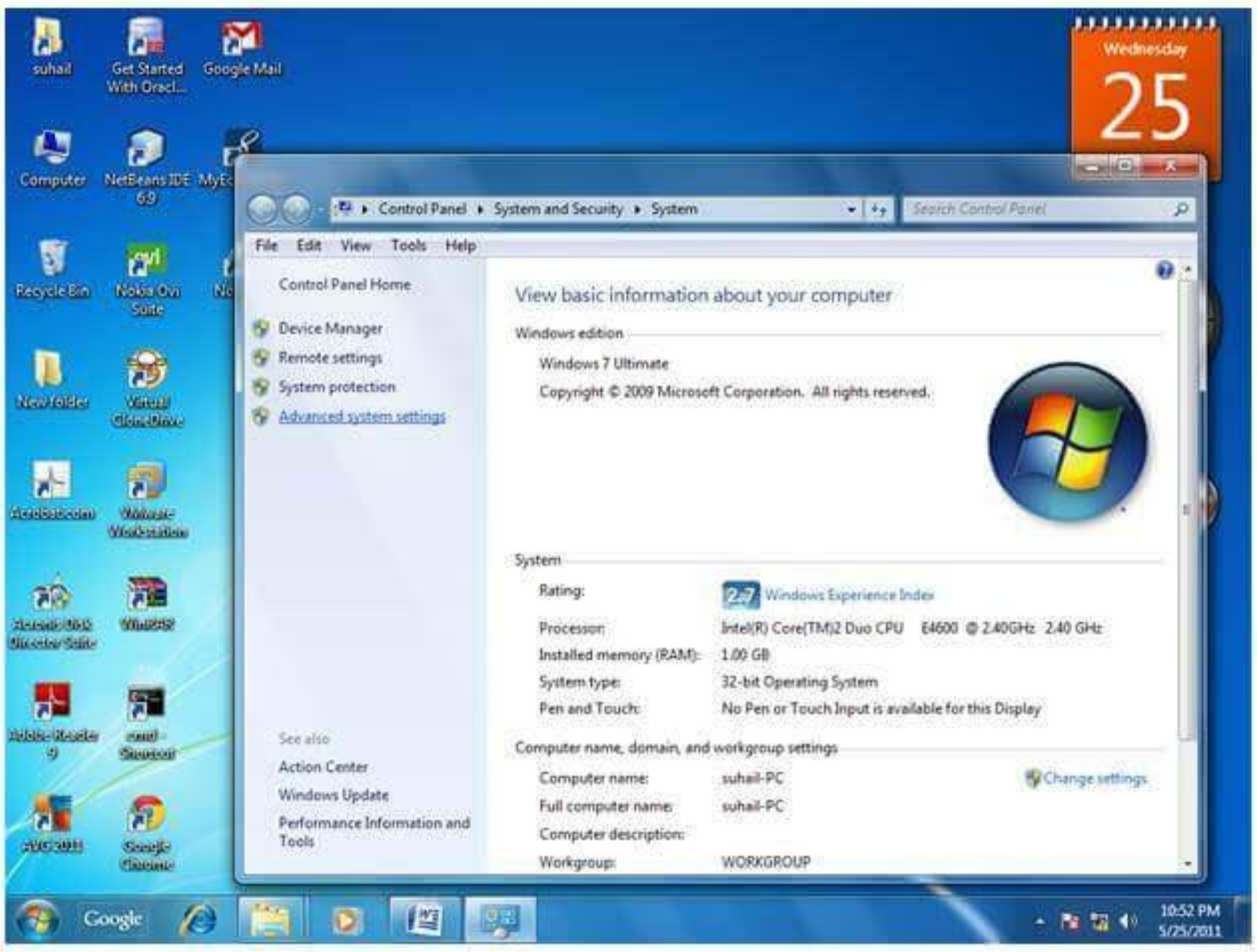
- Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

For Example:

1) Go to MyComputer properties



2) Click on the advanced tab



View basic information about your computer

Windows edition

Windows 7 Ultimate

Copyright © 2009 Microsoft Corporation. All rights reserved.

System

Rating: 2.7 Windows Experience Index

Processor: Intel(R) Core(TM)2 Duo CPU E4600 @ 2.40GHz 2.40 GHz

Installed memory (RAM): 1.00 GB

System type: 32-bit Operating System

Pen and Touch: No Pen or Touch Input is available for this Display

Computer name, domain, and workgroup settings

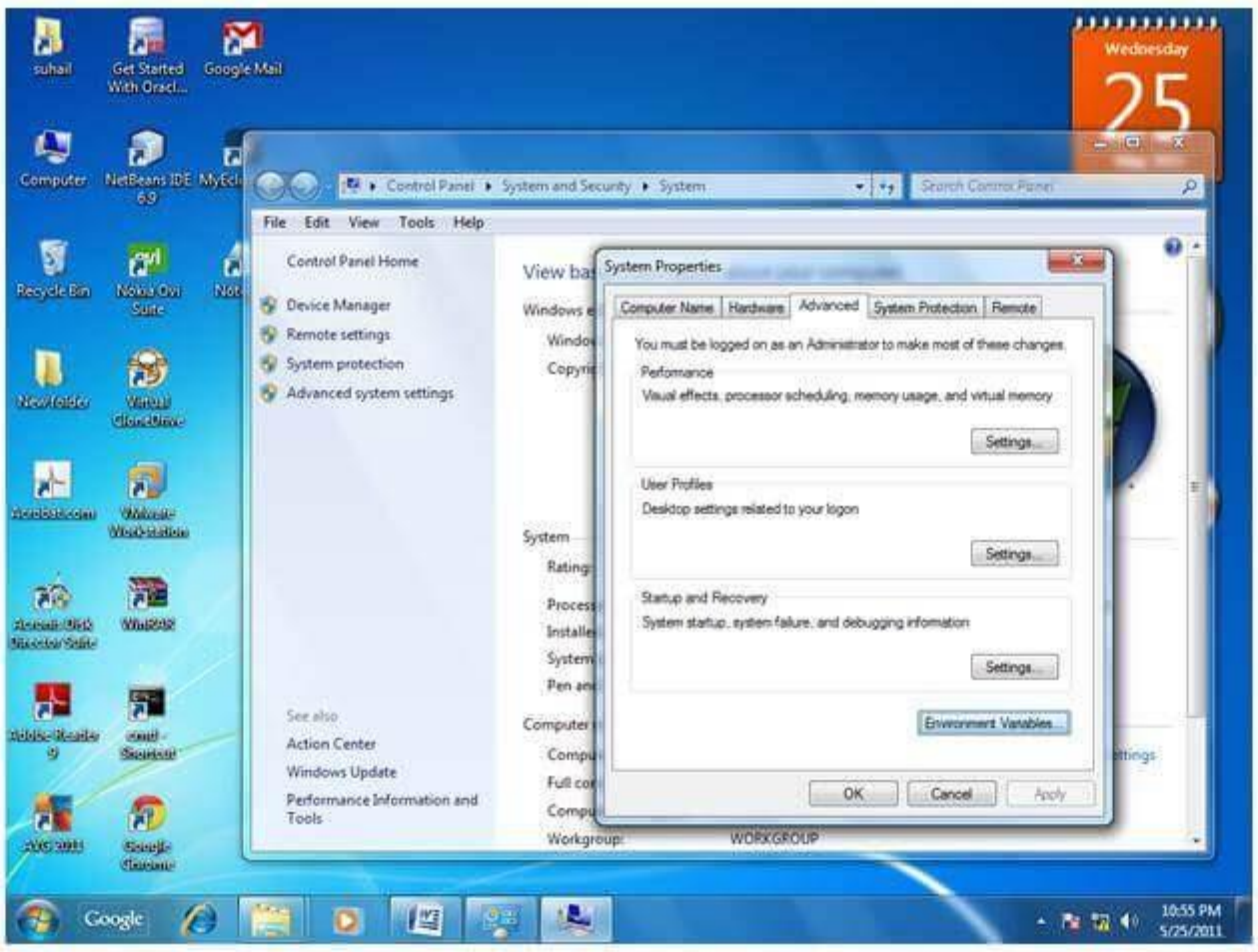
Computer name: suhail-PC

Full computer name: suhail-PC

Computer description:

Workgroup: WORKGROUP

3) Click on environment variables



System Properties

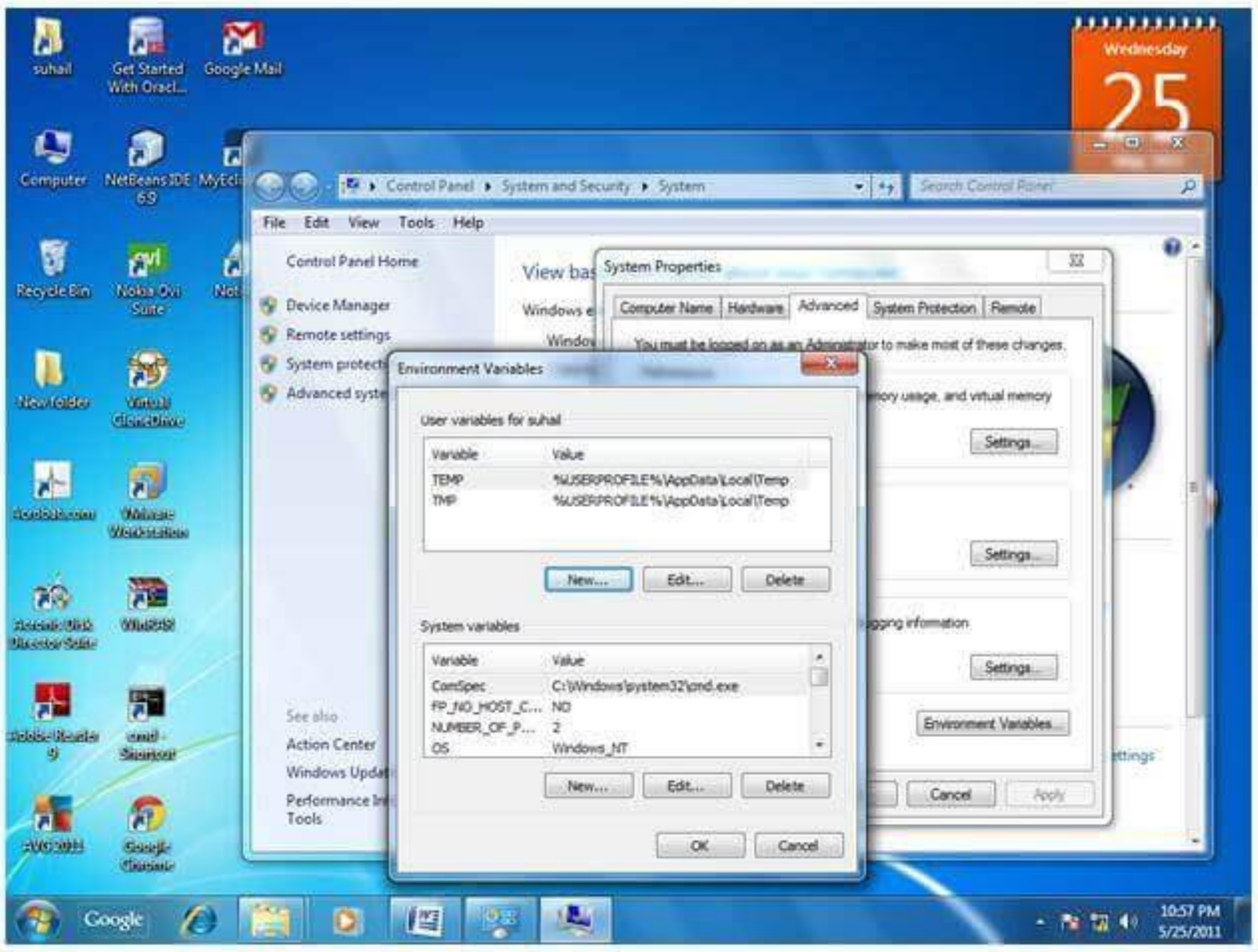
You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your logon

Startup and Recovery
System startup, system failure, and debugging information

4) Click on the new tab of user variables



Environment Variables

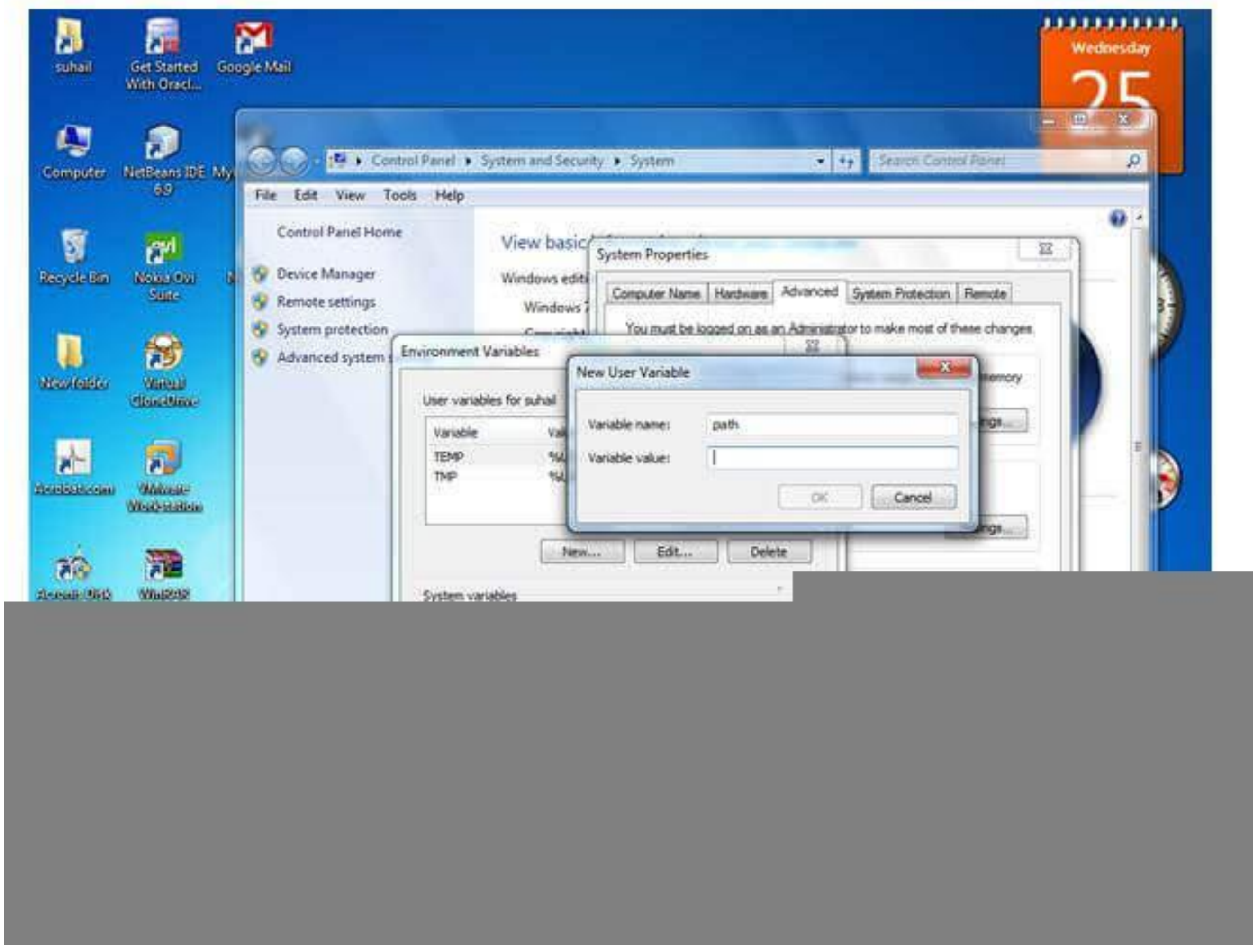
User variables for suhail

Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

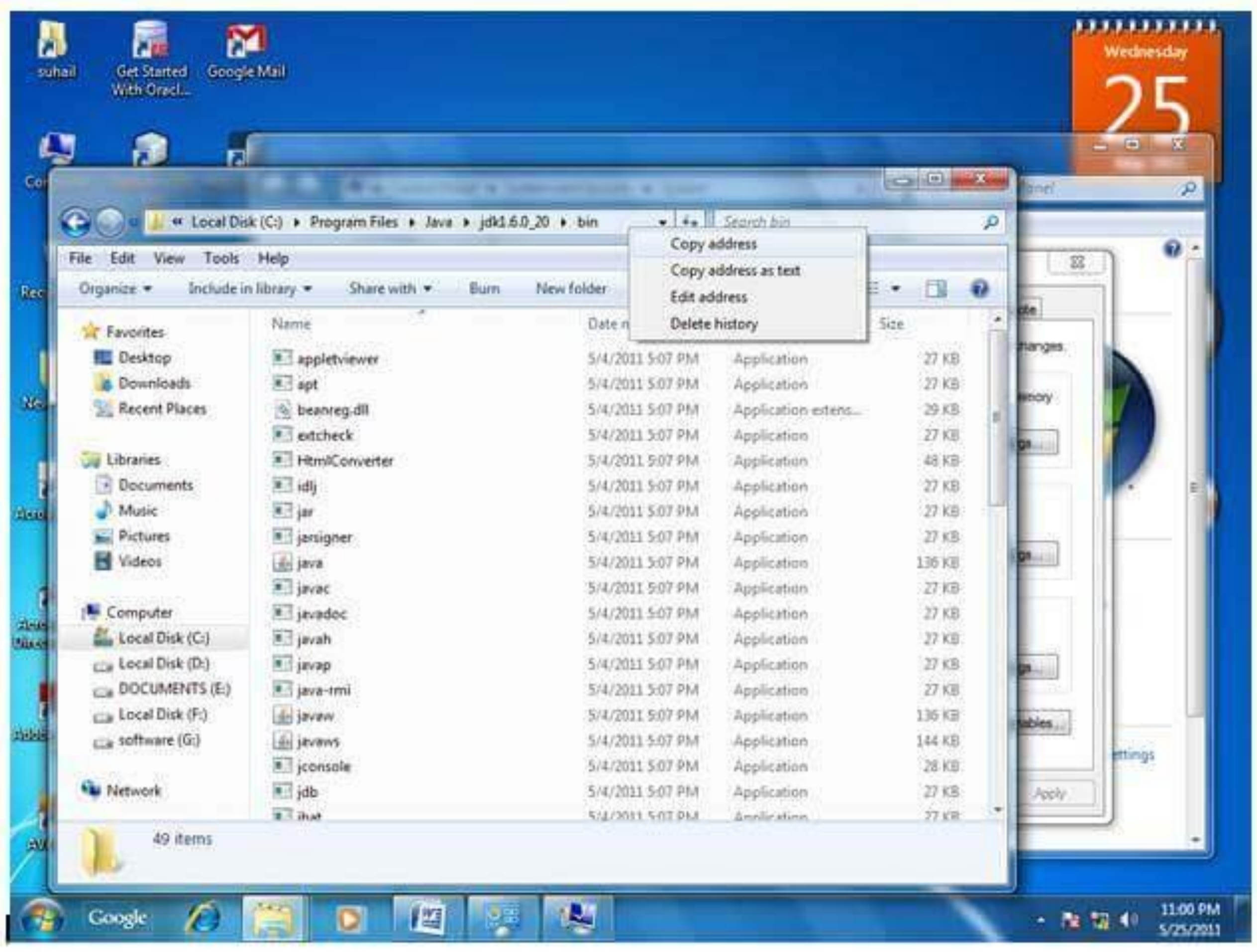
Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	2
OS	Windows_NT

5) Write the path in the variable name



Variable name: path
Variable value:

6) Copy the path of bin folder

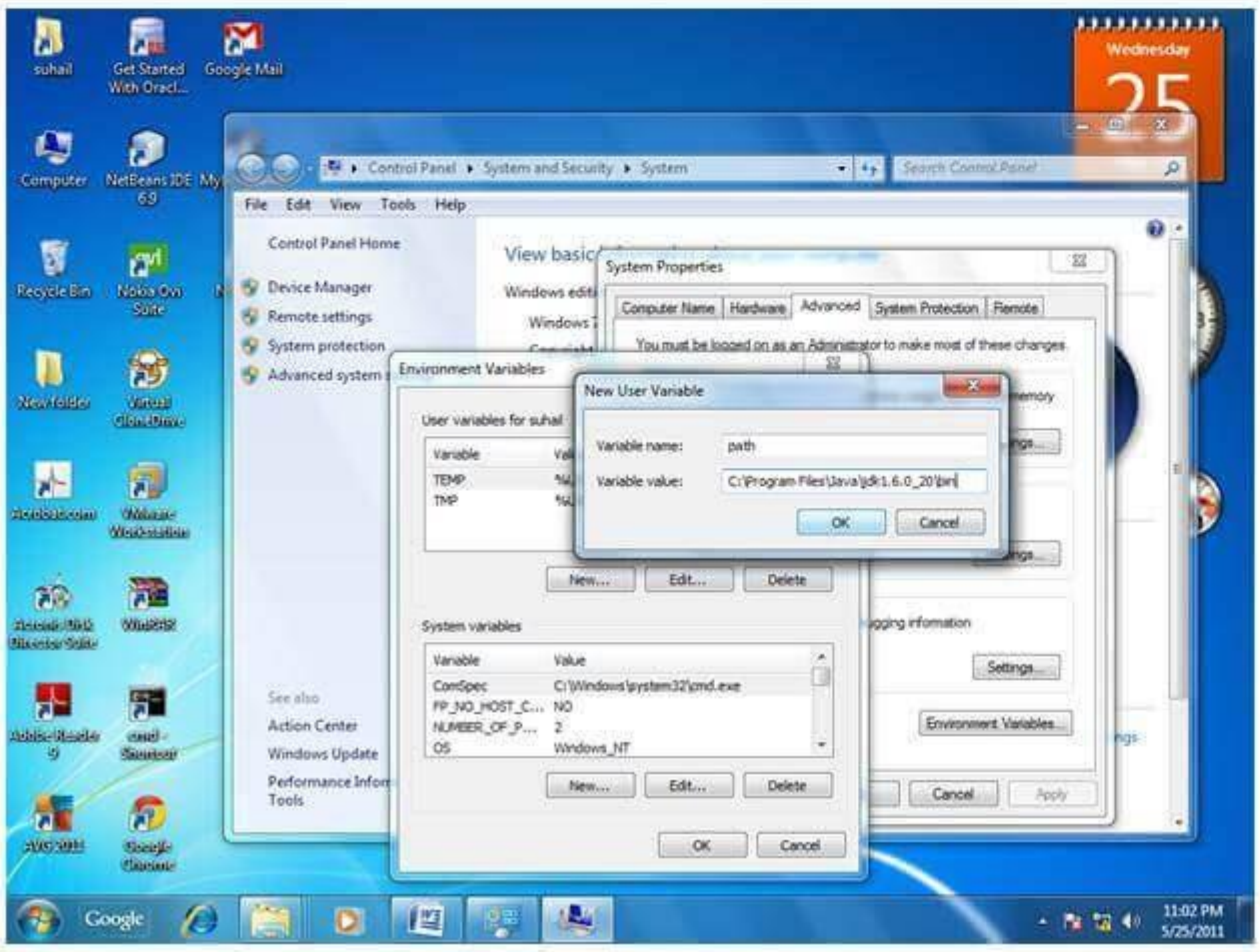


Local Disk (C:) ▸ Program Files ▸ Java ▸ jdk1.6.0_20 ▸ bin

Name	Date modified	Type	Size
appletviewer	5/4/2011 5:07 PM	Application	27 KB
apt	5/4/2011 5:07 PM	Application	27 KB
beanreg.dll	5/4/2011 5:07 PM	Application extens...	29 KB
extcheck	5/4/2011 5:07 PM	Application	27 KB
HtmlConverter	5/4/2011 5:07 PM	Application	48 KB
idlj	5/4/2011 5:07 PM	Application	27 KB
jar	5/4/2011 5:07 PM	Application	27 KB
jarsigner	5/4/2011 5:07 PM	Application	27 KB
java	5/4/2011 5:07 PM	Application	136 KB
javac	5/4/2011 5:07 PM	Application	27 KB
javadoc	5/4/2011 5:07 PM	Application	27 KB
javah	5/4/2011 5:07 PM	Application	27 KB
javap	5/4/2011 5:07 PM	Application	27 KB
java-rmi	5/4/2011 5:07 PM	Application	27 KB
javaw	5/4/2011 5:07 PM	Application	136 KB
javaws	5/4/2011 5:07 PM	Application	144 KB
jconsole	5/4/2011 5:07 PM	Application	28 KB
jdb	5/4/2011 5:07 PM	Application	27 KB

49 items

7) Paste path of bin folder in the variable value



New User Variable

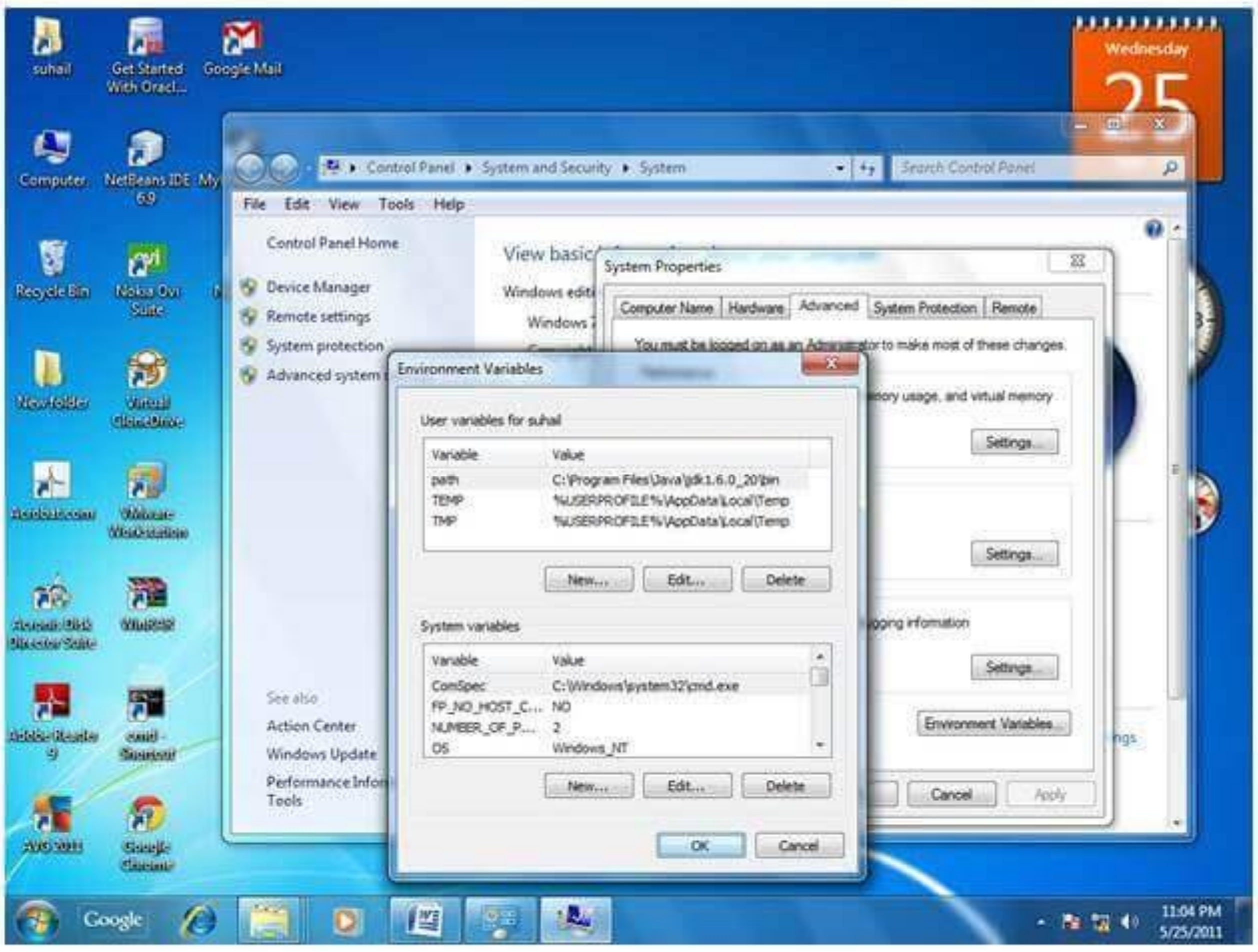
Variable name: path

Variable value: C:\Program Files\Java\jdk1.6.0_20\bin

System variables

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	2
OS	Windows_NT

8) Click on ok button



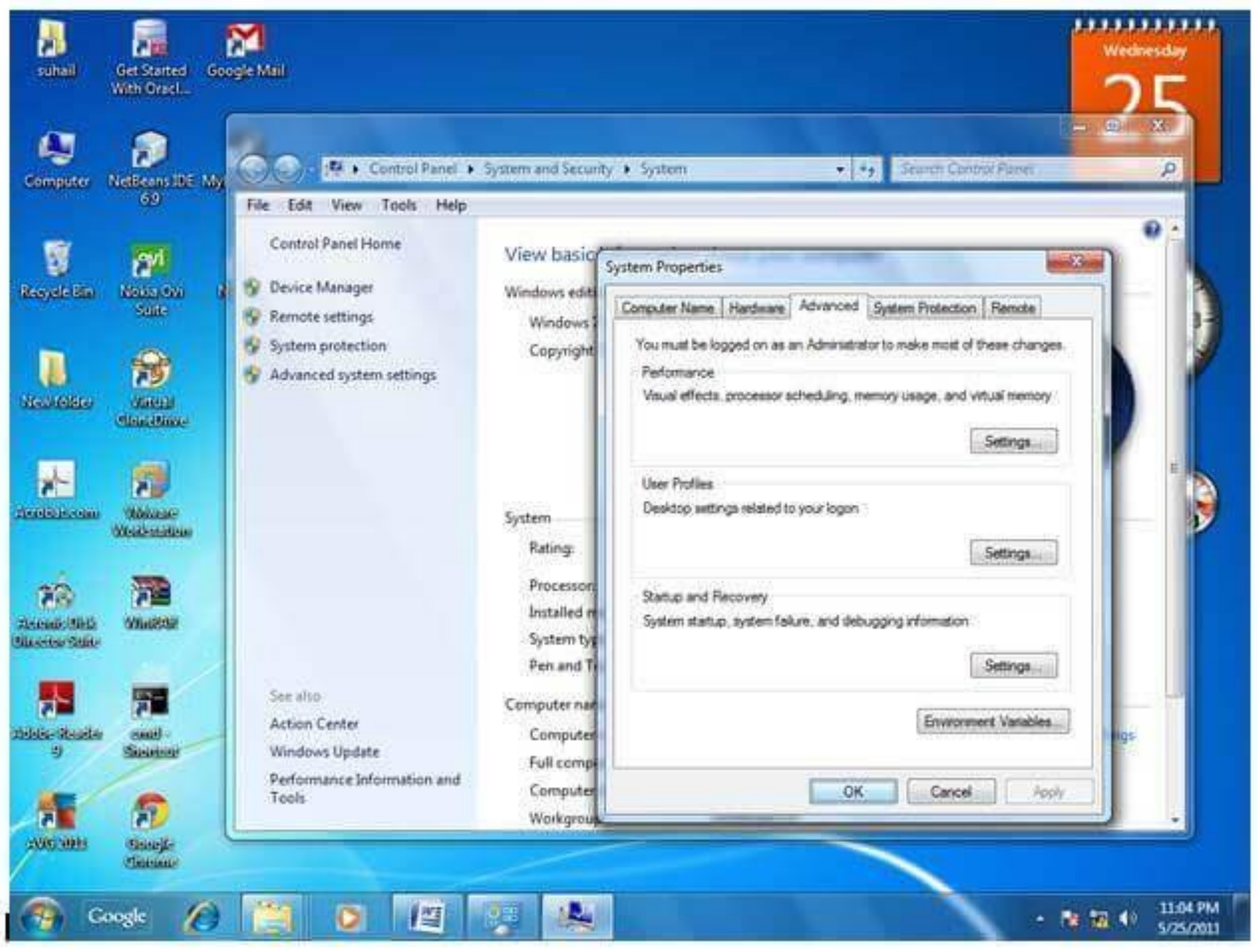
User variables for suhail

Variable	Value
path	C:\Program Files\Java\jdk1.6.0_20\bin
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	2
OS	Windows_NT

9) Click on ok button



You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your logon

Startup and Recovery
System startup, system failure, and debugging information

Now your permanent path is set. You can now execute any program of java from any drive.

Setting Java Path in Linux OS

Setting path in Linux OS is the same as setting the path in the Windows OS. But, here we use the export tool rather than set. Let's see how to set path in Linux OS:

```
export PATH=$PATH:/home/jdk1.6.01/bin/
```

Here, we have installed the JDK in the home directory under Root (/home).

You may also like:

How to set classpath in Java

Difference between JDK, JRE, and JVM

1. A summary of JVM
2. Java Runtime Environment (JRE)
3. Java Development Kit (JDK)

We must understand the differences between JDK, JRE, and JVM before proceeding further to Java. See the brief overview of JVM here.

If you want to get the detailed knowledge of Java Virtual Machine, move to the next page. Firstly, let's see the differences between the JDK, JRE, and JVM.

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:

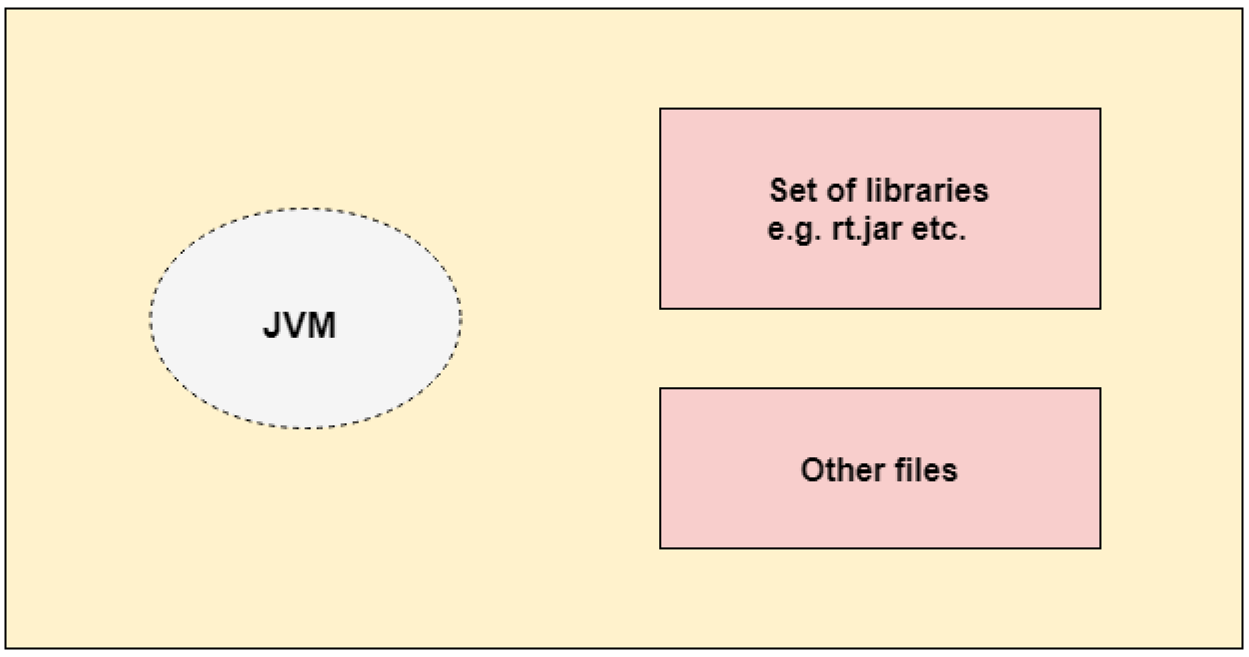
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

More Details.

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 + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



JRE

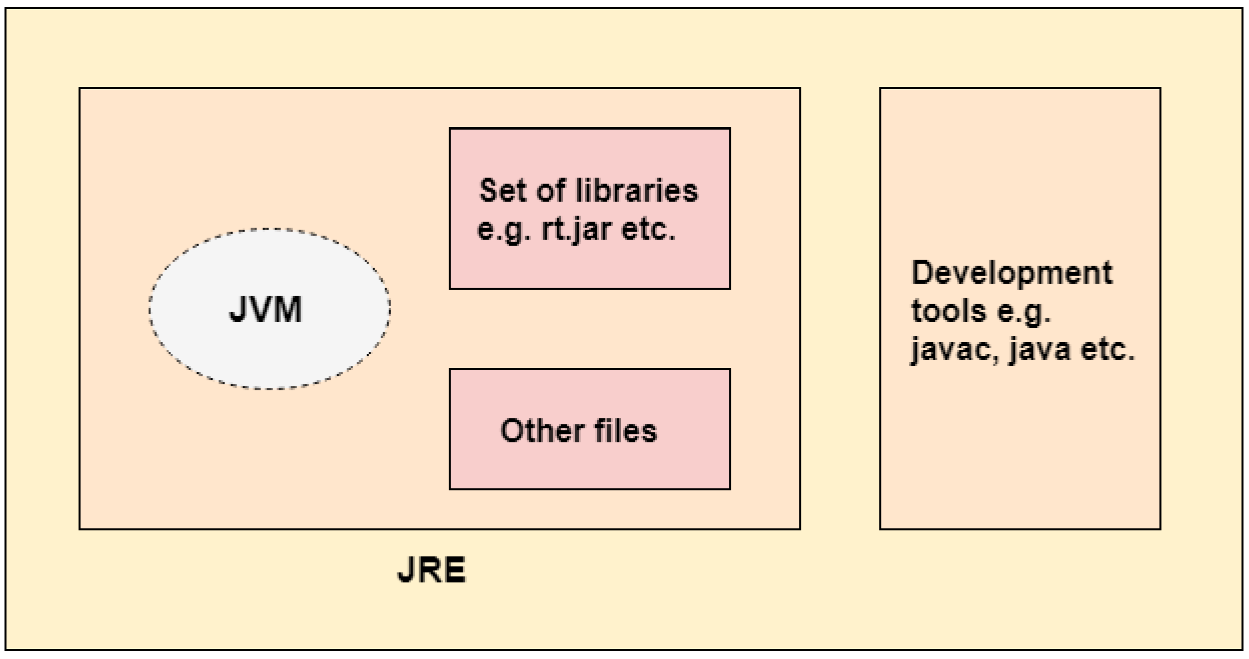
JDK

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.


Set of libraries e.g. rt.jar etc.
JVM
Other files
Development tools e.g. javac, java etc.
JRE

JDK

More Details.

Reference Video

JVM (Java Virtual Machine) Architecture

1. Java Virtual Machine
2. Internal Architecture of JVM

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).

What is JVM

It 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.

What it does

The JVM performs following operation:

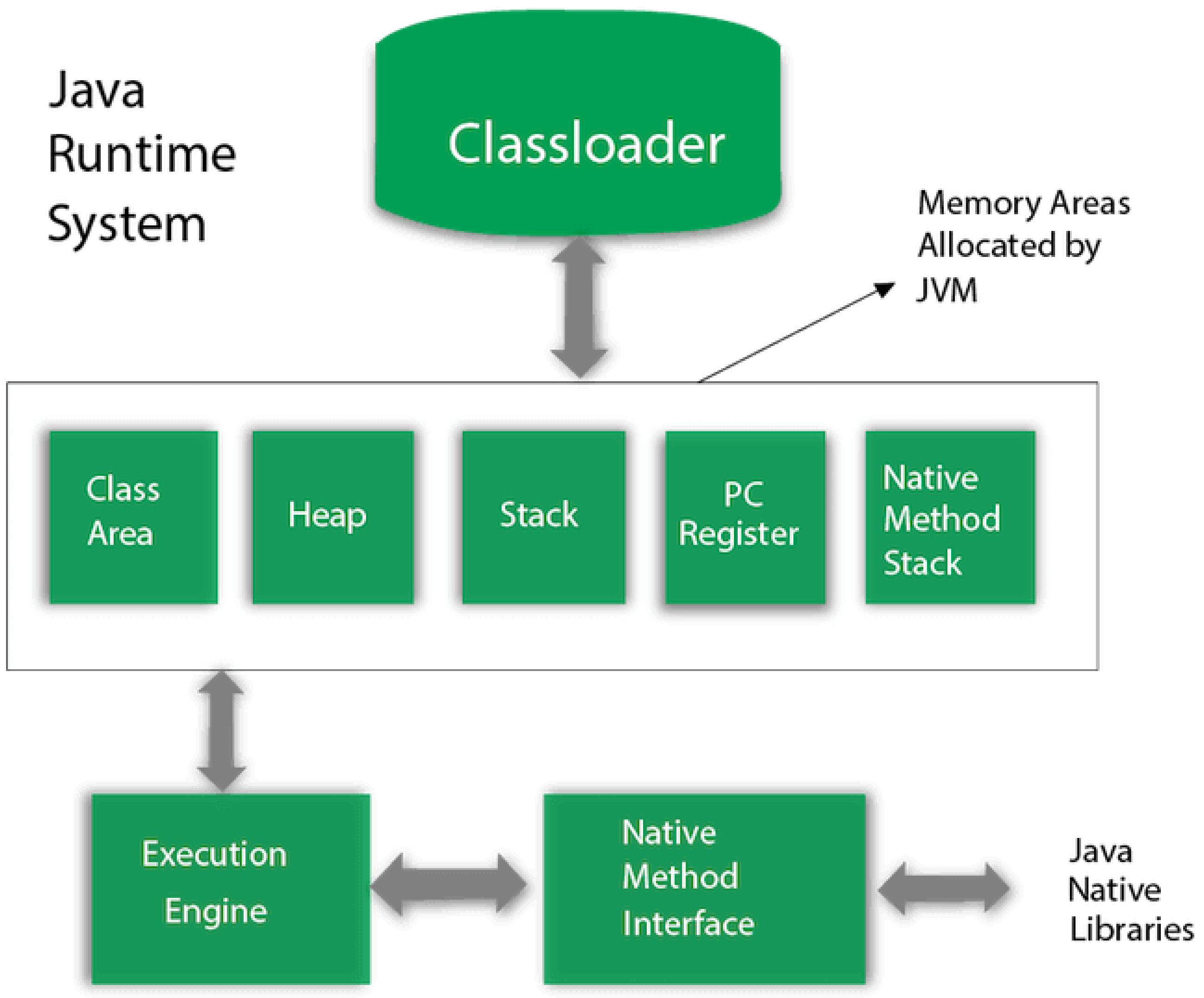
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

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 classloader, 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.

```
//Let's see an example to print the classloader name
public class ClassLoaderExample
{
  public static void main(String[] args)
  {
    // Let's print the classloader name of current class.
    //Application/System classloader will load this class
    Class c=ClassLoaderExample.class;
    System.out.println(c.getClassLoader());
    //If we print the classloader name of String, it will print null because it is an
    //in-built class which is found in rt.jar, so it is loaded by Bootstrap classloader
    System.out.println(String.class.getClassLoader());
  }
}
```

Test it Now

Output:

```
sun.misc.Launcher$AppClassLoader@4e0e2f2a
null
```

These are the internal classloaders provided by Java. If you want to create your own classloader, you need to extend the ClassLoader class.

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

It 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

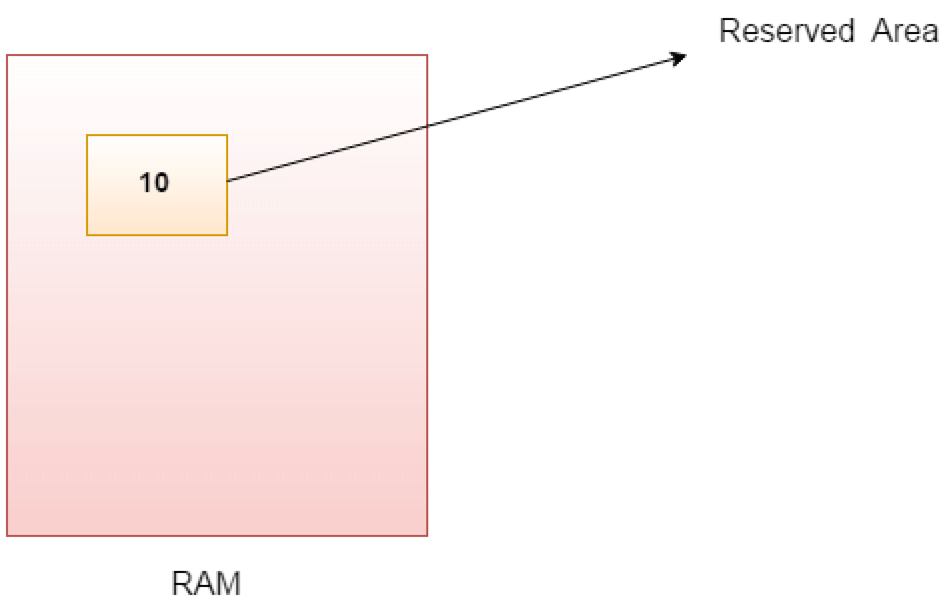
A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in Java: primitive and non-primitive.

Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

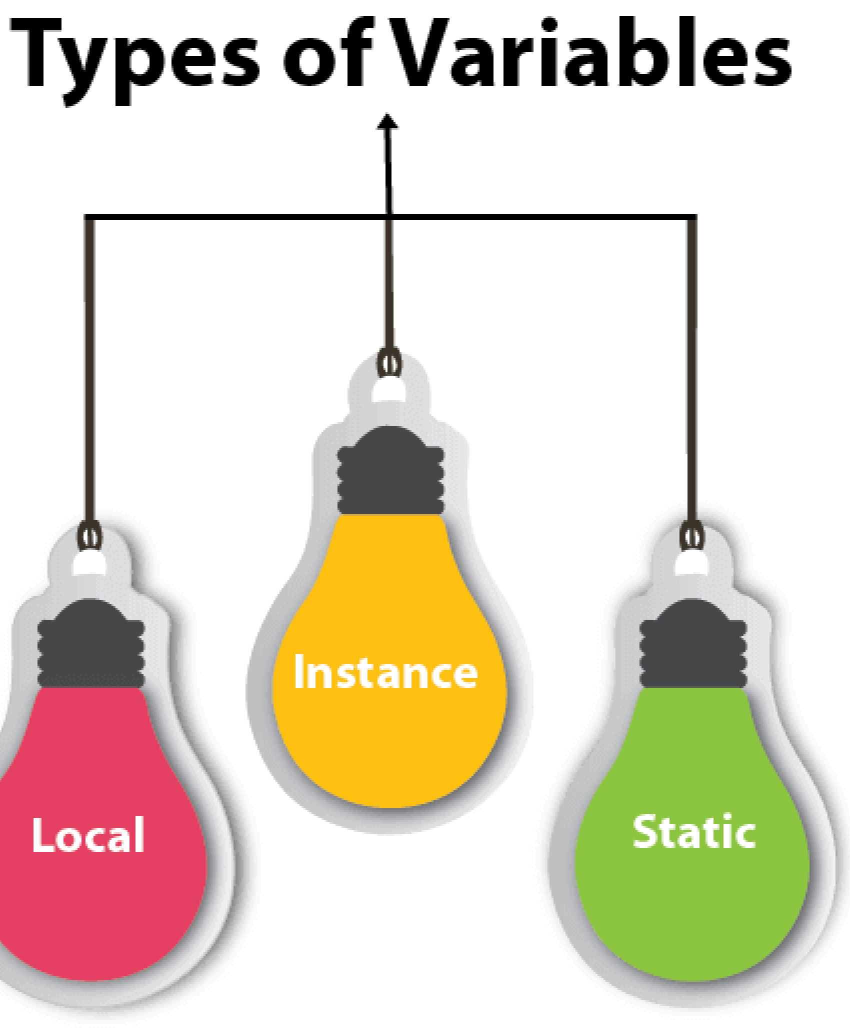


```
1. int data=50;//Here data is variable
```

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. You can use 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. You can create 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

```
public class A
{
  static int m=100;//static variable
  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
int a=(int)f;
System.out.println(f);
System.out.println(a);
}}
```

Output:

```
10.5
10
```

Java Variable Example: Overflow

```
class Simple{
public static void main(String[] args){
//Overflow
int a=130;
byte b=(byte)a;
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
130
-126
```

Java Variable Example: Adding Lower Type

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

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.

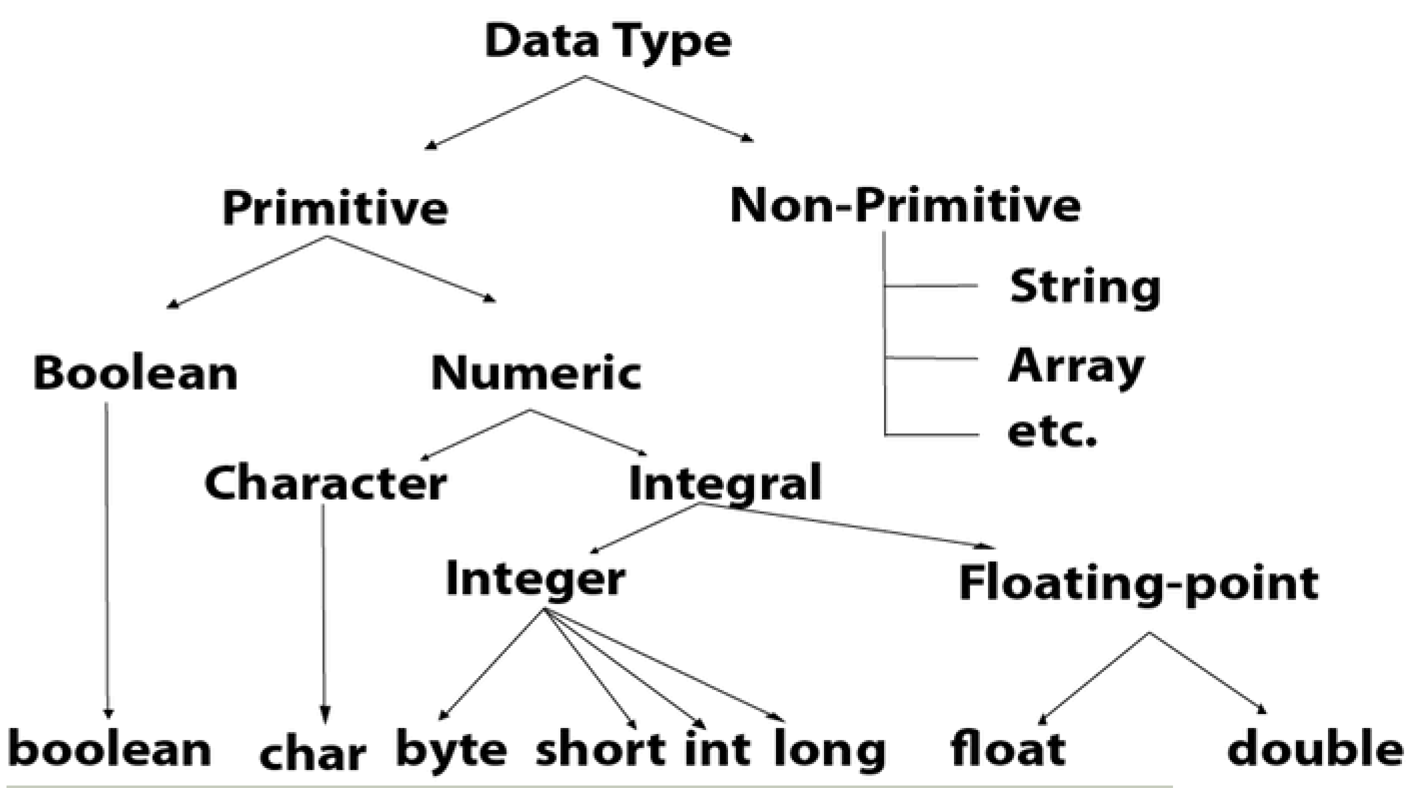
Java Primitive Data Types

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:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



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:

```
1. 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:

```
1. byte a = 10, byte b = -20
```

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). 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:

```
1. short s = 10000, short r = -5000
```

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). 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. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. 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'
```

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

Unicode System

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

Why java uses Unicode System?

Before Unicode, there were many language standards:

- **ASCII** (American Standard Code for Information Interchange) for the United States.
- **ISO 8859-1** for Western European Language.
- **KOI-8** for Russian.
- **GB18030 and BIG-5** for chinese, and so on.

Problem

This caused two problems:

1. A particular code value corresponds to different letters in the various language standards.
2. The encodings for languages with large character sets have variable length.Some common characters are encoded as single bytes, other require two or more byte.

Solution

To solve these problems, a new language standard was developed i.e. Unicode System.

In unicode, character holds 2 byte, so java also uses 2 byte for characters.

lowest value:\u0000

highest value:\uFFFF

Operators in Java

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

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 --

```
public class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21

}}
```

Output:

```
22
21
```

Java Unary Operator Example: ~ and !

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

Output:

```
-11
9
false
true
```

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

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

Output:

```
15
5
50
2
0
```

Java Arithmetic Operator Example: Expression

```java
public class OperatorExample{
public static void main(String args[]){
System.out.println(10*10/5+3-1*4/2);
}}
```

Output:

```
21
```

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

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

Output:

```
40
80
80
240
```

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

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

Output:

```
2
5
2
```

Java Shift Operator Example: >> vs >>>

```
public class OperatorExample{
public static void main(String args[]){
    //For positive number, >> and >>> works same
    System.out.println(20>>2);
    System.out.println(20>>>2);
    //For negative number, >>> changes parity bit (MSB) to 0
    System.out.println(-20>>2);
    System.out.println(-20>>>2);
}}
```

Output:

```
5
5
-5
1073741819
```

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.

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

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);//false && true = false
System.out.println(a);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
false
10
false
11
```

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.

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

Output:

```
true
true
true
10
true
11
```

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

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

Output:

```
2
```

Another Example:

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

Output:

```
5
```

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

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
14
16
```

Java Assignment Operator Example

```
public class OperatorExample{
public static void main(String[] args){
int a=10;
a+=3;//10+3
System.out.println(a);
a-=4;//13-4
System.out.println(a);
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}
```

Output:

```
13
9
18
9
```

Java Assignment Operator Example: Adding short

```
public class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
//a+=b;//a=a+b internally so fine
a=a+b;//Compile time error because 10+10=20 now int
System.out.println(a);
}}
```

Output:

```
Compile time error
```

After type cast:

```
public class OperatorExample{
```

```
public static void main(String args[]){
short a=10;
short b=10;
a=(short)(a+b);//20 which is int now converted to short
System.out.println(a);
}}
```

Output:

```
20
```

You may also like

Operator Shifting in Java

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

A list of Java keywords or reserved words are given below:

1. **abstract:** Java abstract keyword is used to declare an abstract class. An abstract class can provide the implementation of the interface. It can have abstract and non-abstract methods.
2. **boolean:** Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. **break:** Java break keyword is used to break the loop or switch statement. It breaks the current flow of the program at specified conditions.
4. **byte:** Java byte keyword is used to declare a variable that can hold 8-bit data values.
5. **case:** Java case keyword is used with the switch statements to mark blocks of text.
6. **catch:** Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. **char:** Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. **class:** Java class keyword is used to declare a class.
9. **continue:** Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.
10. **default:** Java default keyword is used to specify the default block of code in a switch statement.
11. **do:** Java do keyword is used in the control statement to declare a loop. It can iterate a part of the program several times.
12. **double:** Java double keyword is used to declare a variable that can hold 64-bit floating-point number.
13. **else:** Java else keyword is used to indicate the alternative branches in an if statement.
14. **enum:** Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. **extends:** Java extends keyword is used to indicate that a class is derived from another class or interface.
16. **final:** Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.
17. **finally:** Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an exception is handled or not.
18. **float:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.
19. **for:** Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some condition becomes true. If the number of iteration is fixed, it is recommended to use for loop.
20. **if:** Java if keyword tests the condition. It executes the if block if the condition is true.
21. **implements:** Java implements keyword is used to implement an interface.
22. **import:** Java import keyword makes classes and interfaces available and accessible to the current source code.

23. **instanceof:** Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. **int:** Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. **interface:** Java interface keyword is used to declare an interface. It can have only abstract methods.
26. **long:** Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. **native:** Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. **new:** Java new keyword is used to create new objects.
29. **null:** Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. **package:** Java package keyword is used to declare a Java package that includes the classes.
31. **private:** Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
32. **protected:** Java protected keyword is an access modifier. It can be accessible within the package and outside the package but through inheritance only. It can't be applied with the class.
33. **public:** Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. **return:** Java return keyword is used to return from a method when its execution is complete.
35. **short:** Java short keyword is used to declare a variable that can hold a 16-bit integer.
36. **static:** Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.
37. **strictfp:** Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. **super:** Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.
39. **switch:** The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. **synchronized:** Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. **this:** Java this keyword can be used to refer the current object in a method or constructor.
42. **throw:** The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exceptions. It is followed by an instance.
43. **throws:** The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.
44. **transient:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. **try:** Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. **void:** Java void keyword is used to specify that a method does not have a return value.
47. **volatile:** Java volatile keyword is used to indicate that a variable may change asynchronously.
48. **while:** Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use the while loop.

Control Statements

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. It is one of the fundamental features of Java, which provides a smooth flow of program.

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.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```java
if(condition) {
statement 1; //executes when condition is true
}
```

Consider the following example in which we have used the **if** statement in the java code.

Student.java

Student.java

```java
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
```

```
System.out.println("city is agra");
}else {
System.out.println(city);
}
}
}
```

Output:

```
Delhi
```

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";

if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}
}
}
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.

- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```java
switch (expression){
  case value1:
   statement1;
   break;
  .
  .
  .
  case valueN:
   statementN;
   break;
  default:
   default statement;
}
```

Consider the following example to understand the flow of the switch statement.

Student.java

```java
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output:

```
2
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

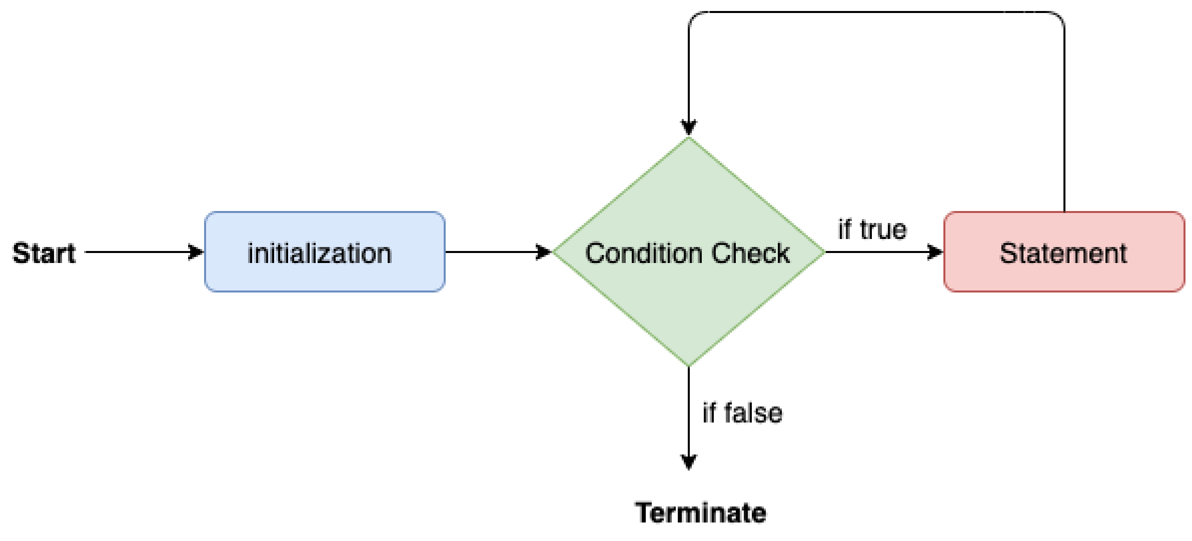
Let's understand the loop statements one by one.

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

```
public class Calculattion {
public static void main(String[] args) {
// TODO Auto-generated method stub
int sum = 0;
for(int j = 1; j<=10; j++) {
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " + sum);
}
}
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name){
//statements
}
```

Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
```

```java
}
}
```

Output:

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

Java while loop

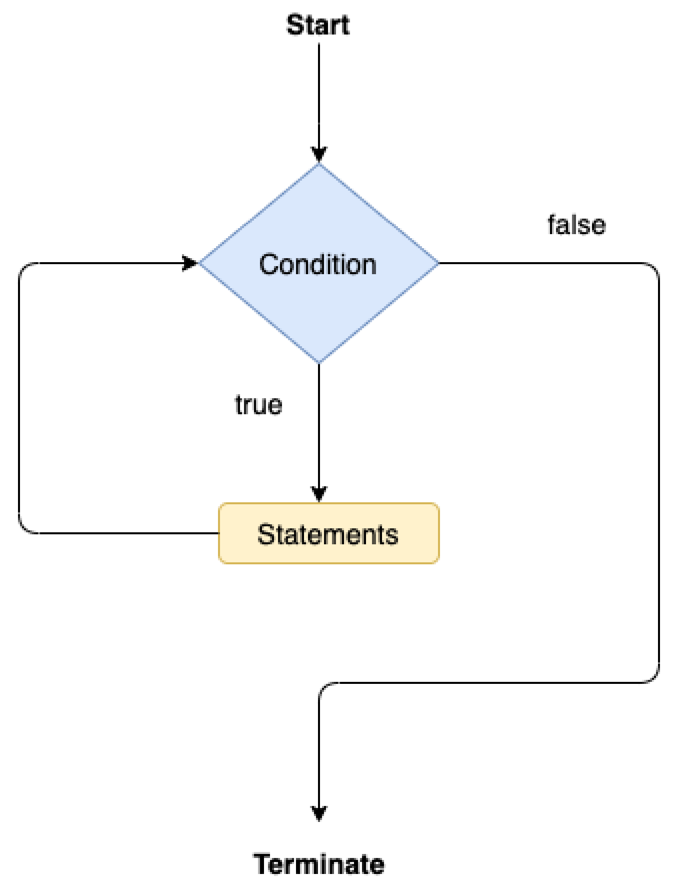
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

```java
while(condition){
//looping statements
}
```

The flow chart for the while loop is given in the following image.



Consider the following example.

Calculation .java

```java
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
}
}
```

Output:

```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

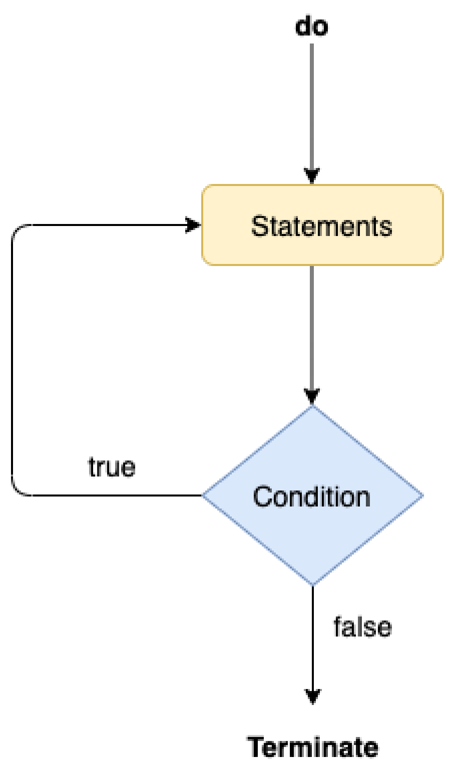
Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

```
do
{
//statements
} while (condition);
```

The flow chart of the do-while loop is given in the following image.



Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);
}
}
```

Output:

```
Printing the list of first 10 even numbers
0
2
4
6
8
10
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

BreakExample.java

```java
public class BreakExample {

public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
}
}
}
}
```

Output:

```
0
1
2
3
4
5
6
```

break statement example with labeled for loop

Calculation.java

```java
public class Calculation {

public static void main(String[] args) {
// TODO Auto-generated method stub
a:
for(int i = 0; i<= 10; i++) {
b:
for(int j = 0; j<=15;j++) {
c:
for (int k = 0; k<=20; k++) {
System.out.println(k);
if(k==5) {
break a;
}
}
}

}
}


}
```

Output:

```
0
1
2
3
4
5
```

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```
public class ContinueExample {

public static void main(String[] args) {
// TODO Auto-generated method stub

for(int i = 0; i<= 2; i++) {

for (int j = i; j<=5; j++) {

if(j == 4) {
continue;
}
System.out.println(j);
}
}
}

}
```

Output:

```
0
1
2
3
5
1
2
3
5
2
3
5
```

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.

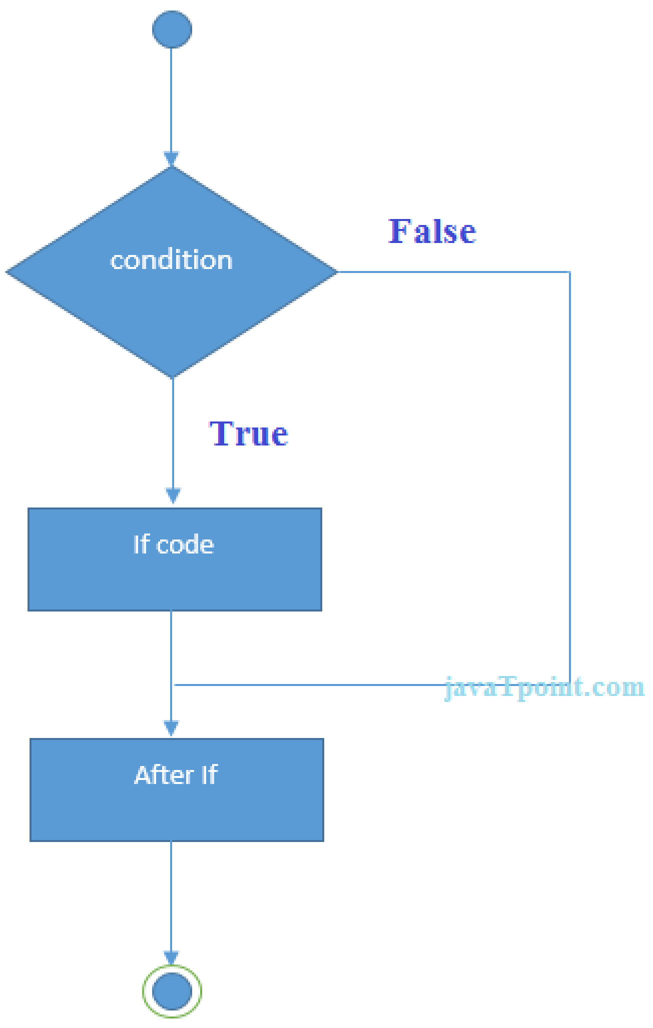
- if statement
- if-else statement
- if-else-if ladder
- nested if statement

Java if Statement

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

Syntax:

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



Example:

```java
//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");
    }
}
}
```

Test it Now

Output:

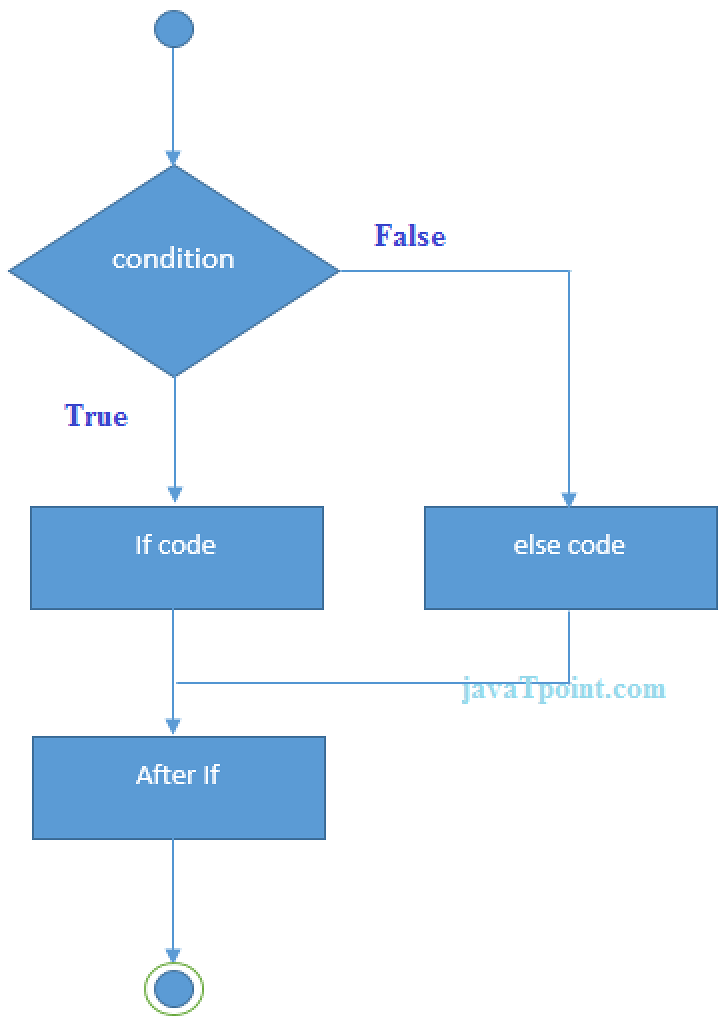
```
Age is greater than 18
```

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:

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



Example:

```
//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");
    }
}
}
```

Test it Now

Output:

```
odd number
```

Leap Year Example:

A year is leap, if it is divisible by 4 and 400. But, not by 100.

```
public class LeapYearExample {
public static void main(String[] args) {
    int year=2020;
    if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
        System.out.println("LEAP YEAR");
    }
    else{
        System.out.println("COMMON YEAR");
    }
}
}
```

Output:

```
LEAP YEAR
```

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:

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

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
}
```

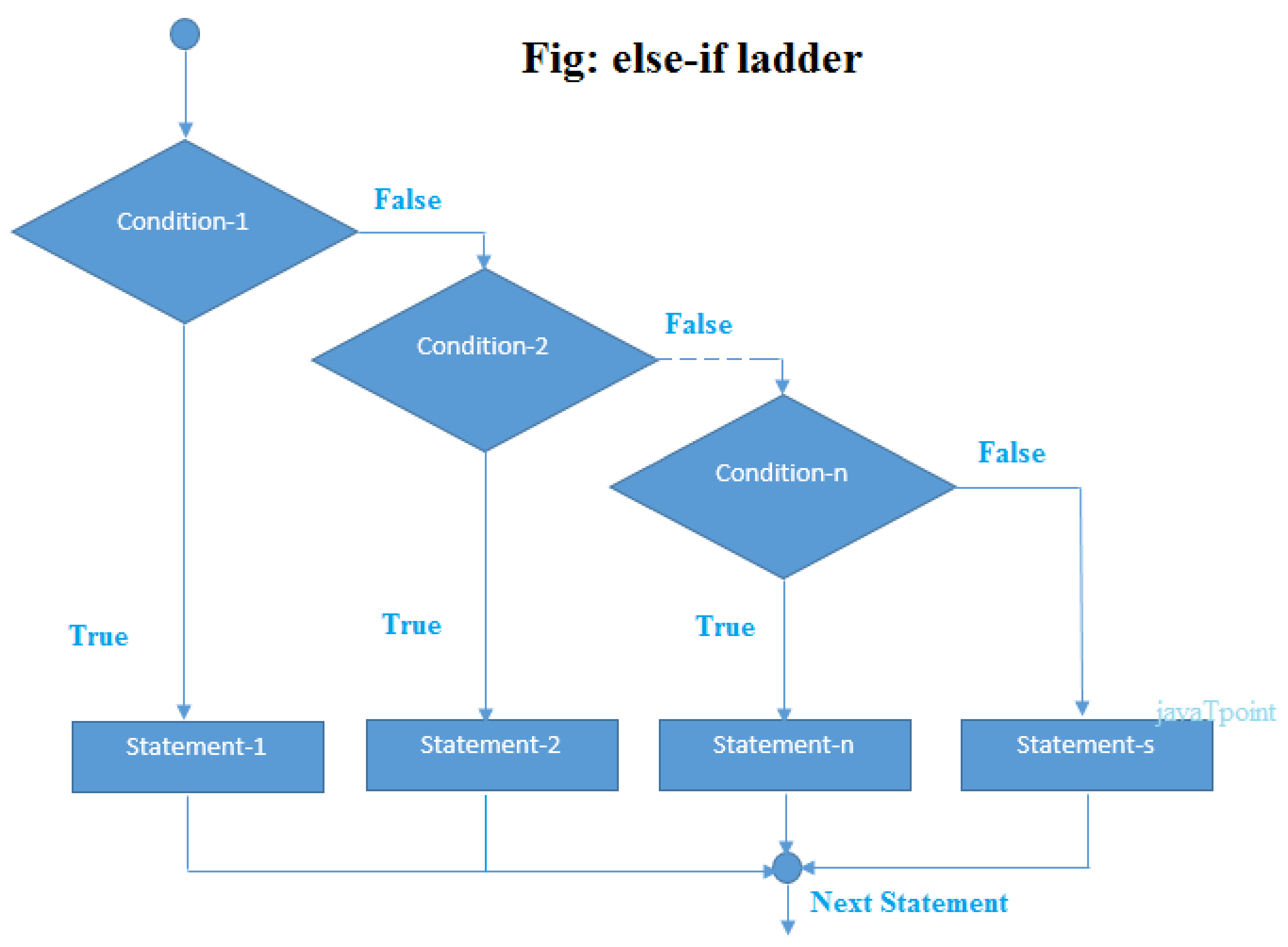


Fig: else-if ladder

Example:

```java
//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
```

Program to check POSITIVE, NEGATIVE or ZERO:

```java
public class PositiveNegativeExample {
public static void main(String[] args) {
    int number=-13;
    if(number>0){
    System.out.println("POSITIVE");
    }else if(number<0){
    System.out.println("NEGATIVE");
    }else{
    System.out.println("ZERO");
  }
}
}
```

Output:

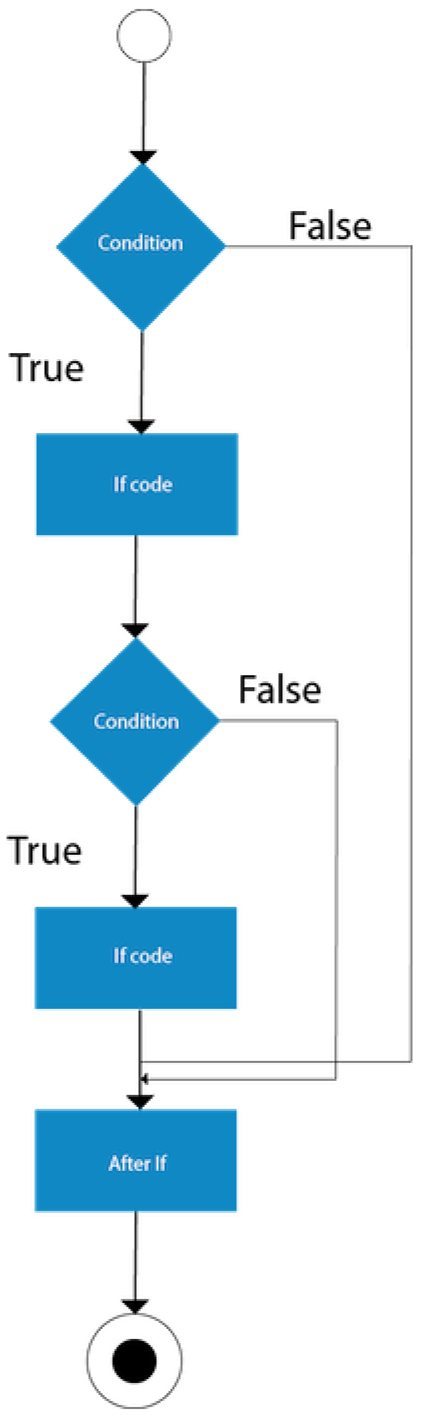
```
NEGATIVE
```

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:

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



Example:

```
//Java Program to demonstrate the use of Nested If Statement.
```

```java
public class JavaNestedIfExample {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=20;
    int weight=80;
    //applying condition on age and weight
    if(age>=18){
        if(weight>50){
            System.out.println("You are eligible to donate blood");
        }
    }
}}
```

Test it Now

Output:

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

Example 2:

```java
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample2 {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=25;
    int weight=48;
    //applying condition on age and weight
    if(age>=18){
        if(weight>50){
            System.out.println("You are eligible to donate blood");
        } else{
            System.out.println("You are not eligible to donate blood");
        }
    } else{
      System.out.println("Age must be greater than 18");
    }
} }
```

Test it Now

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

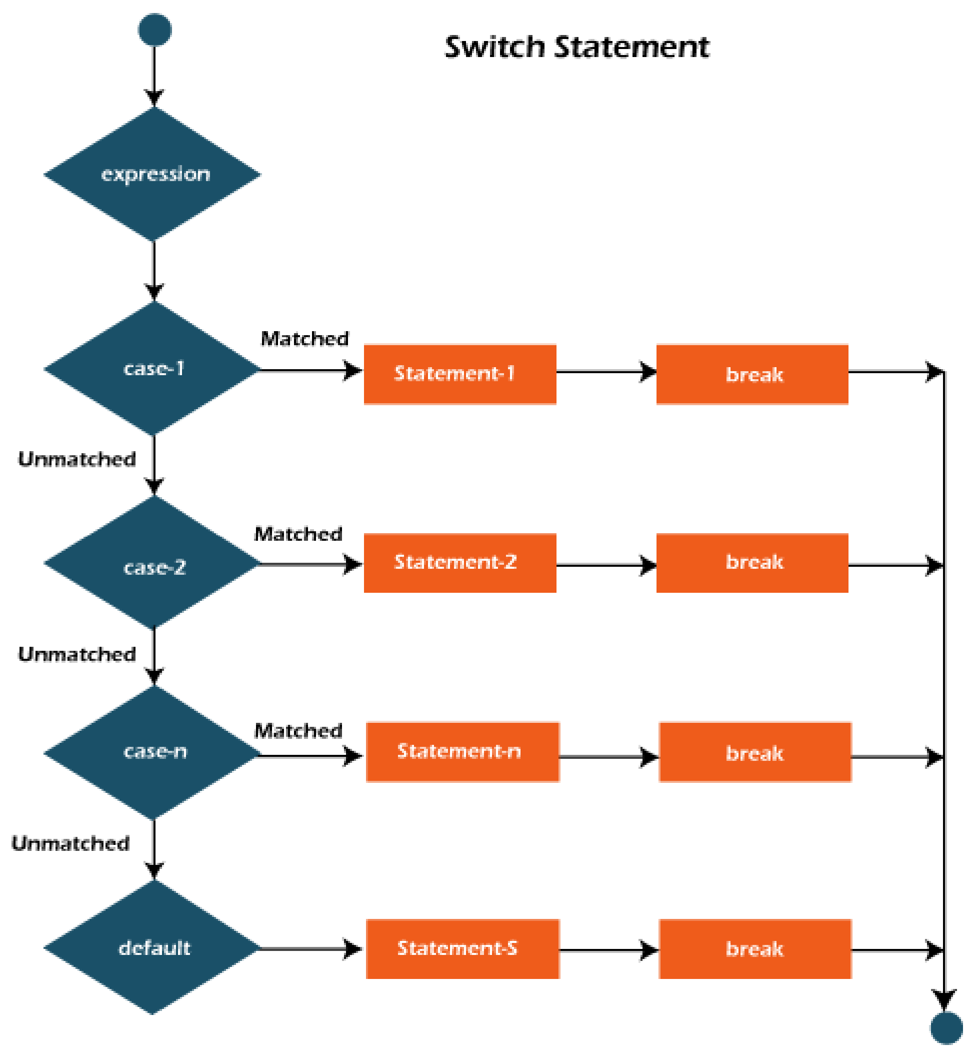
  , it jumps the control after the switch expression. If a break statement is not found, it executes the next case.

- 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

```java
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");
    }
}
}
```
Test it Now

Output:

```
20
```

Finding Month Example:

SwitchMonthExample.javaHTML

```java
//Java Program to demonstrate the example of Switch statement
//where we are printing month name for the given number
public class SwitchMonthExample {
public static void main(String[] args) {
    //Specifying month number
    int month=7;
    String monthString="";
    //Switch statement
    switch(month){
    //case statements within the switch block
    case 1: monthString="1 - January";
    break;
    case 2: monthString="2 - February";
    break;
    case 3: monthString="3 - March";
    break;
    case 4: monthString="4 - April";
    break;
    case 5: monthString="5 - May";
    break;
    case 6: monthString="6 - June";
    break;
    case 7: monthString="7 - July";
    break;
    case 8: monthString="8 - August";
    break;
    case 9: monthString="9 - September";
    break;
    case 10: monthString="10 - October";
    break;
    case 11: monthString="11 - November";
    break;
    case 12: monthString="12 - December";
    break;
    default:System.out.println("Invalid Month!");
    }
    //Printing month of the given number
    System.out.println(monthString);
}
}
```

Test it Now

Output:

```
7 - July
```

Program to check Vowel or Consonant:

If the character is A, E, I, O, or U, it is vowel otherwise consonant. It is not case-sensitive.

SwitchVowelExample.java

```
public class SwitchVowelExample {
public static void main(String[] args) {
    char ch='O';
    switch(ch)
    {
        case 'a':
            System.out.println("Vowel");
            break;
        case 'e':
            System.out.println("Vowel");
            break;
        case 'i':
            System.out.println("Vowel");
            break;
        case 'o':
            System.out.println("Vowel");
            break;
        case 'u':
            System.out.println("Vowel");
            break;
        case 'A':
            System.out.println("Vowel");
            break;
        case 'E':
            System.out.println("Vowel");
            break;
        case 'I':
          System.out.println("Vowel");
           break;
       case 'O':
           System.out.println("Vowel");
          break;
        case 'U':
          System.out.println("Vowel");
           break;
       default:
           System.out.println("Consonant");
    }
}
}
```

Output:

```
Vowel
```

Java Switch Statement is fall-through

The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

Example:

SwitchExample2.java

```
//Java Switch Example where we are omitting the
//break statement
public class SwitchExample2 {
public static void main(String[] args) {
    int number=20;
    //switch expression with int value
    switch(number){
    //switch cases without break statements
    case 10: System.out.println("10");
    case 20: System.out.println("20");
    case 30: System.out.println("30");
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Test it Now

Output:

```
20
30
Not in 10, 20 or 30
```

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:

SwitchStringExample.java

```
//Java Program to demonstrate the use of Java Switch
//statement with String
public class SwitchStringExample {
public static void main(String[] args) {
    //Declaring String variable
    String levelString="Expert";
    int level=0;
    //Using String in Switch expression
    switch(levelString){
    //Using String Literal in Switch case
    case "Beginner": level=1;
    break;
    case "Intermediate": level=2;
    break;
    case "Expert": level=3;
    break;
    default: level=0;
    break;
    }
    System.out.println("Your Level is: "+level);
}
}
```

Test it Now

Output:

```
Your Level is: 3
```

Java Nested Switch Statement

We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

Example:

NestedSwitchExample.java

```
//Java Program to demonstrate the use of Java Nested Switch
public class NestedSwitchExample {
    public static void main(String args[])
     {
     //C - CSE, E - ECE, M - Mechanical
       char branch = 'C';
        int collegeYear = 4;
       switch( collegeYear )
        {
           case 1:
               System.out.println("English, Maths, Science");
```

```
                break;
            case 2:
                switch( branch )
                 {
                   case 'C':
                        System.out.println("Operating System, Java, Data Structure");
                      break;
                    case 'E':
                      System.out.println("Micro processors, Logic switching theory");
                        break;
                   case 'M':
                        System.out.println("Drawing, Manufacturing Machines");
                      break;
                 }
                break;
            case 3:
                switch( branch )
                 {
                   case 'C':
                        System.out.println("Computer Organization, MultiMedia");
                      break;
                    case 'E':
                      System.out.println("Fundamentals of Logic Design, Microelectronics");
                        break;
                   case 'M':
                        System.out.println("Internal Combustion Engines, Mechanical Vibration");
                      break;
                 }
                break;
            case 4:
                switch( branch )
                 {
                   case 'C':
                        System.out.println("Data Communication and Networks, MultiMedia");
                      break;
                    case 'E':
                      System.out.println("Embedded System, Image Processing");
                        break;
                   case 'M':
                        System.out.println("Production Technology, Thermal Engineering");
                      break;
                 }
                break;
        }
    }
}
```
Test it Now

Output:

```
Data Communication and Networks, MultiMedia
```

Java Enum in Switch Statement

Java allows us to use enum in switch statement. Java enum is a class that represent the group of constants. (immutable such as final variables). We use the keyword enum and put the constants in curly braces separated by comma.

Example:

JavaSwitchEnumExample.java

```java
//Java Program to demonstrate the use of Enum
//in switch statement
public class JavaSwitchEnumExample {
      public enum Day {  Sun, Mon, Tue, Wed, Thu, Fri, Sat  }
       public static void main(String args[])
      {
        Day[] DayNow = Day.values();
          for (Day Now : DayNow)
          {
              switch (Now)
              {
                  case Sun:
                       System.out.println("Sunday");
                     break;
                   case Mon:
                     System.out.println("Monday");
                       break;
                  case Tue:
                       System.out.println("Tuesday");
                     break;
                   case Wed:
                     System.out.println("Wednesday");
                       break;
                  case Thu:
                       System.out.println("Thursday");
                     break;
                   case Fri:
                     System.out.println("Friday");
                       break;
                  case Sat:
                       System.out.println("Saturday");
                     break;
              }
          }
       }
}
```
Test it Now

Output:

```
Sunday
Monday
Twesday
Wednesday
Thursday
Friday
Saturday
```

Java Wrapper in Switch Statement

Java allows us to use four wrapper classes

: Byte, Short, Integer and Long in switch statement.

Example:

WrapperInSwitchCaseExample.java

```java
//Java Program to demonstrate the use of Wrapper class
//in switch statement
public class WrapperInSwitchCaseExample {
      public static void main(String args[])
      {
          Integer age = 18;
           switch (age)
          {
               case (16):
                 System.out.println("You are under 18.");
                   break;
              case (18):
                   System.out.println("You are eligible for vote.");
                 break;
               case (65):
                 System.out.println("You are senior citizen.");
                   break;
              default:
                   System.out.println("Please give the valid age.");
                 break;
           }
      }
}
```
Test it Now

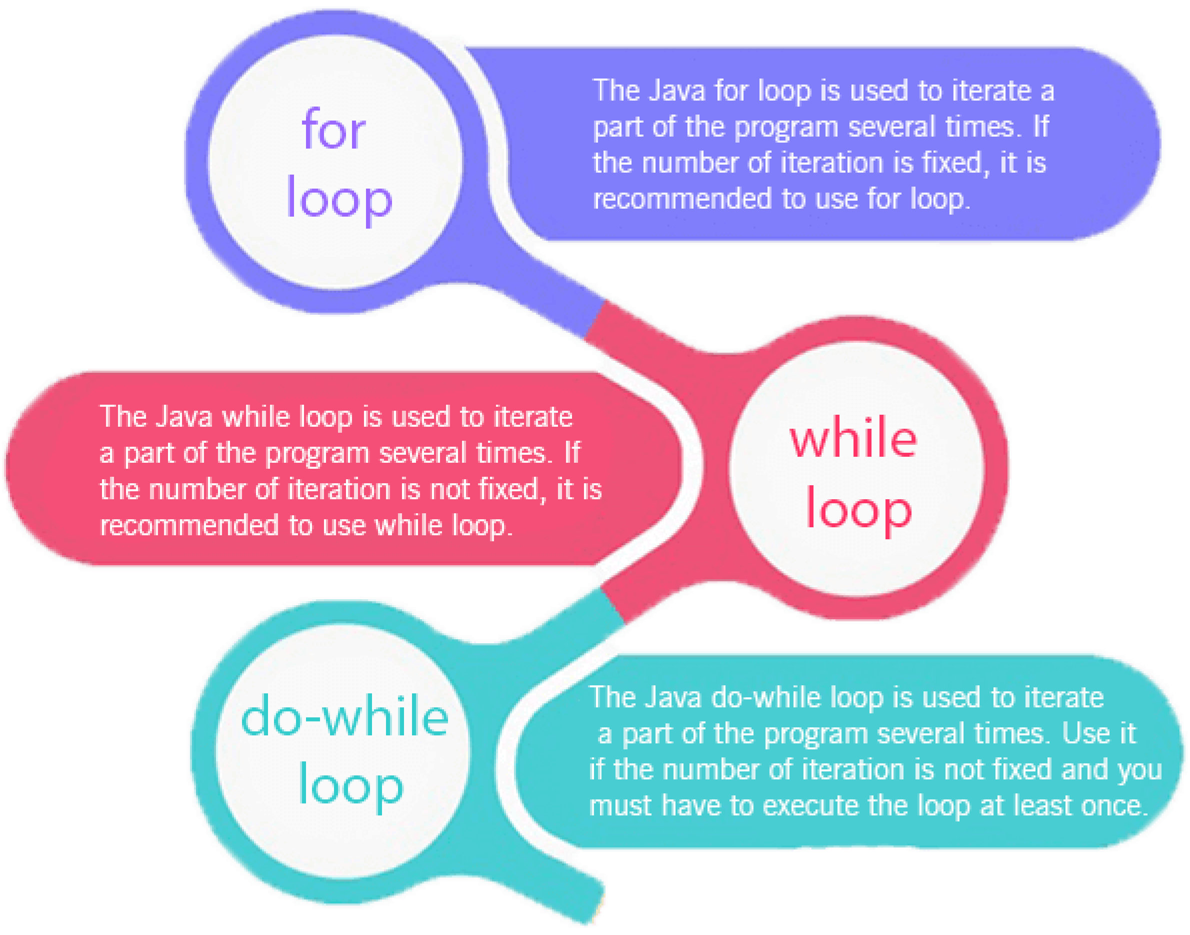
Output:

```
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.


for loop
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.
while loop
The Java while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.
do-while loop
The Java do-while loop is used to iterate a part of the program several times. Use it if the number of iteration is not fixed and you must have to execute the loop at least once.

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

Java Simple for Loop

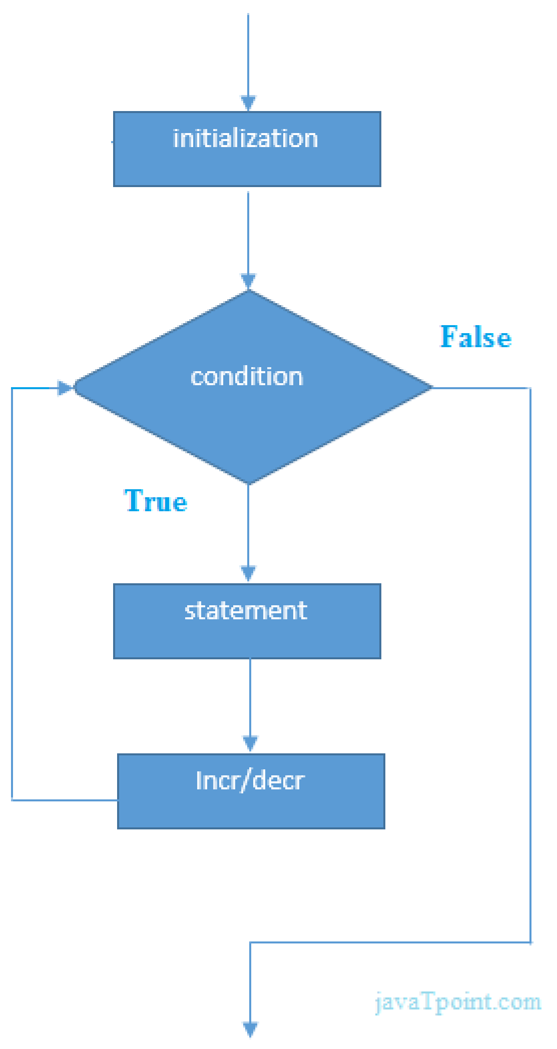
A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

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:

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

Flowchart:



Example:

ForExample.java

```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
public static void main(String[] args) {
    //Code of Java for loop
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
}
}
```

Test it Now

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.

Example:

NestedForExample.java

```
public class NestedForExample {
public static void main(String[] args) {
//loop of i
```

```java
for(int i=1;i<=3;i++){
//loop of j
for(int j=1;j<=3;j++){
        System.out.println(i+" "+j);
}//end of i
}//end of j
}
}
```

Output:

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

Pyramid Example 1:

PyramidExample.java

```java
public class PyramidExample {
public static void main(String[] args) {
for(int i=1;i<=5;i++){
for(int j=1;j<=i;j++){
        System.out.print("* ");
}
System.out.println();//new line
}
}
}
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Pyramid Example 2:

PyramidExample2.java

```java
public class PyramidExample2 {
public static void main(String[] args) {
int term=6;
for(int i=1;i<=term;i++){
for(int j=term;j>=i;j--){
        System.out.print("* ");
}
System.out.println();//new line
}
}
}
```

Output:

```
* * * * * *
* * * * *
* * * *
* * *
* *
*
```

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:

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

Example:

ForEachExample.java

```java
//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);
    }
}
}
```

Test it Now

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:

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

Example:

LabeledForExample.java

```java
//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
```

If you use **break bb;**, it will break inner loop only which is the default behaviour of any loop.

LabeledForExample2.java

```
public class LabeledForExample2 {
public static void main(String[] args) {
    aa:
        for(int i=1;i<=3;i++){
            bb:
                for(int j=1;j<=3;j++){
                    if(i==2&&j==2){
                        break bb;
                    }
                    System.out.println(i+" "+j);
                }
        }
}
}
```

Output:

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

Java Infinitive for Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

Syntax:

```
for(;;){
//code to be executed
}
```

Example:

ForExample.java

```
//Java program to demonstrate the use of infinite for loop
//which prints an statement
public class ForExample {
public static void main(String[] args) {
    //Using no condition in for loop
    for(;;){
        System.out.println("infinitive loop");
    }
  }
}
```

Output:

```
infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
ctrl+c
```

Now, you need to press ctrl+c to exit from the program.

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:

```
while (condition){
//code to be executed
```

3. I ncrement / decrement statement
4. }

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

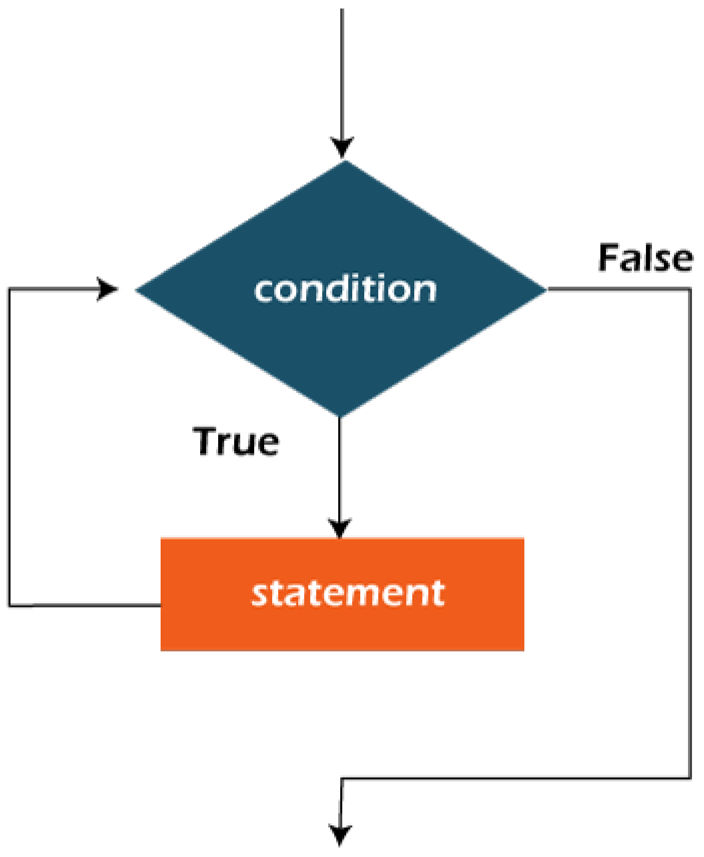
2. Update expression: Every time the loop body is executed, this expression increments or decrements loop variable.

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.

WhileExample.java

```
public class WhileExample {
public static void main(String[] args) {
    int i=1;
    while(i<=10){
        System.out.println(i);
    i++;
    }
}
}
```

Test it Now

Output:

```
1
```

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

Java Infinitive While Loop

If you pass **true** in the while loop, it will be infinitive while loop.

Syntax:

```
while(true){
//code to be executed
}
```

Example:

WhileExample2.java

```
public class WhileExample2 {
public static void main(String[] args) {
 // setting the infinite while loop by passing true to the condition
    while(true){
        System.out.println("infinitive while loop");
    }
}
}
```

Output:

```
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
ctrl+c
```

In the above code, we need to enter Ctrl + C command to terminate the infinite loop.

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:

```
do{
//code to be executed / loop body
//update statement
}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

```java
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

Test it Now

Output:

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

Java Infinitive do-while Loop

If you pass **true** in the do-while loop, it will be infinitive do-while loop.

Syntax:

```java
do{
//code to be executed
}while(true);
```

Example:

DoWhileExample2.java

```java
public class DoWhileExample2 {
public static void main(String[] args) {
    do{
        System.out.println("infinitive do while loop");
    }while(true);
}
}
```

Output:

```
infinitive do while loop
infinitive do while loop
infinitive do while loop
ctrl+c
```

In the above code, we need to enter Ctrl + C command to terminate the infinite loop.

Java Break Statement

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

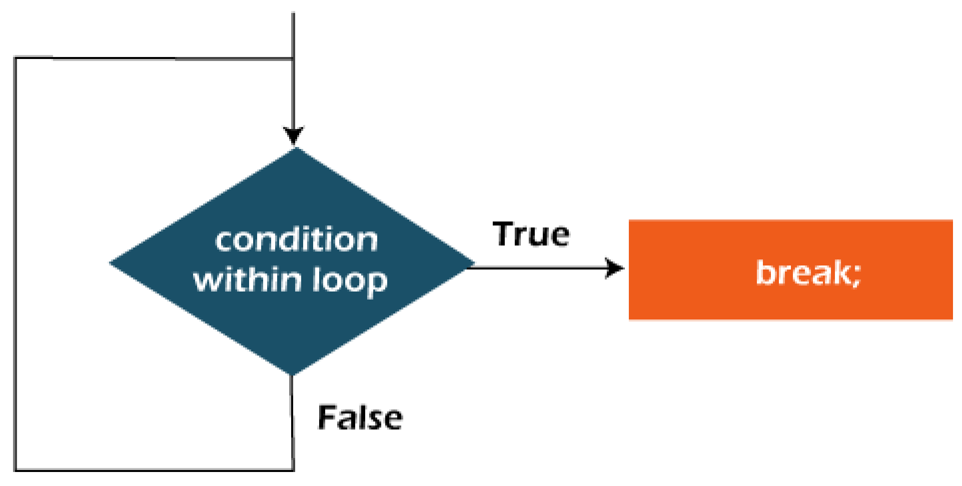
The Java *break* statement is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

```
jump-statement;
break;
```

Flowchart of Break Statement



Flowchart of break statement

Java Break Statement with Loop

Example:

BreakExample.java

```
//Java Program to demonstrate the use of break statement
//inside the for loop.
public class BreakExample {
public static void main(String[] args) {
    //using for loop
    for(int i=1;i<=10;i++){
        if(i==5){
            //breaking the loop
            break;
        }
        System.out.println(i);
    }
}
}
```

Output:

```
1
2
3
4
```

Java Break Statement with Inner Loop

It breaks inner loop only if you use break statement inside the inner loop.

Example:

BreakExample2.java

```java
//Java Program to illustrate the use of break statement
//inside an inner loop
public class BreakExample2 {
public static void main(String[] args) {
            //outer loop
            for(int i=1;i<=3;i++){
                    //inner loop
                    for(int j=1;j<=3;j++){
                        if(i==2&&j==2){
                            //using break statement inside the inner loop
                            break;
                        }
                        System.out.println(i+" "+j);
                    }
            }
}
}
```

Output:

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

Java Break Statement with Labeled For Loop

We can use break statement with a label. The feature is introduced since JDK 1.5. So, we can break any loop in Java now whether it is outer or inner loop.

Example:

BreakExample3.java

```java
//Java Program to illustrate the use of continue statement
//with label inside an inner loop to break outer loop
public class BreakExample3 {
public static void main(String[] args) {
            aa:
            for(int i=1;i<=3;i++){
                    bb:
                    for(int j=1;j<=3;j++){
                        if(i==2&&j==2){
                            //using break statement with label
                            break aa;
                        }
                        System.out.println(i+" "+j);
                    }
            }
}
}
```

Output:

```
1 1
1 2
```

```
1 3
2 1
```

Java Break Statement in while loop

Example:

BreakWhileExample.java

```java
//Java Program to demonstrate the use of break statement
//inside the while loop.
public class BreakWhileExample {
public static void main(String[] args) {
    //while loop
    int i=1;
    while(i<=10){
        if(i==5){
            //using break statement
            i++;
            break;//it will break the loop
        }
        System.out.println(i);
        i++;
    }
}
}
```

Output:

```
1
2
3
4
```

Java Break Statement in do-while loop

Example:

BreakDoWhileExample.java

```java
//Java Program to demonstrate the use of break statement
//inside the Java do-while loop.
public class BreakDoWhileExample {
public static void main(String[] args) {
    //declaring variable
    int i=1;
    //do-while loop
    do{
        if(i==5){
           //using break statement
           i++;
           break;//it will break the loop
        }
        System.out.println(i);
        i++;
    }while(i<=10);
}
}
```

Output:

```
1
2
3
4
```

Java Break Statement with Switch

To understand the example of break with switch statement, please visit here: Java Switch Statement.

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:

```
jump-statement;
continue;
```

Java Continue Statement Example

ContinueExample.java

```java
//Java Program to demonstrate the use of continue statement
//inside the for loop.
public class ContinueExample {
public static void main(String[] args) {
    //for loop
    for(int i=1;i<=10;i++){
        if(i==5){
            //using continue statement
            continue;//it will skip the rest statement
        }
        System.out.println(i);
    }
}
}
```

Test it Now

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

```java
//Java Program to illustrate the use of continue statement
//inside an inner loop
public class ContinueExample2 {
public static void main(String[] args) {
            //outer loop
```

```java
        for(int i=1;i<=3;i++){
            //inner loop
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    //using continue statement inside inner loop
                    continue;
                }
                System.out.println(i+" "+j);
            }
        }
}
}
```

Output:

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

Java Continue Statement with Labelled For Loop

We can use continue statement with a label. This feature is introduced since JDK 1.5. So, we can continue any loop in Java now whether it is outer loop or inner.

Example:

ContinueExample3.java

```java
//Java Program to illustrate the use of continue statement
//with label inside an inner loop to continue outer loop
public class ContinueExample3 {
public static void main(String[] args) {
        aa:
        for(int i=1;i<=3;i++){
            bb:
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    //using continue statement with label
                    continue aa;
                }
                System.out.println(i+" "+j);
            }
        }
}
}
```

Output:

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

Java Continue Statement in while loop

ContinueWhileExample.java

```java
//Java Program to demonstrate the use of continue statement
//inside the while loop.
```

```java
public class ContinueWhileExample {
public static void main(String[] args) {
    //while loop
    int i=1;
    while(i<=10){
        if(i==5){
            //using continue statement
            i++;
            continue;//it will skip the rest statement
        }
        System.out.println(i);
        i++;
    }
}
}
```

Test it Now

Output:

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

Java Continue Statement in do-while Loop

ContinueDoWhileExample.java

```java
//Java Program to demonstrate the use of continue statement
//inside the Java do-while loop.
public class ContinueDoWhileExample {
public static void main(String[] args) {
    //declaring variable
    int i=1;
    //do-while loop
    do{
        if(i==5){
                //using continue statement
                 i++;
            continue;//it will skip the rest statement
        }
        System.out.println(i);
        i++;
    }while(i<=10);
}
}
```

Test it Now

Output:

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

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

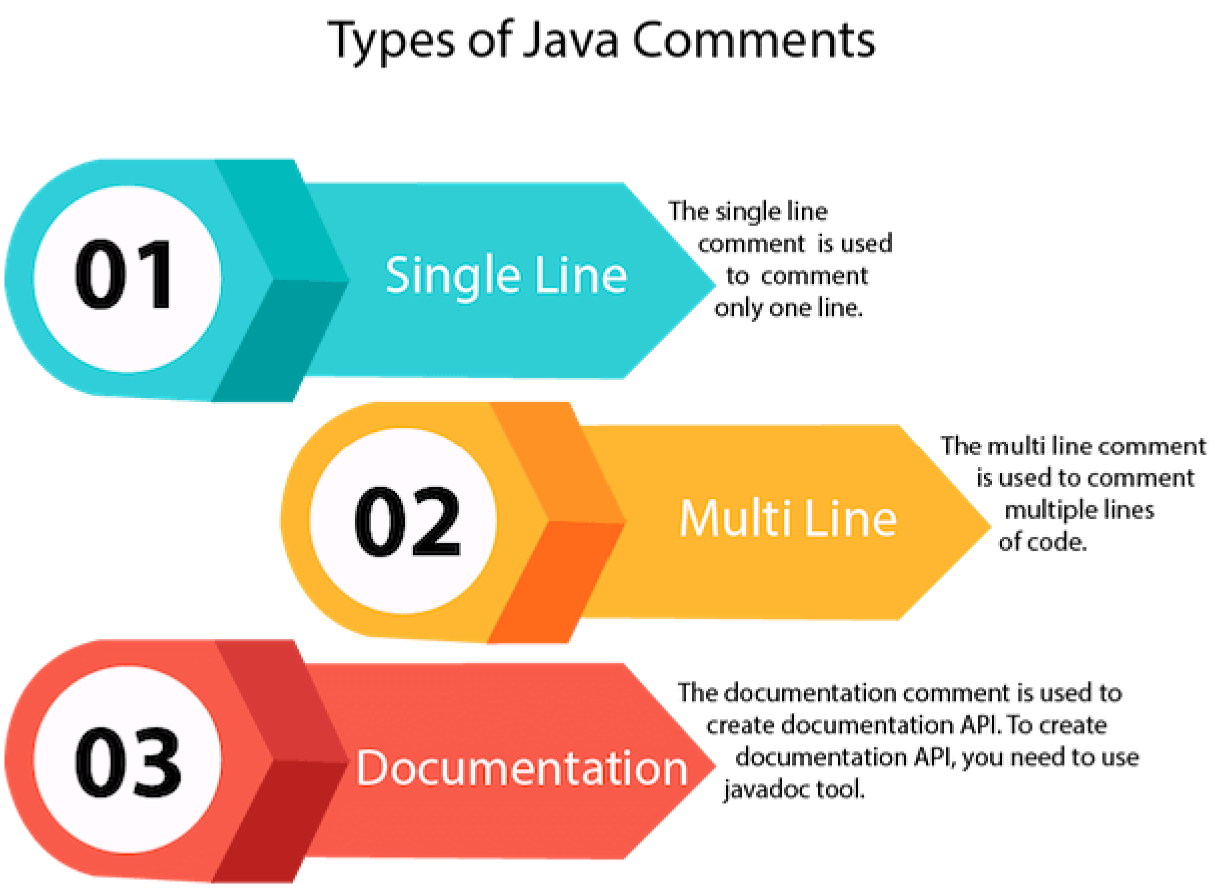
  , method, class

  , 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


Types of Java Comments
01
Single Line
The single line comment is used to comment only one line.
02
Multi Line
The multi line comment is used to comment multiple lines of code.
03
Documentation
The documentation comment is used to create documentation API. To create documentation API, you need to use javadoc tool.

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:

```
//This is single line comment
```

Let's use single line comment in a Java program.

CommentExample1.java

```java
public class CommentExample1 {
public static void main(String[] args) {
int i=10; // i is a variable with value 10
System.out.println(i);  //printing the variable i
}
}
```

Output:

```
10
```

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:

```java
/*
This
is
multi line
comment
*/
```

Let's use multi-line comment in a Java program.

CommentExample2.java

```java
public class CommentExample2 {
public static void main(String[] args) {
/* Let's declare and
 print variable in java. */
  int i=10;
    System.out.println(i);
/* float j = 5.9;
    float k = 4.4;
    System.out.println( j + k ); */
}
}
```

Output:

```
10
```

Note: Usually // is used for short comments and /* */ is used for longer comments.

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
*
*/
```

javadoc tags

Some of the commonly used tags in documentation comments:

Tag	Syntax	Description
{@docRoot}	{@docRoot}	to depict relative path to root directory of generated document from any page.
@author	@author name - text	To add the author of the class.
@code	{@code text}	To show the text in code font without interpreting it as html markup or nested javadoc tag.
@version	@version version-text	To specify "Version" subheading and version-text when -version option is used.
@since	@since release	To add "Since" heading with since text to generated documentation.
@param	@param parameter-name description	To add a parameter with given name and description to 'Parameters' section.
@return	@return description	Required for every method that returns something (except void)

Let's use the Javadoc tag in a Java program.

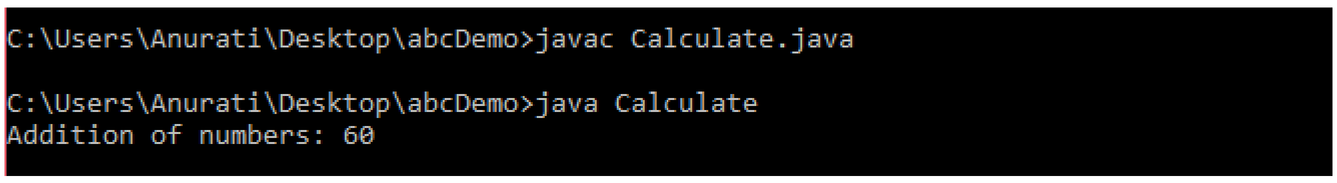
Calculate.java

```
import java.io.*;
```

```java

/**
 * <h2> Calculation of numbers </h2>
 * This program implements an application
 * to perform operation such as addition of numbers
 * and print the result
 * <p>
 * <b>Note:</b> Comments make the code readable and
 * easy to understand.
 *
 * @author Anurati
 * @version 16.0
 * @since 2021-07-06
 */

public class Calculate{
   /**
    * This method calculates the summation of two integers.
    * @param input1 This is the first parameter to sum() method
    * @param input2 This is the second parameter to the sum() method.
    * @return int This returns the addition of input1 and input2
    */
   public int sum(int input1, int input2){
       return input1 + input2;
   }
   /**
   * This is the main method uses of sum() method.
   * @param args Unused
   * @see IOException
   */
   public static void main(String[] args) {
       Calculate obj = new Calculate();
       int result = obj.sum(40, 20);

       System.out.println("Addition of numbers: " + result);
   }
}
```

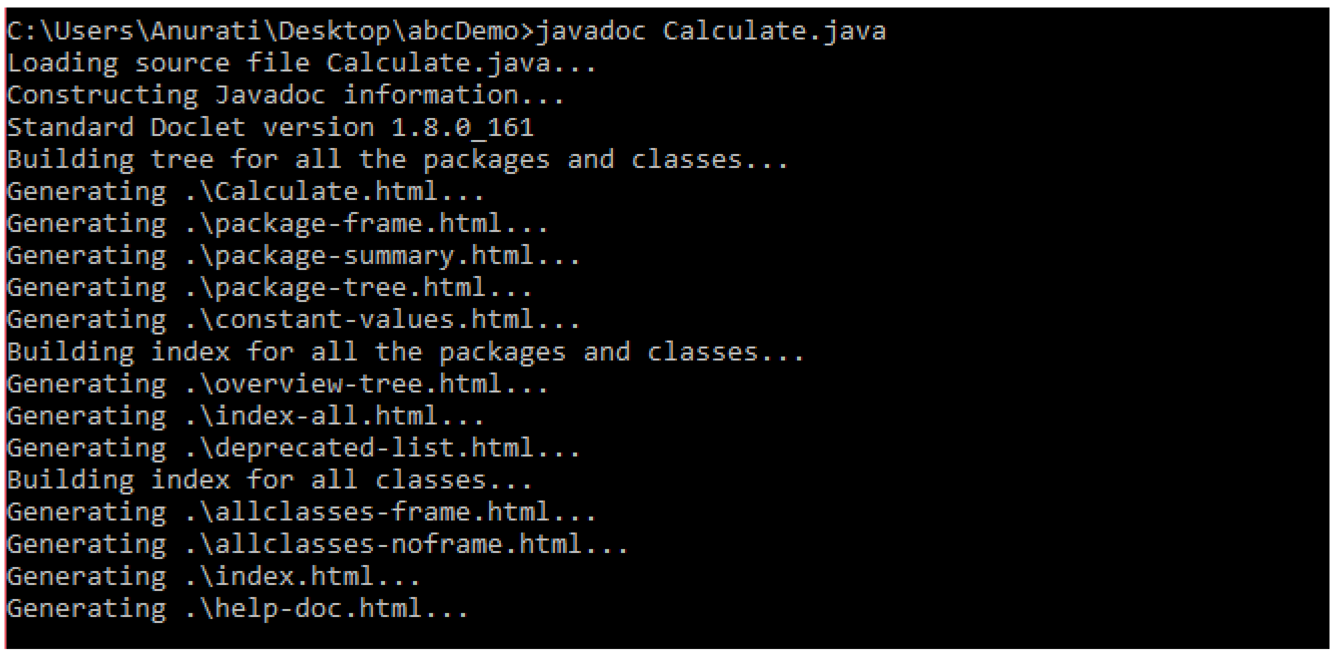
Compile it by javac tool:

Create Document



```
C:\Users\Anurati\Desktop\abcDemo>javac Calculate.java

C:\Users\Anurati\Desktop\abcDemo>java Calculate
Addition of numbers: 60
```

Create documentation API by **javadoc** tool:


```
C:\Users\Anurati\Desktop\abcDemo>javadoc Calculate.java
Loading source file Calculate.java...
Constructing Javadoc information...
Standard Doclet version 1.8.0_161
Building tree for all the packages and classes...
Generating .\Calculate.html...
Generating .\package-frame.html...
Generating .\package-summary.html...
Generating .\package-tree.html...
Generating .\constant-values.html...
Building index for all the packages and classes...
Generating .\overview-tree.html...
Generating .\index-all.html...
Generating .\deprecated-list.html...
Building index for all classes...
Generating .\allclasses-frame.html...
Generating .\allclasses-noframe.html...
Generating .\index.html...
Generating .\help-doc.html...
```

Now, the HTML

files are created for the **Calculate** class in the current directory, i.e., **abcDemo**. Open the HTML files, and we can see the explanation of Calculate class provided through the documentation comment.

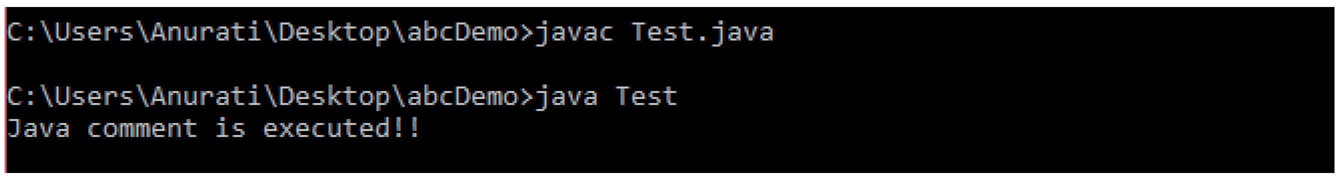
Are Java comments executable?

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.

Test.java

```
public class Test{
   public static void main(String[] args) {
       //the below comment will be executed
// \u000d System.out.println("Java comment is executed!!");
   }
}
```

Output:


```
C:\Users\Anurati\Desktop\abcDemo>javac Test.java

C:\Users\Anurati\Desktop\abcDemo>java Test
Java comment is executed!!
```

The above code generate the output because the compiler parses the Unicode character **u000d** as a **new line** before the lexical transformation, and thus the code is transformed as shown below:

Test.java

```
public class Test{
   public static void main(String[] args) {
       //the below comment will be executed
//
System.out.println("Java comment is executed!!");
   }
}
```

Thus, the Unicode character shifts the print statement to next line and it is executed as a normal Java code.