



Java is one of the world's most important and widely used computer languages, and it has held this distinction for many years. Unlike some other computer languages whose influence has wearied with passage of time, while Java's has grown.

Java is a high level, robust, object-oriented and a secure and stable programming language but it is not a pure object-oriented language because it supports primitive data types like int, char etc.

Java is a platform-independent language because it has runtime environment i.e JRE and API. Here platform means a hardware or software environment in which an application runs.

Java codes are compiled into byte code or machine-independent code. This byte code is run on JVM (Java Virtual Machine).

The syntax in Java is almost the same as C/C++. But Java does not support low-level programming functions like pointers. The codes in Java are always written in the form of Classes and objects.

As of 2020, Java is one of the most popular programming languages in use, especially for client-server web applications. It has been estimated that there are around nine million Java developers inside the world.

Creation of Java

Java was developed by James Gosling, Patrick Naughton, Mike Sheridan at Sun Microsystems Inc. in 1991. It took 18 months to develop the first working version.

The initial name was **Oak** but it was renamed to **Java** in 1995 as OAK was a registered trademark of another Tech company.

History of Java

Originally Java was designed for Interactive television, but this technology was very much advanced for the industry of digital cable television at that time. Java history was started with the **Green Team**.

The Green Team started a project to develop a language for digital devices such as television. But it works best for internet programming. After some time Java technology was joined by Netscape.

The objective to create Java Programming Language was it should be "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Object-Oriented, Interpreted, and Dynamic".

Java was developed in Sun Microsystem by James Gosling, Patrick Naughton, Mike Sheridan in 1991. It took 18 months to develop the first working version. James Gosling is also known as the Father of Java.

Initially, Java was called "**Greentalk**" by James Gosling and at that time the file extension was **.gt**.

Later on Oak was developed as a part of the Green Team project. Oak is a symbol for strength and Oak is also a national tree in many countries like the USA, Romania etc.

Oak was renamed as Java in 1995 because Oak was already a trademark by Oak Technologies. Before selecting the Java word the team suggested many names

like **dynamic, revolutionary, Silk, jolt, DNA**, etc.

Java is an island in Indonesia, here the first coffee was produced or we call Java coffee. Java coffee is a type of espresso bean. James Gosling chose this name while having coffee near his office.

The word JAVA does not have an acronym. It is just a name.

In 1995 Java was one of the best product by the Time magazine.

Java Version History

Here we have listed down all the versions of Java along with the main features introduced in those versions.

Version Name	Coad Name	Release Date	Description
Java Alpha and Beta		1995	<ul style="list-style-type: none"> • It was the 1st version but was having unstable APIs and ABIs. • It was the 1st version but was having unstable APIs and ABIs.
JDK 1.0	Oak	January 1996	<ul style="list-style-type: none"> • 1st stable version
JDK 1.1		February 1997	<ul style="list-style-type: none"> • AWT Event modelling retooling. • Added Inner class, Java Beans, JDBC, RMI, Reflection, JIT • Added Inner class, Java Beans, JDBC, RMI, Reflection, JIT
J2SE 1.2	Playground	December 1998	<ul style="list-style-type: none"> • JDK replaced by J2SE. • Support strictfp keyword. • Swing API integrated with core classes. • Collection framework.

J2SE 1.3	Kestrel	May 2000	<ul style="list-style-type: none"> • HotSPot JVM included • RMI Modified. • JNDI(Java Naming and Directory Interface) Supported • JPDA(Java Platform Debugger Architecture). • Included Proxy Classes.
J2SE 1.4	Merlin	February 2002	<ul style="list-style-type: none"> • Support assert Keyword. • Improvement in libraries. • Support Regular expression. • Support Exception Chaining. • Support Exception Chaining. • Included Java Web Start. • Support API Preferences (java.util.prefs).
J2SE 5.0	Tiger	September 2004	<ul style="list-style-type: none"> • Included Generics, Metadata, Autoboxing/Unboxing, Enumerations, Varargs. • Enhanced for each loop. • Support static imports.
Java SE 6	Mustang	December 2006	<ul style="list-style-type: none"> • Support Win9x version. • Support Scripting languages. • Improved Swing performance. • Support JDBC 4.0 • Upgrade of JAXB to 2.0. • Improvement in GUI and JVM.

Java SE 7	Dolphine	July 2011	<ul style="list-style-type: none"> • Support of dynamic language in JVM. • Included 64-bit pointers. • Support string in the switch. • Support resource management in the try block. • Support binary integer literals. • Support underscore in numeric literals. • Support multiple exceptions. • Included I/O library.
Java SE 8(LTS)		March 2014	<ul style="list-style-type: none"> • Support of JSR 335 and JEP 126. • Support unsigned integer. • Support Date and time API. • Included JavaFX. • Support Windows XP.

Java SE 9		September 2017	<ul style="list-style-type: none"> • Support multiple gigabyte heaps. • Included garbage collector.
Java SE 10		March 2018	<ul style="list-style-type: none"> • Support local variables type inference. • Support local variables type inference. • Included Application class.
Java SE 11(LTS)		September 2018	<ul style="list-style-type: none"> • Support bug fixes. • Include long term support(LTS). • Support transport layer security.
Java SE 12		March 2019	<ul style="list-style-type: none"> • Support JVM Constant API. • Include CDS Archives.
Java SE 13		September 2019	<ul style="list-style-type: none"> • Updated Switch Expressions. • Include Text Blocks. • Support Legacy socket API.
Java SE 14		March 2020	<ul style="list-style-type: none"> • Support Event Streaming. • Improved NullPointerException. • Removal of the Concurrent Mark Sweep (CMS) in the garbage collector.
Java SE 15		September 2020	
Java SE 16		March 2021	
Java SE 17(LTS)		September 2021	

Evolution of Java

Java was initially launched as Java 1.0 but soon after its initial release, Java 1.1 was launched. Java 1.1 redefined event handling, new library elements were added.

In **Java 1.2** Swing and Collection framework was added and `suspend()`, `resume()` and `stop()` methods were deprecated from **Thread** class.

No major changes were made into **Java 1.3** but the next release that was **Java 1.4** contained several important changes. Keyword `assert`, chained exceptions and channel based I/O System was introduced.

Java 1.5 was called **J2SE 5**, it added following major new features :

- Generics
- Annotations
- Autoboxing and autounboxing
- Enumerations
- For-each Loop
- Varargs
- Static Import
- Formatted I/O
- Concurrency utilities

Next major release was **Java SE 7** which included many new changes, like :

- Now **String** can be used to control Switch statement.
- Multi Catch Exception
- *try-with-resource* statement
- Binary Integer Literals
- *Underscore* in numeric literals, etc.

Java SE 8 was released on March 18, 2014. Some of the major new features introduced in JAVA 8 are,

- Lambda Expressions
- New Collection Package `java.util.stream` to provide Stream API.
- Enhanced Security
- Nashorn Javascript Engine included

- Parallel Array Sorting
- The JDBC-ODBC Bridge has been removed etc.

Java SE 9 was released on September 2017. Some of the major new features introduced in JAVA 9 are,

- Platform Module System (Project Jigsaw)
- Interface Private Methods
- Try-With Resources
- Anonymous Classes
- @SafeVarargs Annotation
- Collection Factory Methods
- Process API Improvement

Java SE 10 was released on March 2018. Some of the major new features introduced in JAVA 10 are,

- Support local variables type inference.
- Support local variables type inference.
- Included Application class.

Java SE 11 was released on September 2018. Some of the major new features introduced in JAVA 11 are,

- Support bug fixes.
- Include long term support(LTS).
- Support transport layer security.

Java SE 12 was released on March 2019. Some of the major new features introduced in JAVA 12 are,

- Support JVM Constant API.
- Include CDS Archives.

Java SE 13 was released on September 2019. Some of the major new features introduced in JAVA 13 are,

- Updated Switch Expressions.
- Include Text Blocks.
- Support Legacy socket API.

Java SE 14 was released on March 2020. Some of the major new features introduced in JAVA 14 are,

- Support Event Streaming.
 - Improved NullPointerException.
 - Removal of the Concurrent Mark Sweep (CMS) in the garbage collector.
-

Application of Java

Java is widely used in every corner of world and of human life. Java is not only used in softwares but is also widely used in designing hardware controlling software components. There are more than 930 million JRE downloads each year and 3 billion mobile phones run java. Following are some other usage of Java :

1. Developing Desktop Applications
 2. Web Applications like Linkedin.com, Snapdeal.com etc
 3. Mobile Operating System like Android
 4. Embedded Systems
 5. Robotics and games etc.
-

Types of Java Application

Following are different types of applications that we can develop using Java:

1. Standalone Applications

Standalone applications are the application which runs on separate computer process without adding any file processes. The standalone application is also known as Java GUI Applications or **Desktop Applications** which uses some standard GUI components such as AWT(Abstract Windowing Toolkit), swing and JavaFX and this component are deployed to the desktop. These components have buttons, menu, tables, GUI widget toolkit, 3D graphics etc. using this component a traditional software is developed which can be installed in every machine.

Example: Media player, antivirus, Paint, POS Billing software, etc.

2. Web Applications

Web Applications are the client-server software application which is run by the client. Servlets, struts, JSP, Spring, hibernate etc. are used for the development of a client-server application. eCommerce application is also developed in java using eCommerce platform i.e Broadleaf.

Example: mail, e-commerce website, bank website etc.

3. Enterprise Application

Enterprise application is middleware applications. To use software and hardware systems technologies and services across the enterprises. It is designed for the corporate area such as banking business systems.

Example: e-commerce, accounting, banking information systems etc.

4. Mobile Application

For mobile applications, Java uses ME or J2ME framework. This framework are the cross platform that runs applications across phones and smartphones. Java provides a platform for **application development in Android** too.

Example: WhatsApp, Xender etc.

Download JDK

For running Java programs in your system you will have to download and install [JDK kit from here](#) (recommended Java Version is Java 11 but you can download Java 13 or Java 14).

Features of Java

The prime reason behind creation of Java was to bring portability and security feature into a computer language. Beside these two major features, there were many other features that played an important role in moulding out the final form of this outstanding language. Those features are :

1) Simple

Java is easy to learn and its syntax is quite simple, clean and easy to understand. The confusing and ambiguous concepts of C++ are either left out in Java or they have been re-implemented in a cleaner way.

Eg : Pointers and Operator Overloading are not there in java but were an important part of C++.

2) Object Oriented

In java, everything is an object which has some data and behaviour.

Java can be easily extended as it is based on Object Model. Following are some basic concept of OOP's.

- i. Object
- ii. Class
- iii. Inheritance
- iv. Polymorphism
- v. Abstraction
- vi. Encapsulation

3) Robust

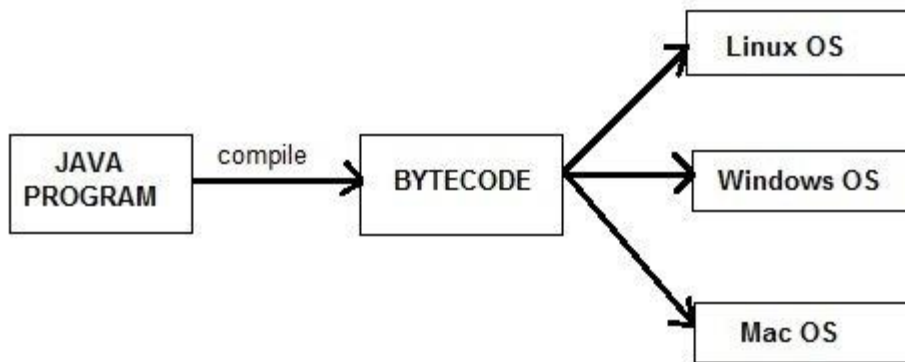
Java makes an effort to eliminate error prone codes by emphasizing mainly on compile time error checking and runtime checking. But the main areas which Java improved were Memory Management and mishandled Exceptions by introducing automatic **Garbage**

Collector and **Exception Handling**.

4) Platform Independent

Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be write-once, run-anywhere language.

On compilation Java program is compiled into bytecode. This bytecode is platform independent and can be run on any machine, plus this bytecode format also provide security. Any machine with Java Runtime Environment can run Java Programs.



5) Secure

When it comes to security, Java is always the first choice. With java secure features it enable us to develop virus free, temper free system. Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.

6) Multi Threading

Java multithreading feature makes it possible to write program that can do many tasks simultaneously. Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time, like While typing, grammatical errors are checked along.

7) Architectural Neutral

Compiler generates bytecodes, which have nothing to do with a particular computer architecture, hence a Java program is easy to intrepret on any machine.

8) Portable

Java Byte code can be carried to any platform. No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types

9) High Performance

Java is an interpreted language, so it will never be as fast as a compiled language like C or C++. But, Java enables high performance with the use of just-in-time compiler.

10) Distributed

Java is also a distributed language. Programs can be designed to run on computer networks. Java has a special class library for communicating using TCP/IP protocols. Creating network connections is very much easy in Java as compared to C/C++.

New Features of JAVA 8

Below mentioned are some of the core upgrades done as a part of Java 8 release. Just go through them quickly, we will explore them in details later.

- Enhanced Productivity by providing Optional Classes feature, Lambda Expressions, Streams etc.
- Ease of Use
- Improved Polyglot programming. A **Polyglot** is a program or script, written in a form which is valid in multiple programming languages and it performs the same operations in multiple programming languages. So Java now supports such type of programming technique.
- Improved Security and performance.

New Features of JAVA 11

Java 11 is a recommended LTS version of Java that includes various important features. These features includes new and upgrades in existing topic. Just go through them quickly, we will explore them in details later.

- includes support for Unicode 10.0.0
- The HTTP Client has been standardized
- Lazy Allocation of Compiler Threads
- Updated Locale Data to Unicode CLDR v33
- JEP 331 Low-Overhead Heap Profiling
- JEP 181 Nest-Based Access Control
- Added Brainpool EC Support (RFC 5639)
- Enhanced KeyStore Mechanisms
- JEP 332 Transport Layer Security (TLS) 1.3

- JEP 330 Launch Single-File Source-Code Programs

Java Editions

Java Editions or we can say the platform is a collection of programs which helps to develop and run the programs that are written in Java Programming language. Java Editions includes execution engine, compiler and set of libraries. As Java is Platform independent language so it is not specific to any processor or operating system.

1. Java Standard Edition

Java Standard edition is a computing platform which is used for development and deployment of portable code that is used in desktop and server environments. Java Standard Edition is also known as Java 2 Platform, Standard Edition (J2SE).

Java Standard Edition has a wide range of APIs such as Java Class Library etc. the best implementation of Java SE is Oracle Corporation's Java Development Kit (JDK).

2. Java Micro Edition

Java Micro Edition is a computing platform which is used for the development and deployment of portable codes for the embedded and mobile devices. Java Micro Edition is also known as Java 2 Platform Micro Edition (J2ME). The Java Micro Edition was designed by Sun Microsystems and then later on Oracle corporation acquired it in 2010.

Example: micro-controllers, sensors, gateways, mobile phones, printers etc.

3. Java Enterprise Edition

Java Enterprise Edition is a set of specifications and extending Java SE 8 with features such as distributed computing and web services. The applications of Java Enterprise Edition run on reference runtimes. This reference runtime handle transactions, security, scalability, concurrency and the management of components to be deployed. Java Enterprise Edition is also known as Java 2 Platform Enterprise Edition (J2EE), and currently, it has been rebranded as Jakarta EE.

Example: e-commerce, accounting, banking information systems.

4. JavaFX

JavaFX is used for creating desktop applications and also rich internet applications(RIAs) which can be run on a wide variety of devices.

JavaFX has almost replaced Swing as the standard GUI library for Java Standard Edition. JavaFX support for desktop computers and web browsers.

Setting Java Environment and Classpath

An **Environment variable** is a dynamic "object" on a computer that stores a value(like a key-value pair), which can be referenced by one or more software programs in Windows. Like for Java, we will set an environment variable with name "**java**" and its value will be the path of the **/bin** directory present in Java directory. So whenever a program will require Java environment, it will look for the **java** environment variable which will give it the path to the execution directory.

Setting up path for windows (2000/XP/vista/Window 7,8)

Assuming that you have installed Java in **C:\ Program files/ Java / JDK directory**

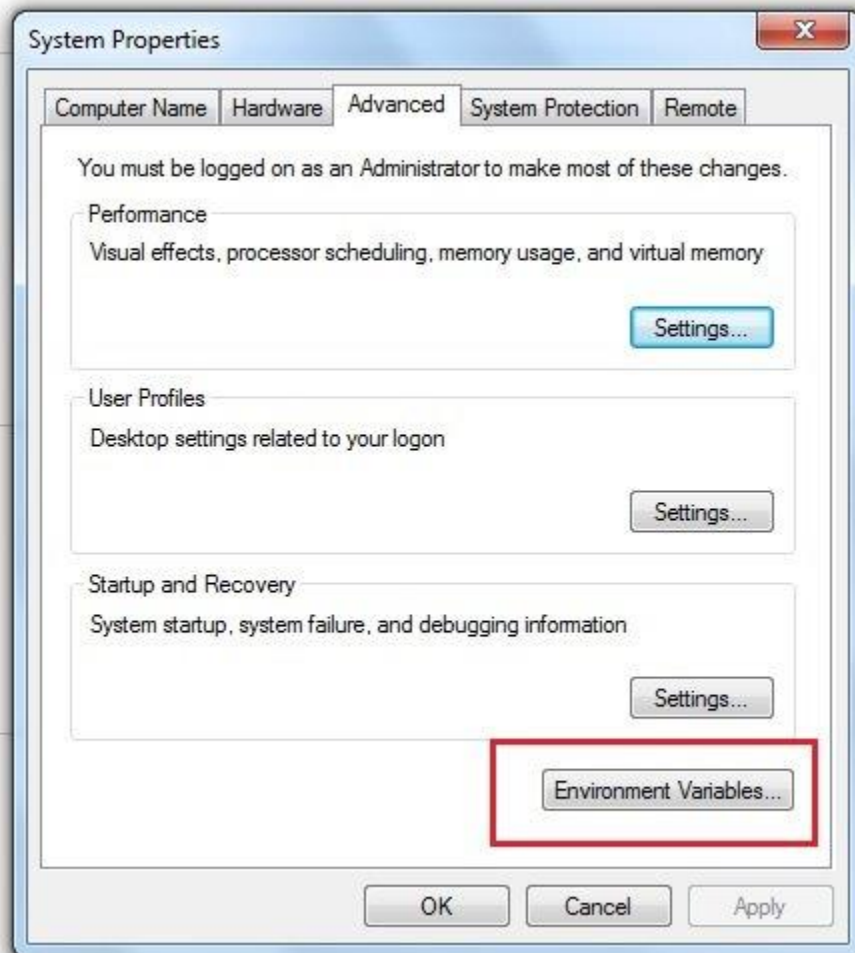
Step 1: Right click on my computer and select properties.



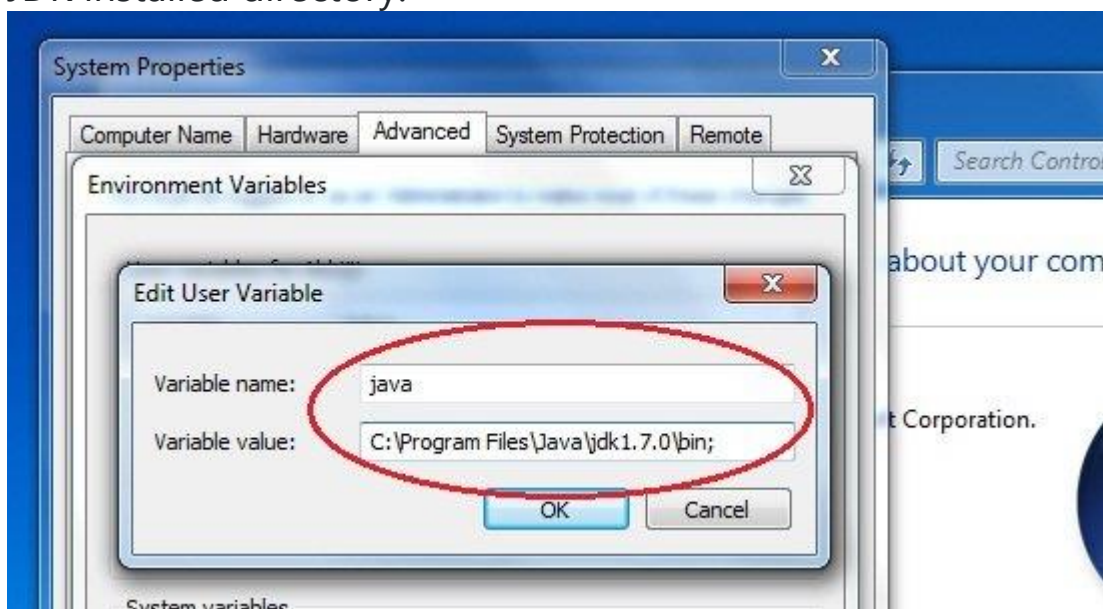
Step 2: Go to the Advance System Settings tab.



Step 3: Click on Environment Variables button.



Step 4: Now alter the path variable so that it also contains the path to JDK installed directory.



For e.g:- Change **C:\windows/ system 32.** to **C:\windows/system 32;**
C:\program files / Java/ JDK.

Setting up path for window 95/98/ME

Assuming that you have installed Java in C:\program files\ java\ JDK directory, do the following:

Step 1: Edit the **C:\autoexec.bat** file and add the following line at the end.

```
SET PATH =% PATH% C:\ PROGRAM FILE/JAVA/JDK/bin
```

Setting up path for Linux , Unix , Solaris, free BSD

Step 1: Environment variable path should be set to point where java binaries have been installed. Refer to your shell if you have trouble doing this.

For Example: If you use bash as your shell, then you would add following line to the end

```
bash mc: export PATH=/ Path/to/java
```

We recommend you to download latest Java 11 version. Java 11 is a LTS(Long Term Support) version and currently widely in used. You can refer our step by step installation guide to install the Java 11.

Recommended Java Version: [Download the latest version of JDK](#)

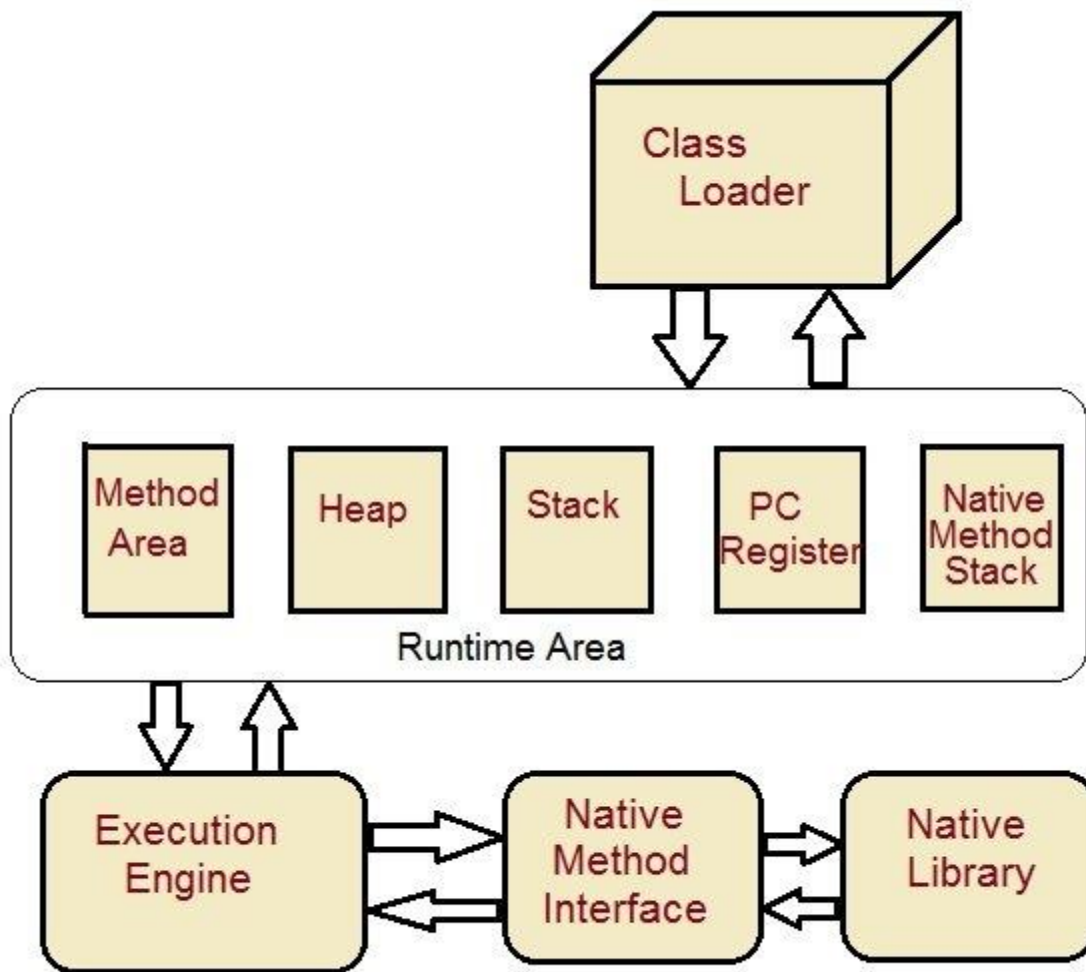
Java JVM, JDK and JRE

In this tutorial we will cover what Java Virtual Machine is, and what is JRE and JDK.

Java virtual Machine(JVM) is a virtual Machine that provides runtime environment to execute java byte code. The JVM doesn't understand Java typo, that's why you compile your *.java files to obtain *.class files that contain the bytecodes understandable by the JVM.

JVM control execution of every Java program. It enables features such as automated exception handling, Garbage-collected heap.

JVM Architecture



Class Loader : Class loader loads the Class for execution.

Method area : Stores pre-class structure as constant pool.

Heap : Heap is a memory area in which objects are allocated.

Stack : Local variables and partial results are store here. Each thread has a private JVM stack created when the thread is created.

Program register : Program register holds the address of JVM instruction currently being executed.

Native method stack : It contains all native used in application.

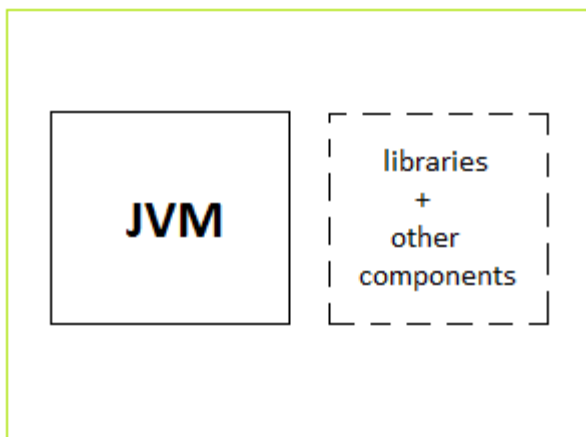
Executive Engine : Execution engine controls the execute of instructions contained in the methods of the classes.

Native Method Interface : Native method interface gives an interface between java code and native code during execution.

Native Method Libraries : Native Libraries consist of files required for the execution of native code.

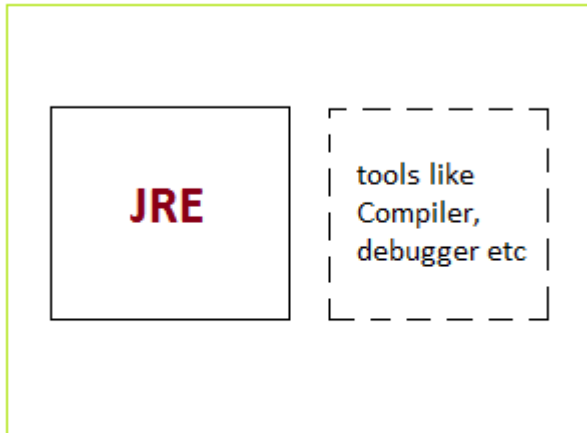
Difference between JDK and JRE

JRE : The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language. JRE does not contain tools and utilities such as compilers or debuggers for developing applets and applications.



JRE - Java Runtime Environment

JDK : The JDK also called Java Development Kit is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.



JDK - Java Development Kit

Java Hello World! Program

Creating a Hello World Program in Java is not a single line program. It consists of various other lines of code. Since Java is a Object-oriented language so it require to write a code inside a class. Let us look at a simple java program.

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

Copy

Lets understand what above program consists of and its keypoints.

class : class keyword is used to declare classes in Java

public : It is an access specifier. Public means this function is visible to all.

static : static is again a keyword used to make a function static. To execute a static function you do not have to create an Object of the class. The **main()** method here is called by JVM, without creating any object for class.

void : It is the return type, meaning this function will not return anything.

main : main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method. If a java class is not having a main() method, it causes compilation error.

String[] args : This represents an array whose type is String and name is args. We will discuss more about array in Java Array section.

System.out.println : This is used to print anything on the console like *printf* in C language.

Steps to Compile and Run your first Java program

Step 1: Open a text editor and write the code as above.

Step 2: Save the file as Hello.java

Step 3: Open command prompt and go to the directory where you saved your first java program assuming it is saved in C drive.

Step 4: Type `javac Hello.java` and press Return(**Enter KEY**) to compile your code. This command will call the Java Compiler asking it to compile the specified file. If there are no errors in the code the command prompt will take you to the next line.

Step 5: Now type `java Hello` on command prompt to run your program.

Step 6: You will be able to see **Hello world program** printed on your command prompt.

Ways to write a Java Program

Following are some of the ways in which a Java program can be written:

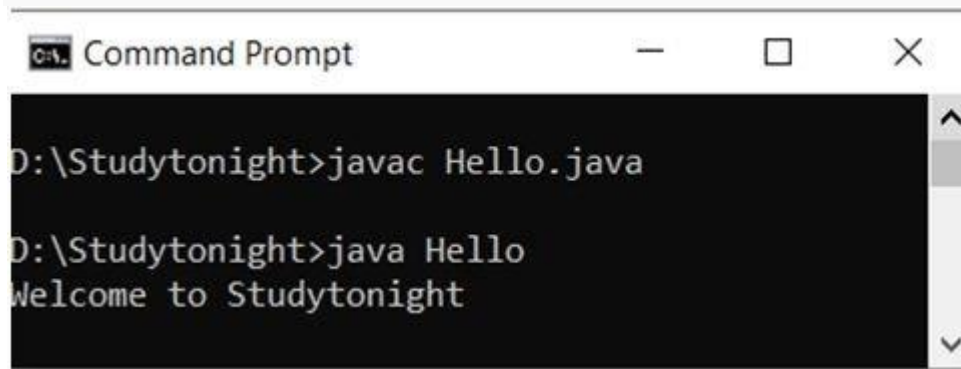
Changing the sequence of the modifiers and methods is accepted by Java.

Syntax: static public void main(String as[])

Example:

```
class Hello
{
    static public void main(String as[])
    {
        System.out.println ("Welcome to
Studytonight");
    }
}
```

Copy

A screenshot of a Windows Command Prompt window. The title bar reads "Command Prompt". The command prompt shows the following sequence of commands and output:
D:\Studytonight>javac Hello.java

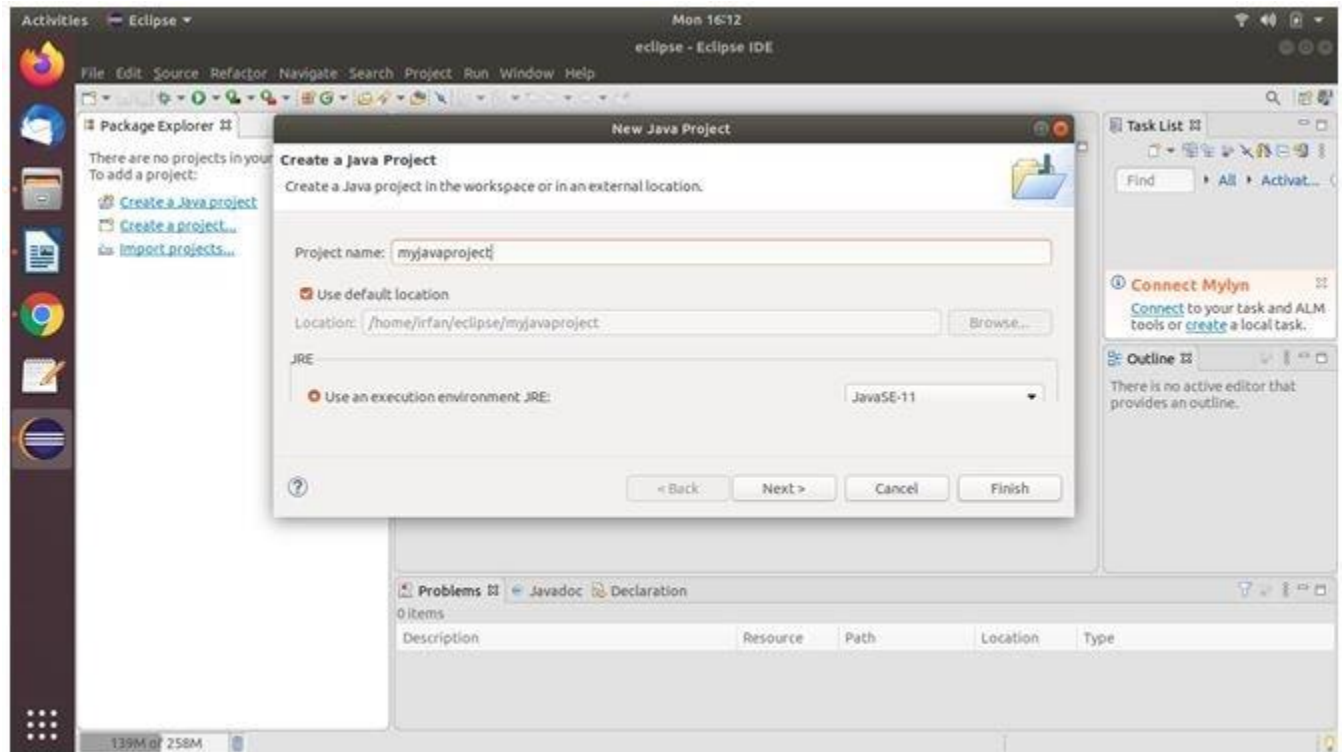
D:\Studytonight>java Hello
Welcome to Studytonight

Hello World Program using Eclipse

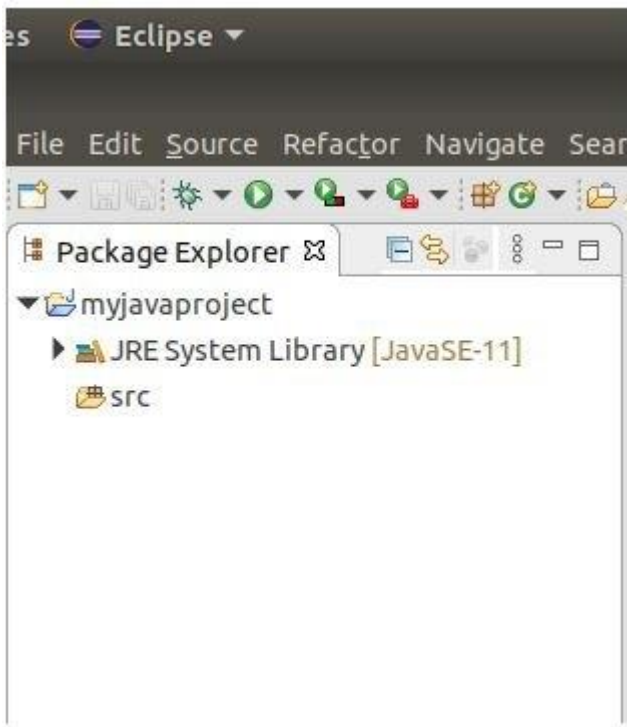
Eclipse is an IDE (Integrated Development Environment) which is used to develop applications. It is design and developed by Eclipse foundation, if you don't have eclipse download, then download it from its official site by following this download link [Download Eclipse from here](#) Here we will see how to create and run **hello world** program using eclipse IDE. It require following steps that consists of **creating project, class file, writing code, running code etc.**

Run Eclipse and Create Project

Open eclipse startup and then create new project. To create project click on **File** menu and select **Java project** option. It will open a window that ask for project name. Provide the project name and click on the finish button. See the below screenshot.

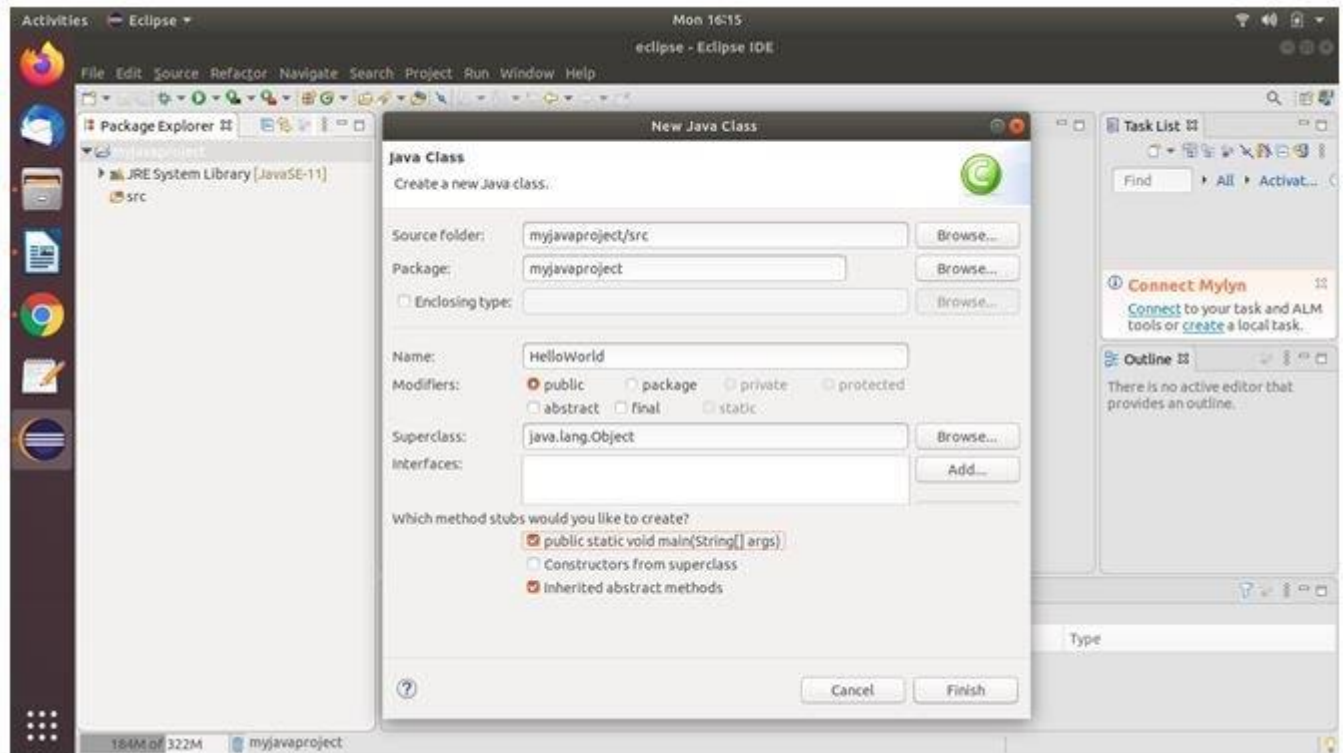


After creating project, we can see our new created project in the left side bar that looks like below.



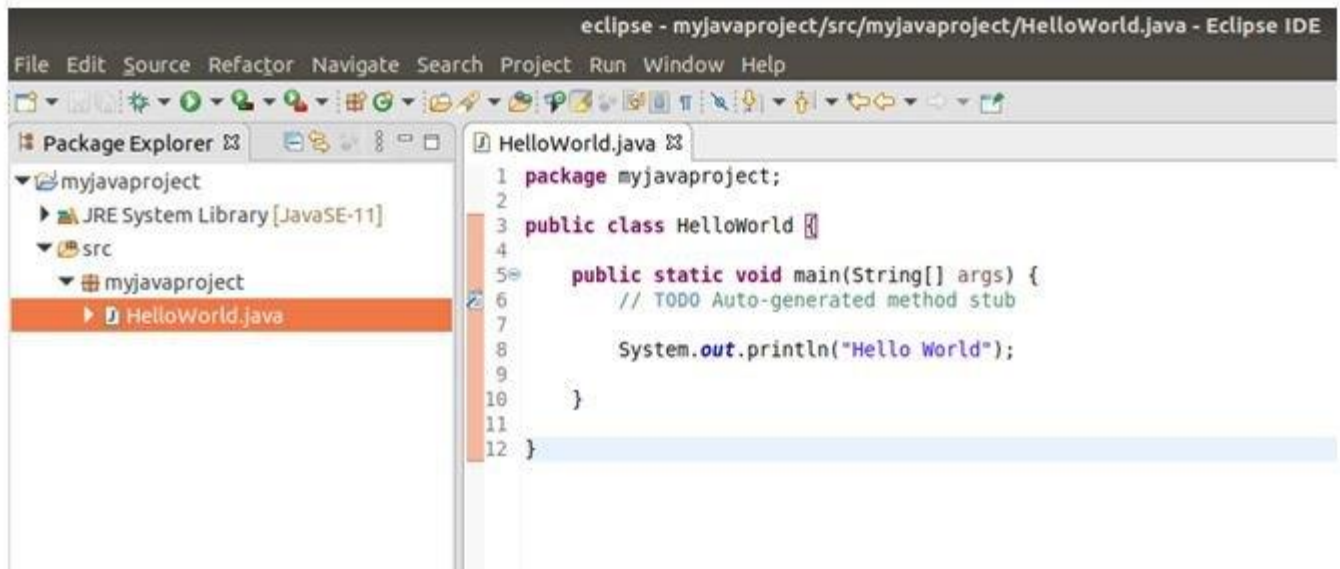
Create Java Class

Now create Java class file by **right click** on the **project** and **select class** file option. It will open a window to ask for class name, provide the class name and click on finish button.



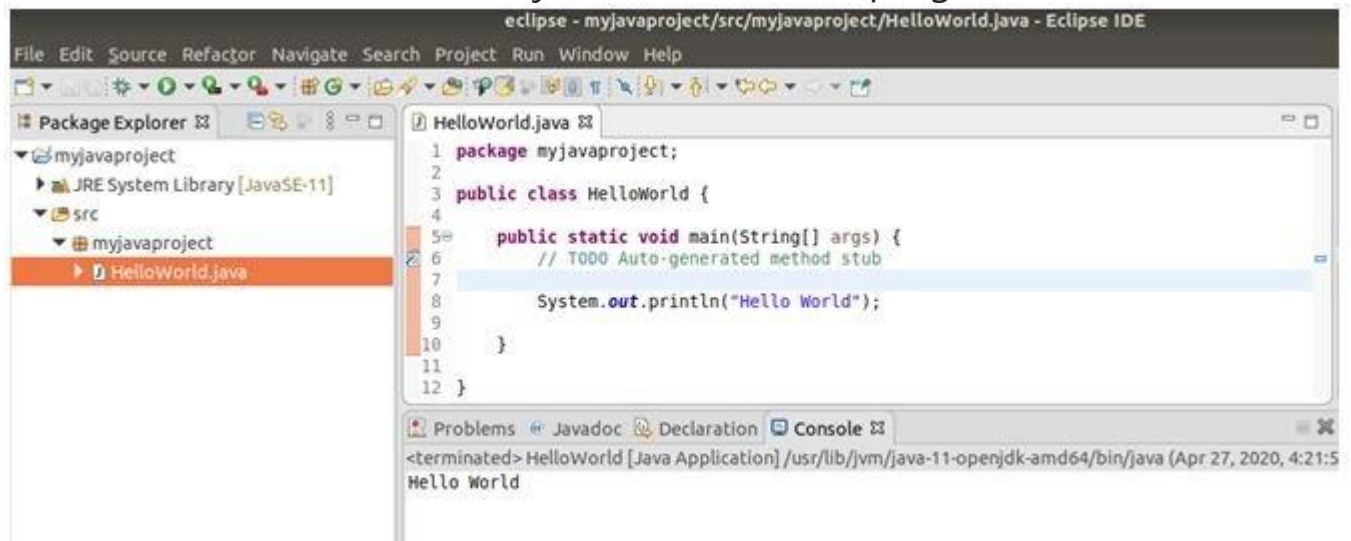
Write Hello World

The above created class file includes some line of codes including main method as well. Now we need to write just print statement to print Hello World message.



Run The Program

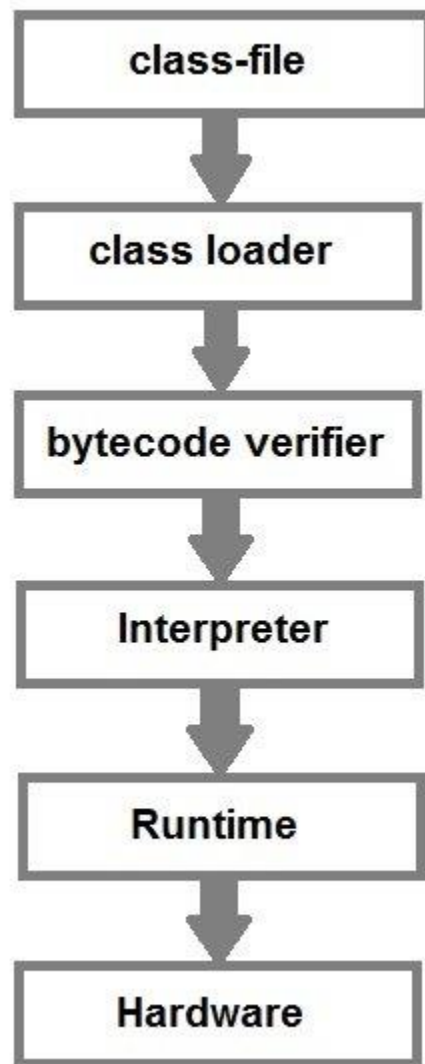
Now run the program by selecting **Run** menu from the menu bar or use **Ctrl+F11** button combination. After running, it will print Hello World to the console which is just bottom to the program window.



This is a simple program that we run here while using IDE we can create and build large scale of applications. If you are a beginner and not familiar to the Eclipse then don't worry it is very easy to operate just follow the above steps to create the program.

Now let us see What happens at Runtime

After writing your Java program, when you will try to compile it. Compiler will perform some compilation operation on your program. Once it is compiled successfully byte code(.class file) is generated by the compiler.



After compiling when you will try to run the byte code(.class file), the following steps are performed at runtime:-

1. Class loader loads the java class. It is subsystem of JVM Java Virtual machine.
2. Byte Code verifier checks the code fragments for illegal codes that can violate access right to the object.

3. Interpreter reads the byte code stream and then executes the instructions, step by step.

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

Java Operators

Operator is a symbol which tells to the compiler to perform some operation. Java provides a rich set of operators to deal with various types of operations. Sometimes we need to perform arithmetic operations then we use plus (+) operator for addition, multiply(*) for multiplication etc.

Operators are always essential part of any programming language.

Java operators can be divided into following categories:

- Arithmetic operators
- Relation operators

- Logical operators
- Bitwise operators
- Assignment operators
- Conditional operators
- Misc operators

Arithmetic operators

Arithmetic operators are used to perform arithmetic operations like: addition, subtraction etc and helpful to solve mathematical expressions. The below table contains Arithmetic operators.

Operator	Description
+	adds two operands
-	subtract second operands from first
*	multiply two operand
/	divide numerator by denominator
%	remainder of division
++	Increment operator increases integer value by one
--	Decrement operator decreases integer value by one

Example:

Lets create an example to understand arithmetic operators and their operations.

```
class Arithmetic_operators1{
public static void main(String as[])
{
    int a, b, c;
    a=10;
    b=2;
    c=a+b;
    System.out.println("Addtion: "+c);
    c=a-b;
    System.out.println("Substraction: "+c);
    c=a*b;
```

```

        System.out.println("Multiplication: "+c);
        c=a/b;
        System.out.println("Division: "+c);
        b=3;
        c=a%b;
        System.out.println("Remainder: "+c);
        a=++a;
        System.out.println("Increment Operator: "+a);
        a=--a;
        System.out.println("decrement Operator: "+a);
    }
}

```

Relation operators

Relational operators are used to test comparison between operands or values. It can be use to test whether two values are equal or not equal or less than or greater than etc.

The following table shows all relation operators supported by Java.

Operator	Description
==	Check if two operand are equal
!=	Check if two operand are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

Example:

In this example, we are using relational operators to test comparison like less than, greater than etc.

```

class Relational_operators1{
public static void main(String as[])
{
    int a, b;

```

```

a=40;
b=30;
System.out.println("a == b = " + (a == b) );
System.out.println("a != b = " + (a != b) );
System.out.println("a > b = " + (a > b) );
System.out.println("a < b = " + (a < b) );
System.out.println("b >= a = " + (b >= a) );
System.out.println("b <= a = " + (b <= a) );
}
}

```

Logical operators

Logical Operators are used to check conditional expression. For example, we can use logical operators in if statement to evaluate conditional based expression. We can use them into loop as well to evaluate a condition.

Java supports following 3 logical operator. Suppose we have two variables whose values are: **a=true** and **b=false**.

Operator	Description	Example
&&	Logical AND	(a && b) is false
	Logical OR	(a b) is true
!	Logical NOT	(!a) is false

```

class Logical_operators1{
public static void main(String as[])
{
    boolean a = true;
    boolean b = false;
    System.out.println("a && b = " + (a&&b));
    System.out.println("a || b = " + (a||b) );
    System.out.println("!(a && b) = " + !(a && b));
}
}

```

}

Bitwise operators

Bitwise operators are used to perform operations bit by bit.

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

The following table shows all bitwise operators supported by Java.

Operator	Description
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift

Now lets see truth table for bitwise &, | and ^

a	b	a & b	a b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The bitwise **shift operators** shifts the bit value. The **left operand specifies the value to be shifted** and the **right operand specifies the number of positions** that the bits in the value are to be shifted. Both operands have the same precedence.

Example:

Lets create an example that shows working of bitwise operators.

a = 0001000

b = 2

a << b = 0100000

a >> b = 0000010

```
class Bitwise_operators1{
public static void main(String as[])
{
    int a = 50;
    int b = 25;
    int c = 0;

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

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

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

    c = ~a;
    System.out.println("~a = " + c );

    c = a << 2;
    System.out.println("a << 2 = " + c );

    c = a >> 2;
    System.out.println("a >>2 = " + c );

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

Assignment Operators

Assignment operators are used to assign a value to a variable. It can also be used combine with arithmetic operators to perform arithmetic operations and then assign the result to the variable. Assignment operator supported by Java are as follows:

Operator	Description	Example
=	assigns values from right side operands to left side operand	a = b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
=	multiply left operand with the right operand and assign the result to left operand	a=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

Example:

Lets create an example to understand use of assignment operators. All assignment operators have right to left associativity.

```
class Assignment_operators1{  
public static void main(String as[])  
{  
    int a = 30;  
    int b = 10;  
    int c = 0;
```

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

```
c += a ;  
System.out.println("c += a = " + c);
```

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

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

```
a = 20;  
c = 25;  
c /= a ;  
System.out.println("c /= a = " + c);
```

```
a = 20;  
c = 25;  
c %= a ;  
System.out.println("c %= a = " + c);
```

```
c <<= 2 ;  
System.out.println("c <<= 2 = " + c);
```

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

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

```
c &= a ;
```

```
System.out.println("c &= a = " + c );
```

```
    c ^= a ;
```

```
System.out.println("c ^= a = " + c );
```

```
    c |= a ;
```

```
System.out.println("c |= a = " + c );
```

```
    }  
}
```

Misc. operator

There are few other operator supported by java language.

Conditional operator

It is also known as ternary operator because it works with **three operands. It is short alternate of if-else statement.** It can be used to evaluate Boolean expression and **return either true or false** value

expr1 ? expr2 : expr3

Example:

In ternary operator, if **expr1** is true then expression evaluates after **question mark (?)** else evaluates **after colon (:)**. See the below example.

```
class Conditional_operators1{  
    public static void main(String as[])  
    {  
        int a, b;  
        a = 20;  
        b = (a == 1) ? 30: 40;  
        System.out.println( "Value of b is : " + b );  
    }  
}
```

```
        b = (a == 20) ? 30: 40;
System.out.println( "Value of b is : " + b );
    }
}
```

instanceOf operator

It is a java **keyword** and used to test whether the given **reference belongs to provided type** or not. Type can be a class or interface. **It returns either true or false.**

Example:

Here, we created a string reference variable that stores "studytonight". Since it stores string value so we test it using instanceof operator to check whether it belongs to string class or not. See the below example.

```
class instanceof_operators1{
    public static void main(String as[])
    {
        String a = "ducatindia";
        boolean b = a instanceof String;
        System.out.println( b );
    }
}
```

TYPE CASTING IN JAVA

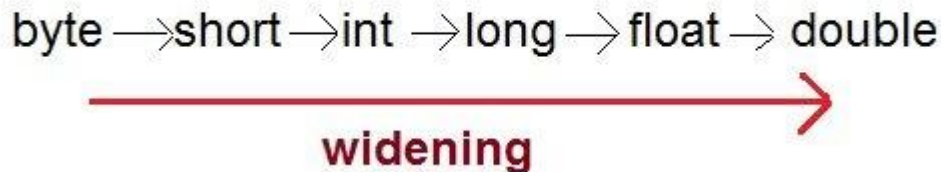
Casting is a process of changing one type value to another type. In Java, we can cast one type of value to another type. It is known as type casting.

Example :

```
int x = 10;
byte y = (byte)x;
```

In Java, type casting is classified into two types,

- Widening Casting(Implicit)



- Narrowing Casting(Explicitly done)



Widening or Automatic type conversion

Automatic Type casting take place when,

- the two types are compatible
- the target type is larger than the source type

```
public class Test
{
    public static void main(String[] args)
    {
        int i = 100;
        long l = i; //no explicit type casting required
        float f = l; //no explicit type casting required
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

Narrowing or Explicit type conversion

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting. If we don't perform casting then compiler reports compile time error.

```

public class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;
        long l = (long)d; //explicit type casting required
        int i = (int)l; //explicit type casting required

        System.out.println("Double value "+d);
        System.out.println("Long value "+l);
        System.out.println("Int value "+i);
    }
}

```

Example of Explicit Conversion

Here, we have one more example of explicit casting, double type is stored into long, long is stored into int etc.

```

class CastingDemo1
{
    public static void main(String[] args)
    {
        double d = 120.04;
        long l = (long)d;
        int i = (int)l;
        System.out.println("Double value "+d);
        System.out.println("Long value "+l);
        System.out.println("Int value "+i);
    }
}

```

Example for Conversion of int and double into a byte

Here, we are converting int and double type to byte type by using explicit type casting.

```

class Demo2
{
    public static void main(String args[])
    {
        byte b;
        int i = 355;
        double d = 423.150;
        b = (byte) i;
        System.out.println("Conversion of int to byte: i = " + i + " b = " + b);
    }
}

```

```

        System.out.println("*****");
        b = (byte) d;
        System.out.println("Conversion of double to byte: d = " + d + " b= " + b);
    }
}

```

JAVA IF ELSE STATEMENT

In Java, if statement is used for testing the conditions. The condition matches the statement it returns true else it returns false. There are four types of If statement they are:

For example, if we want to create a program to **test positive integers** then we have to test the integer whether it is greater than zero or not.

In this scenario, if statement is helpful.

There are four types of if statement in Java:

- i. if statement
- ii. if-else statement
- iii. if-else-if ladder
- iv. nested if statement

if Statement

The if statement is a single conditional based statement that executes only if the provided condition is true.

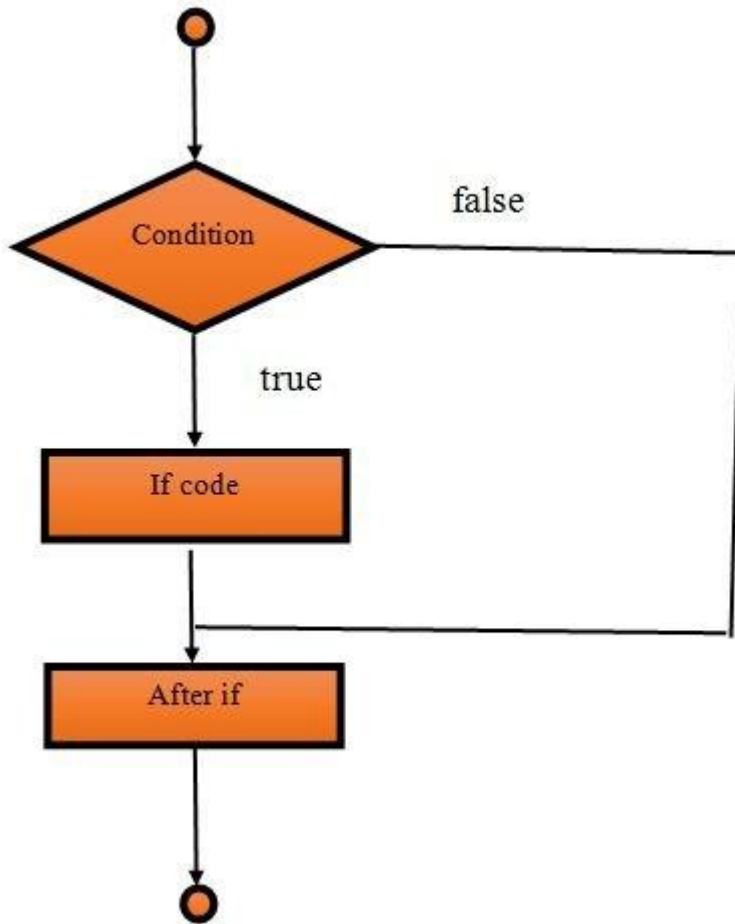
```

if(condition)
{
    //code
}

```

We can understand flow of if statement by using the below diagram. It shows that **code written inside the if will execute only if the condition is true.**

Data-flow-diagram of If Block



Example:

In this example, we are testing students marks. If the marks are greater than 65 then student will get first division.

```
public class IfDemo1 {  
    public static void main(String[] args)  
    {  
        int marks=70;  
        if(marks > 65)  
        {  
            System.out.print("First division");  
        }  
    }  
}
```

if-else Statement

The if-else statement is used for testing condition. If the condition is true, if block executes otherwise else block executes.

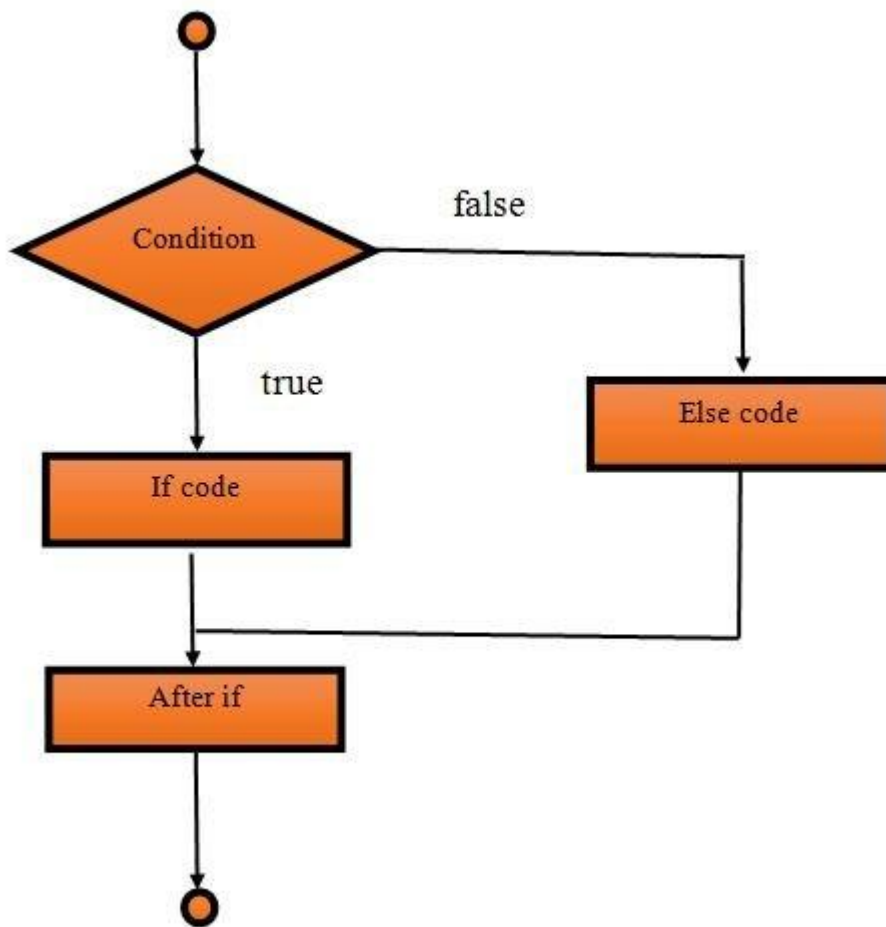
It is useful in the scenario when we want to perform some operation based on the **false** result.

The else block execute only when condition is **false**.

Syntax:

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

In this block diagram, we can see that when condition is true, if block executes otherwise else block executes.



if else Example:

In this example, we are testing student marks, if marks is greater than 65 then if block executes otherwise else block executes.

```
public class IfElseDemo1 {
    public static void main(String[] args)
    {
        int marks=50;
        if(marks > 65)
        {
            System.out.print("First division");
        }
        else
        {
            System.out.print("Second division");
        }
    }
}
```

if-else-if ladder Statement

In Java, the if-else-if ladder statement is used for testing conditions. It is used for testing one condition from multiple statements.

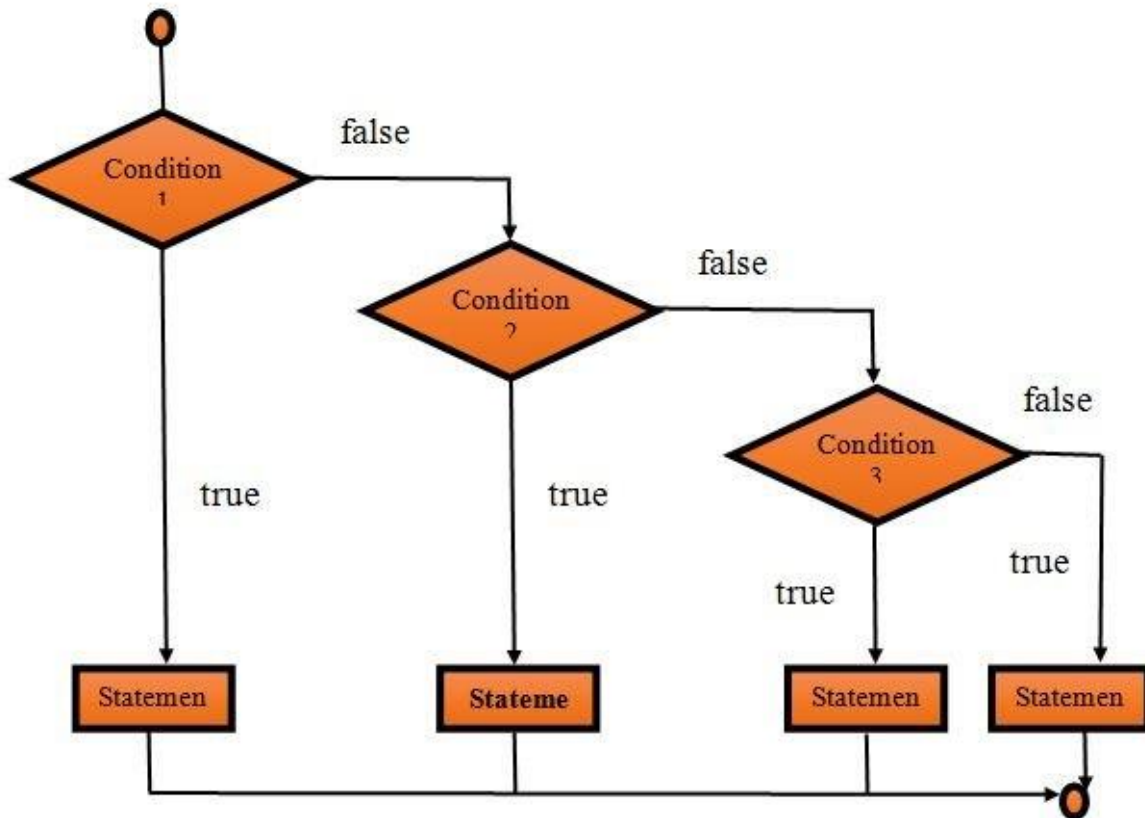
When we have multiple conditions to execute then it is recommend to use if-else-if ladder.

Syntax:

```
if(condition1)
{
    //code for if condition1 is true
}
else if(condition2)
{
    //code for if condition2 is true
}
else if(condition3)
{
    //code for if condition3 is true
}
...
else
{
    //code for all the false conditions
}
```

It contains multiple conditions and execute if any condition is true otherwise executes else block.

Data-flow-diagram of If Else If Block



Example:

Here, we are testing student marks and displaying result based on the obtained marks. If marks are greater than 50 student gets his grades.

```
public class IfElseIfDemo1 {  
    public static void main(String[] args) {  
        int marks=75;  
        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!");
    }
}
}
```

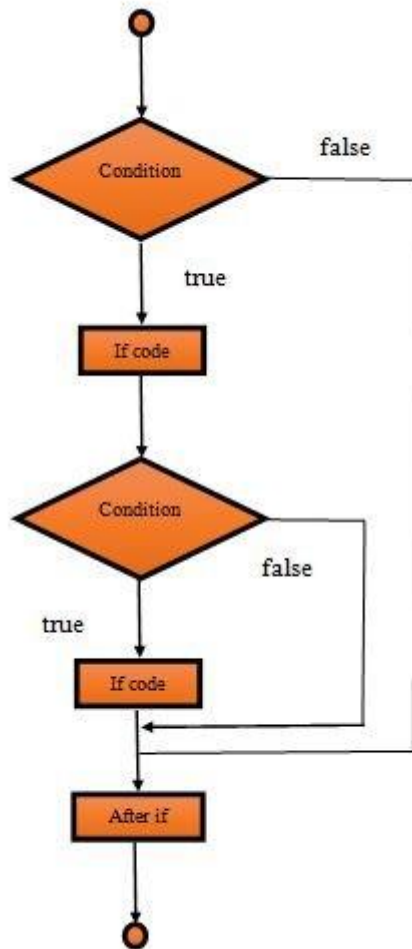
Nested if statement

In Java, the Nested if statement is a if inside another if. In this, one if block is created inside another if block when the outer block is true then only the inner block is executed.

Syntax:

```
if(condition)
{
    //statement
    if(condition)
    {
        //statement
    }
}
```

Data-flow-diagram of Nested If Block



Example:

```
public class NestedIfDemo1 {  
    public static void main(String[] args)  
    {  
        int age=25;  
        int weight=70;  
        if(age>=18)  
        {  
            if(weight>50)  
            {  
                System.out.println("You are eligible");  
            }  
        }  
    }  
}
```

JAVA SWITCH STATEMENT

In Java, the switch statement is used for executing one statement from multiple conditions. it is similar to an if-else-if ladder.

Switch statement consists of conditional based cases and a default case.

In a switch statement, the expression can be of **byte, short, char and int** type.

From **JDK-7, enum, String** can also be used in switch cases.

Following are some of the rules while using the switch statement:

1. There can be one or N numbers of cases.
2. The values in the case must be unique.
3. Each statement of the case can have a break statement. It is optional.

Syntax:

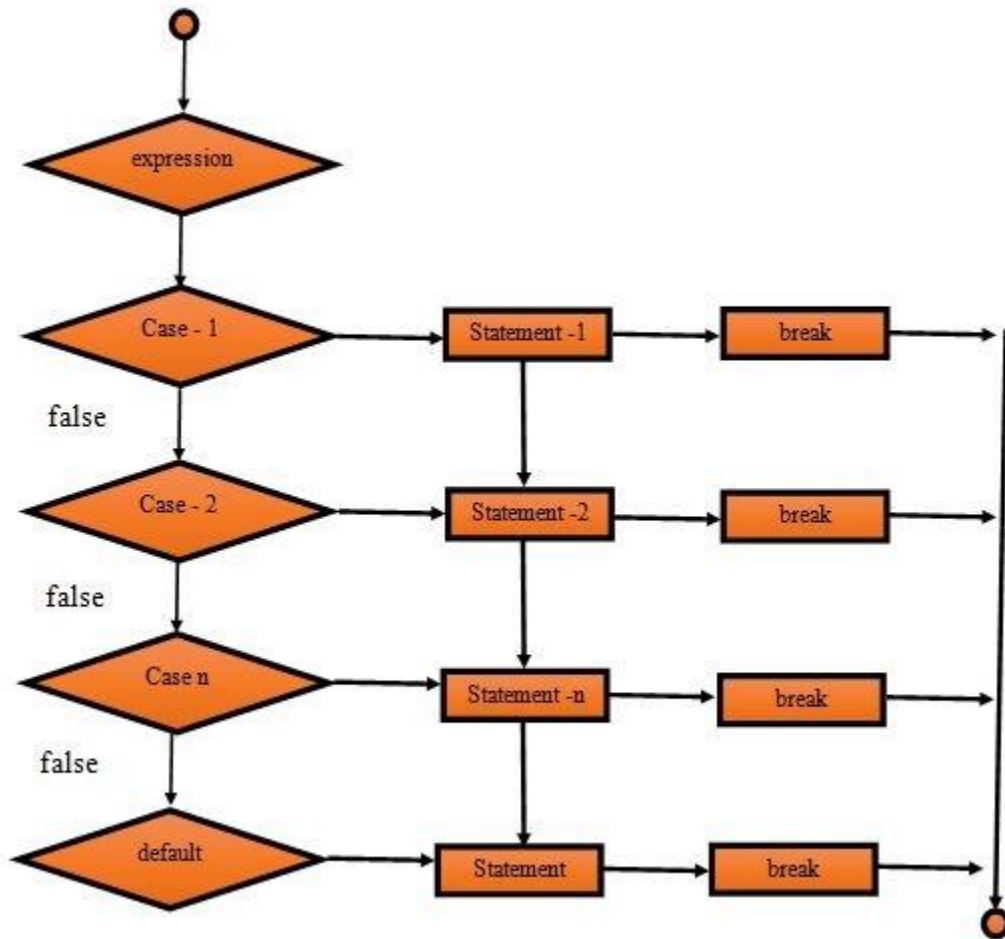
Following is the syntax to declare the switch case in Java.

```
switch(expression)
{
case value1:
                //code for execution;
                break; //optional

case value2:
    // code for execution
    break; //optional
.....
.....
.....
.....
Case value n:
// code for execution
break; //optional

default:
    code for execution when none of the case is true;
}
```

Data Flow Diagram Of switch Block



Example: Using integer value

In this example, we are using **int type** value to match cases. This example returns day based on the numeric value.

```
public class SwitchDemo1 {
    public static void main(String[] args)
    {
        int day = 3;
        String dayName;
        switch (day) {
            case 1:
                dayName = "Today is Monday";
                break;
            case 2:
                dayName = "Today is Tuesday";
                break;
            case 3:
                dayName = "Today is Wednesday";
                break;
        }
    }
}
```



```

        case 4:
dayName = "Today is Thursday";
        break;
        case 5:
dayName = "Today is Friday";
        break;
        case 6:
dayName = "Today is Saturday";
        break;
        case 7:
dayName = "Today is Sunday";
        break;
        default:
dayName = "Invalid day";
        break;
    }
System.out.println(dayName);
}
}

```

Example using Enum in Switch statement

As we have said, Java allows to use **enum** in switch cases. So we are creating an enum of vowel alphabets and using its elements in switch case.

```

public class SwitchDemo2{
    public enum vowel{a, e, i, o, u}
    public static void main(String args[])
    {
vowel[] character= vowel.values();
        for (vowel Now : character)
        {
            switch (Now)
            {
                case a:
System.out.println("'a' is a Vowel");
                break;
                case e:
System.out.println("'e' is a Vowel");
                break;
                case i:
System.out.println("'i' is a Vowel");
                break;
                case o:
System.out.println("'o' is a Vowel");
                break;

```

```

        case u:
System.out.println("'u' is a Vowel");
        break;
        default:
System.out.println("It is a consonant");
    }
}
}
}

```

Example: String in switch case

Since Java has allowed to use string values in switch cases, so we are using string to create a string based switch case example.

```

public static void main(String[] args) {
    String name = "Mango";
    switch(name){
    case "Mango":
        System.out.println("It is a fruit");
        break;
    case "Tomato":
        System.out.println("It is a vegetable");
        break;
    case "Coke":
        System.out.println("It is cold drink");
    }
}
}

```

Example: without break switch case

Break statement is used to **break the current execution** of the program. In switch case, break is used to **terminate the switch case** execution and transfer control to the outside of switch case. Use of **break is optional in the switch case**. So lets see what happens if we don't use the break.

```

public class Demo{

    public static void main(String[] args) {
        String name = "Mango";
        switch(name){
        case "Mango":
            System.out.println("It is a fruit");
        case "Tomato":

```

```

        System.out.println("It is a vegitable");
    case "Coke":
        System.out.println("It is cold drink");
    }
}
}

```

JAVA LOOPS

Loop is an important concept of a programming that allows to iterate over the sequence of statements.

Loop is designed to execute particular code block till the specified condition is true or all the elements of a collection(array, list etc) are completely traversed.

The most common use of loop is to perform repetitive tasks. For example if we want to print table of a number then we need to write print statement 10 times. However, we can do the same with a single print statement by using loop.

Loop is designed to execute its block till the specified condition is true. Java provides mainly three loop based on the loop structure.

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

We will explain each loop individually in details in the below.

For Loop

The for loop is used for executing a part of the program **repeatedly**.

When the number of execution is fixed then it is suggested to use for loop. For loop can be categories into two type.

1. for loop
2. for-each loop

For Loop Syntax:

Following is the syntax for declaring for loop in Java.

```

for(initialization;condition;increment/decrement)
{
    //statement
}

```

For loop Parameters:

To create a for loop, we need to set the following parameters.

1) Initialization

It is the initial part, where we set initial value for the loop. It is executed only once at the starting of loop. It is optional, if we don't want to set initial value.

2) Condition

It is used to test a condition each time while executing. The execution continues until the condition is false. It is optional and if we don't specify, loop will be infinite.

3) Statement

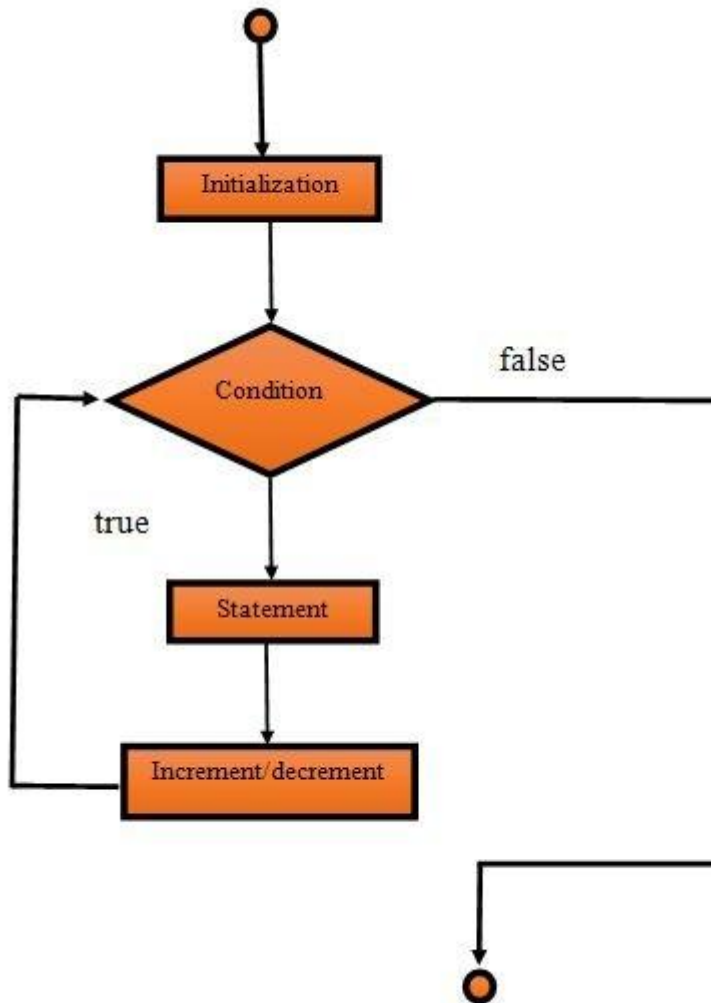
It is loop body and executed every time until the condition is false.

4) Increment/Decrement

It is used for set increment or decrement value for the loop.

Data Flow Diagram of for loop

This flow diagram shows flow of the loop. Here we can understand flow of the loop.



For loop Example

In this example, initial value of loop is set to 1 and incrementing it by 1 till the condition is true and executes 10 times.

```
public class ForDemo1
{
    public static void main(String[] args)
    {
        int n, i;
        n=2;
        for(i=1;i<=10;i++)
        {
            System.out.println(n+"*"+i+"="+n*i);
        }
    }
}
```

```
}
```

Example for Nested for loop

Loop can be nested, loop created **inside another loop** is called nested loop. Sometimes based on the requirement, we have to create nested loop.

Generally, nested loops are used to **iterate tabular data**.

```
public class ForDemo2
{
    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();
        }
    }
}
```

for-each Loop

In Java, for each loop is used for traversing array or collection elements. In this loop, there is no need for increment or decrement operator.

For-each loop syntax

Following is the syntax to declare for-each loop in the Java.

```
for(Type var:array)
{
    //code for execution
}
```

Example:

In this example, we are traversing array elements using the for-each loop. For-each **loop terminates automatically when no element is left in the array object**.

```
public class ForEachDemo1
{
    public static void main(String[] args)
    {
        inta[]={20,21,22,23,24};
        for(int i:a)
        {
            System.out.println(i);
        }
    }
}
```

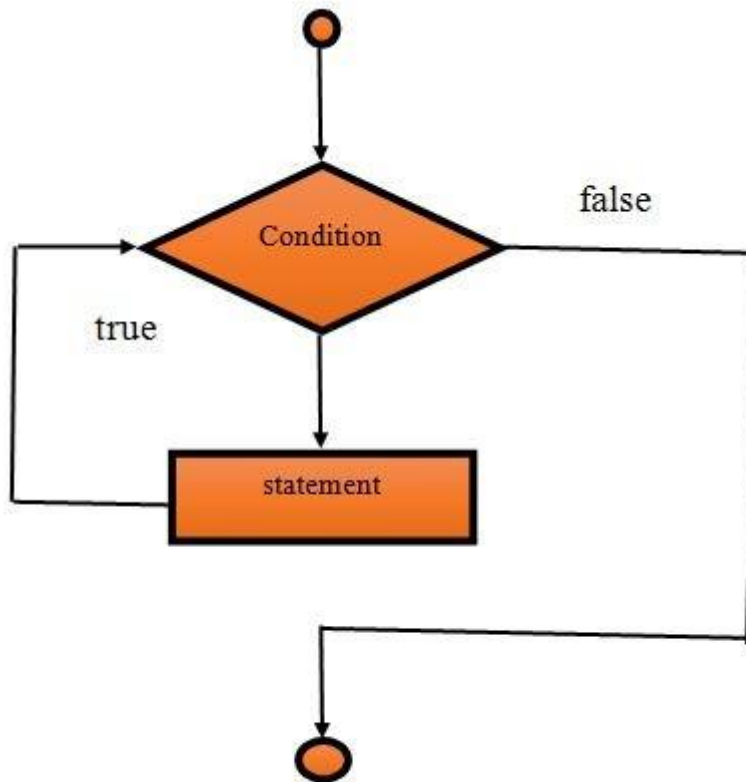
While Loop

Like for loop, while loop is also used to execute code repeatedly. a control statement. It is used for iterating a part of the program several times. When the number of iteration is not fixed then while loop is used.

Syntax:

```
while(condition)
{
    //code for execution
}
```

Data-flow-diagram of While Block



Example:

In this example, we are using while loop to print 1 to 10 values. In first step, we set conditional variable then test the condition and if condition is true execute the loop body and increment the variable by 1.

```
public class WhileDemo1
{
    public static void main(String[] args)
    {
        inti=1;
        while(i<=10)
        {
            System.out.println(i);
            i++;
        }
    }
}
```



```
}
```

Example for infinite while loop

A while loop which conditional expression **always returns true** is called infinite while loop. We can also create infinite loop by passing **true** literal in the loop.

Be careful, when creating infinite loop because it can issue memory overflow problem.

```
public class WhileDemo2
{
    public static void main(String[] args)
    {
        while(true)
        {
            System.out.println("infinitive while loop");
        }
    }
}
```

do-while loop

In Java, the do-while loop is used to execute statements again and again. This loop **executes at least once** because the loop is executed before the condition is checked. It means loop **condition evaluates after executing** of loop body.

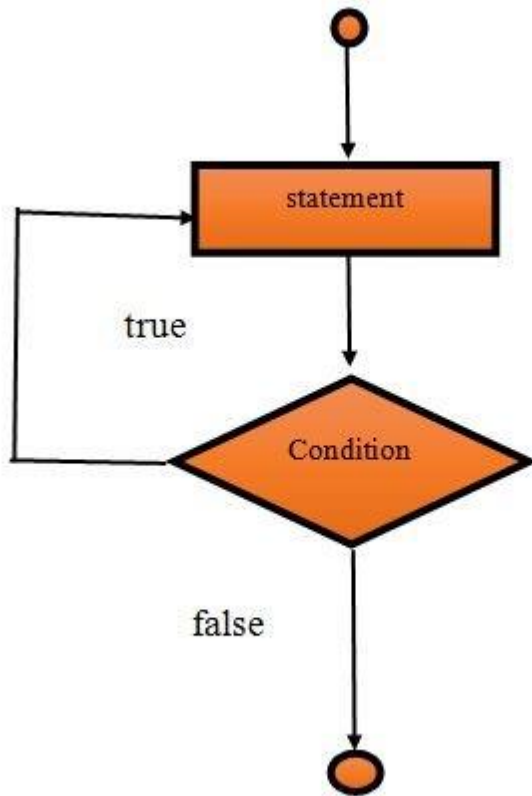
The main **difference between while and do-while loop** is, in do while loop condition evaluates after executing the loop.

Syntax:

Following is the syntax to declare do-while loop in Java.

```
do
{
    //code for execution
}
while(condition);
```

Data Flow Diagram of do-while Block



Example:

In this example, we are printing values from 1 to 10 by using the do while loop.

```
public class DoWhileDemo1
{
    public static void main(String[] args)
    {
        inti=1;
        do
        {
            System.out.println(i);
            i++;
        }while(i<=10);
    }
}
```

Example for infinite do-while loop

Like infinite while loop, we can create infinite do while loop as well. To create an infinite do while loop just **pass the condition that always remains true**.

```
public class DoWhileDemo2
{
    public static void main(String[] args)
    {
        do
        {
            System.out.println("infinite do while loop");
        }while(true);
    }
}
```

JAVA **BREAK** AND **CONTINUE** STATEMENTS

Java **break** and **continue** statements are used to manage program flow. We can use them in a loop to control loop iterations. These statements let us to control loop and switch statements by enabling us to either break out of the loop or jump to the next iteration by skipping the current loop iteration.

In this tutorial, we will discuss each in details with examples.

Break Statement

In Java, break is a statement that is used to **break current execution** flow of the program.

We can use break statement inside loop, switch case etc.

If break is used **inside loop** then it will terminate the loop.

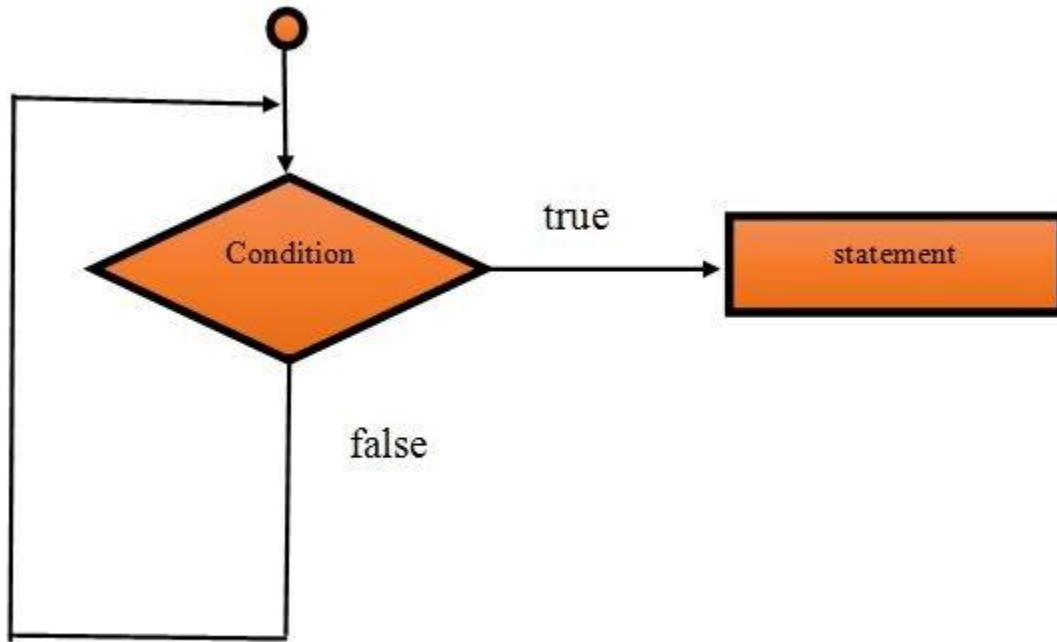
If break is used **inside the innermost loop** then break will terminate the innermost loop only and execution will start from the outer loop.

If break is used in switch case then it will terminate the execution after the matched case. Use of break, we have covered in our switch case topic.

Syntax:

jump-statement;
break;

Data Flow Diagram of break statement



Example:

In this example, we are using break inside the loop, the loop will terminate when the value is 8.

```
public class BreakDemo1 {  
    public static void main(String[] args) {  
  
        for(int i=1; i<=10; i++){  
            if(i==8){  
                break;  
            }  
            System.out.println(i);  
        }  
    }  
}
```

Example using break in do while loop

Loop can be any one whether it is for or while, break statement will do the same. Here, we are using break inside the do while loop.

```
public class BreakDoWhileDemo1
{
    public static void main(String[] args)
    {
        inti=1;
        do
        {
            if(i==15)
            {
                i++;
                break;
            }
            System.out.println(i);
            i++;
        }while(i<=20);
    }
}
```

Example: Break in innermost loop

In this example, we are using **break inside the innermost loop**. But the loop breaks each time when j is equal to 2 and **control goes to outer loop** that starts from the next iteration.

```
public class Demo{
    public static void main(String[] args) {
        for(int i=1;i<=2;i++){
            for (int j = 0; j <=3; j++) {
                if(j==2)
                    break;
                System.out.println(j);
            }
        }
    }
}
```

continue Statement

In Java, the continue statement is used to **skip the current iteration of the loop**. It **jumps to the next iteration** of the loop immediately.

We can use continue statement with **for loop**, **while loop** and **do-while loop** as well.

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

Example:

In this example, we are using continue statement inside the for loop. See, it does not print 5 to the console because at fifth iteration continue statement skips the iteration that's why print statement does not execute.

```
public class ContinueDemo1  
{  
    public static void main(String[] args)  
    {  
        for(int i=1;i<=10;i++)  
        {  
            if(i==5)  
            {  
                continue;  
            }  
            System.out.println(i);  
        }  
    }  
}
```

Example:

We can use **label along with continue statement** to **set flow control**. By using label, we can **transfer control at specified location**. In this example, we are transferring control to outer loop by using label.

```
public class ContinueDemo2 {  
    public static void main(String[] args) {  
        xy:  
        for(int i=1;i<=5;i++){  
            pq:  
            for(int j=1;j<=5;j++){  
                if(i==2&&j==2){  
                    continue xy;  
                }  
            }  
        }  
    }  
}
```

```

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

```

Example: Continue in While loop

continue statement can be used with while loop to manage the flow control of the program. As we already know continue statement is used to skip the current iteration of the loop. Here too, it will skip the execution if the value of variable is 5.

```

public class Demo{

    public static void main(String[] args) {
        int i=1;

        while (i < 10) {
            if (i == 5) {
                i++;
                continue;
            }
            System.out.println(i);
            i++;
        }
    }
}

```

DIFFERENT WAYS TO CREATE OBJECTS IN JAVA

Java is an object-oriented language, everything revolve around the object. An object represents runtime entity of a class and is essential to call variables and methods of the class.

To create an object Java provides various ways that we are going to discuss in this topic.

- New keyword
- New instance
- Clone method
- Deserialization
- NewInstance() method

1) new Keyword

In Java, creating objects using new keyword is very popular and common. Using this method user or system defined default constructor is called that initialize instance variables. And new keyword creates a memory area in heap to store created object.

Example:

In this example, we are creating an object by using new keyword.

```
public class NewKeyword
{
    String s = "ducatIndia";
    public static void main(String as[])
    {
        NewKeyword a = new NewKeyword();
        System.out.println(a.s);
    }
}
```

2) New Instance

In this case, we use a static method `forName()` of `Class` class. This method loads the class and returns an object of type `Class`. That further we cast in our required type to get required type of object. This is what we do in this case.

Example:

You can see the example and understand that we loaded our class `NewInstance` by using `Class.forName()` method that returns the type `Class` object.

```
public class NewInstance
{
    String a = "ducatindia";
    public static void main(String[] args)
    {
        try
        {
            Class b = Class.forName("NewInstance");
            NewInstance c = (NewInstance) b.newInstance();
            System.out.println(c.a);
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
        }
        catch (InstantiationException e)
        {
            e.printStackTrace();
        }
    }
}
```



```

    }
    catch (IllegalAccessException e)
    {
e.printStackTrace();
    }
}
}

```

3) Clone() method

In Java, clone() is called on an object. When a clone() method is called JVM creates a new object and then copy all the content of the old object into it. When an object is created using the clone() method, a constructor is not invoked. To use the clone() method in a program the class implements the cloneable and then define the clone() method.

Example:

In this example, we are creating an object using clone() method.

```

public class CloneEg implements Cloneable
{
    @Override
    protected Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
    String s = "ducatindia";

    public static void main(String[] args)
    {
CloneEg a= new CloneEg();
        try
        {
CloneEg b = (CloneEg) a.clone();
System.out.println(b.s);
        }
        catch (CloneNotSupportedException e)
        {
e.printStackTrace();
        }
    }
}

```

4) deserialization

In Java, when an object is serialized and then deserialized, JVM create another separate object. When deserialization is performed JVM does not use any constructor for creating an object.

Example:

Lets see an example of creating object by deserialization concept.

```
import java.io.*;

class DeserializationEg implements Serializable
{
    private String a;
    DeserializationEg(String name)
    {
        this.a = a;
    }

    public static void main(String[] args)
    {
        try
        {
            DeserializationEg b = new DeserializationEg("studytonight");
            FileOutputStream c = new FileOutputStream("CoreJava.txt");
            ObjectOutputStream d = new ObjectOutputStream(c);
            d.writeObject(b);
            d.close();
            d.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

newInstance() method of Constructor class

In Java, Under java.lang.reflect package Constructor class is located. We can use it to create object. The Constructor class provides a method newInstance() that can be used for creating an object. This method is also called a parameterized constructor.

Example:

Lets create an example to create an object using newInstance() method of Constructor class.

```
import java.lang.reflect.*;

public class ReflectionEg
{
    private String s;
    ReflectionEg()
    {
    }
}
```

```

public void setName(String s)
{
    this.s = s;
}
public static void main(String[] args)
{
    try
    {
        Constructor constructor = ReflectionEg.class.getDeclaredConstructor();
        ReflectionEg r = constructor.newInstance();
        r.setName("studytonight");
        System.out.println(r.s);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
}

```

Java Command line argument

The command line argument is the argument that passed to a program during runtime. It is the way to pass argument to the main method in Java. These arguments store into the String type args parameter which is main method parameter.

To access these arguments, you can simply traverse the args parameter in the loop or use direct index value because args is an array of type String.

For example, if we run a HelloWorld class that contains main method and we provide argument to it during runtime, then the syntax would be like.

```
java HelloWorld arg1 arg2 ...
```

We can pass any number of arguments because argument type is an array. Lets see an example.

Example

In this example, we created a class HelloWorld and during running the program we are providing command-line argument.

```

class cmd
{
    public static void main(String[] args)
    {
        for(int i=0;i< args.length;i++)
        {
            System.out.println(args[i]);
        }
    }
}

```

```
}  
}
```

To terminate the program in between based on some condition or program logic, Java provides `exit()` that can be used to terminate the program at any point. Here we are discussing about the `exit()` method with the example.

Java `System.exit()` Method

In Java, `exit()` method is in the `java.lang.System` class. This method is used to take an exit or terminating from a running program. It can take either zero or non-zero value. `exit(0)` is used for successful termination and `exit(1)` or `exit(-1)` is used for unsuccessful termination. The `exit()` method does not return any value.

Example:

In this program, we are terminating the program based on a condition and using `exit()` method.

```
import java.util.*;  
import java.lang.*;  
  
class ExitDemo1  
{  
    public static void main(String[] args)  
    {  
        intx[] = {5, 10, 15, 20, 25, 30, 35, 40, 45, 50};  
  
        for (inti = 0; i<x.length; i++)  
        {  
            if (x[i] >= 40)  
            {  
                System.out.println("Program is Terminated...");  
                System.exit(0);  
            }  
            else  
                System.out.println("x["+i+"] = " + x[i]);  
        }  
    }  
}
```

What is OOPS

OOPS is a programming approach which provides solution to real life problems with the help of algorithms based on real world. It uses real world approach to solve a problem. So object

oriented technique offers better and easy way to write program then procedural programming languages such as C, ALGOL, PASCAL etc.

Java is an object oriented language which supports object oriented concepts like: class and object. In OOPS data is treated important and encapsulated within the class, object then use to access that data during runtime.

OOPS provides advantages over the other programming paradigm and include following features.

Main Features of OOPS

Inheritance

Polymorphism

Encapsulation

Abstraction

As an object oriented language Java supports all the features given above. We will discuss all these features in detail later.

Java Class

In Java everything is encapsulated under classes. Class is the core of Java language. It can be defined as a template that describe the behaviors and states of a particular entity.

A class defines new data type. Once defined this new type can be used to create object of that type.

Object is an instance of class. You may also call it as physical existence of a logical template class.

In Java, to declare a class class keyword is used. A class contain both data and methods that operate on that data. The data or variables defined within a class are called instance variables and the code that operates on this data is known as methods.

Thus, the instance variables and methods are known as class members.

Rules for Java Class

A class can have only public or default(no modifier) access specifier.

It can be either abstract, final or concrete (normal class).

It must have the class keyword, and class must be followed by a legal identifier.

It may optionally extend only one parent class. By default, it extends Object class.

The variables and methods are declared within a set of curly braces.

A Java class can contains fields, methods, constructors, and blocks. Lets see a general structure of a class.

Java class Syntax

```
class class_name {  
    field;  
    method;  
}
```

A simple class example

Suppose, Student is a class and student's name, roll number, age are its fields and info() is a method. Then class will look like below.

```
class Student.  
{  
    String name;  
    int rollno;  
    int age;  
    void info(){  
        // some code  
    }  
}
```

This is how a class look structurally. It is a blueprint for an object. We can call its fields and methods by using the object.

The fields declared inside the class are known as instance variables. It gets memory when an object is created at runtime.

Methods in the class are similar to the functions that are used to perform operations and represent behavior of an object.

After learning about the class, now lets understand what is an object.

Java Object

Object is an instance of a class while class is a blueprint of an object. An object represents the class and consists of properties and behavior.

Properties refer to the fields declared with in class and behavior represents to the methods available in the class.

In real world, we can understand object as a cell phone that has its properties like: name, cost, color etc and behavior like calling, chatting etc.

So we can say that object is a real world entity. Some real world objects are: ball, fan, car etc.

There is a syntax to create an object in the Java.

Java Object Syntax

```
className variable_name = new className();
```

Here, className is the name of class that can be anything like: Student that we declared in the above example.

variable_name is name of reference variable that is used to hold the reference of created object.

The new is a keyword which is used to allocate memory for the object.

Lets see an example to create an object of class Student that we created in the above class section.

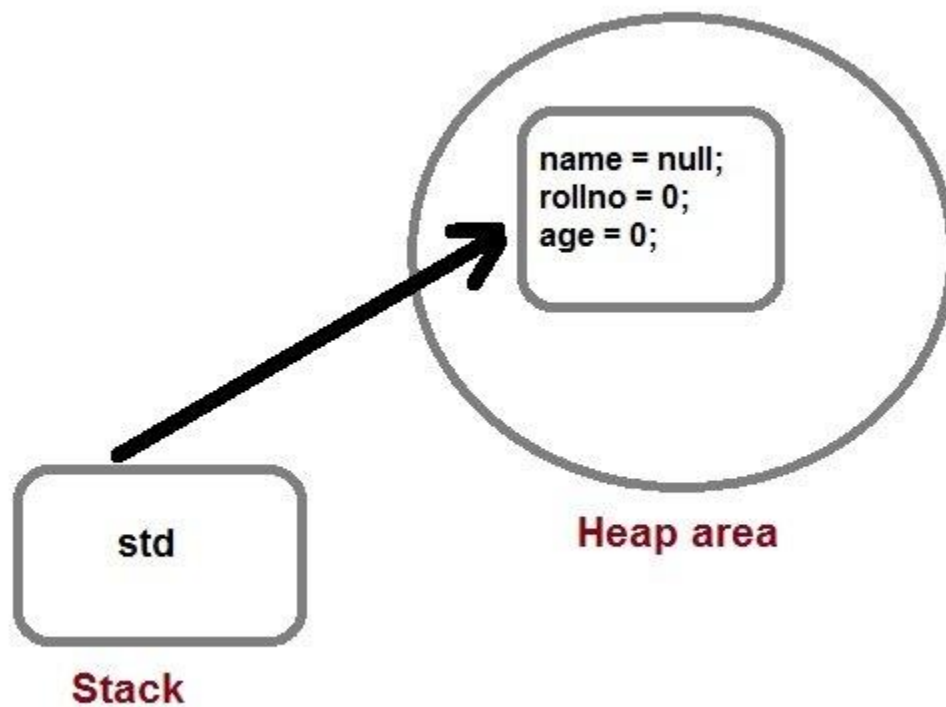
Although there are many other ways by which we can create object of the class. we have covered this section in details in a separate topics.

Example: Object creation

```
Student std = new Student();
```

Here, std is an object that represents the class Student during runtime.

The new keyword creates an actual physical copy of the object and assign it to the std variable. It will have physical existence and get memory in heap area. The new operator dynamically allocates memory for an object.



Example: Creating a Class and its object

```
public class Student{  
  
    String name;  
    int rollno;  
    int age;  
  
    void info(){  
        System.out.println("Name: "+name);  
        System.out.println("Roll Number: "+rollno);  
        System.out.println("Age: "+age);  
    }  
  
    public static void main(String[] args) {  
        Student student = new Student();  
  
        // Accessing and property value  
        student.name = "Ramesh";  
        student.rollno = 253;  
    }  
}
```



```

        student.age = 25;

        // Calling method
        student.info();
    }
}

```

Example: Class fields

```

public class Student{

    String name;
    int rollno;
    int age;

    void info(){
        System.out.println("Name: "+name);
        System.out.println("Roll Number: "+rollno);
        System.out.println("Age: "+age);
    }

    public static void main(String[] args) {
        Student student = new Student();

        // Calling method
        student.info();
    }
}

```

In case, if we don't initialize values of class fields then they are initialized with their default values.

Default values of instance variables

int, byte, short, long -> 0

float, double -> 0.0

string or any reference = null

boolean -> false

These values are initialized by the default constructor of JVM during object creation at runtime.

Methods in Java

Method in Java is similar to a function defined in other programming languages. Method describes behavior of an object. A method is a collection of statements that are grouped together to perform an operation.

For example, if we have a class Human, then this class should have methods like eating(), walking(), talking() etc, which describes the behavior of the object.

Declaring method is similar to function. See the syntax to declare the method in Java.

```
return-type methodName(parameter-list)
{
    //body of method
}
```

return-type refers to the type of value returned by the method.

methodName is a valid meaningful name that represent name of a method.


parameter-list represents list of parameters accepted by this method.

Method may have an optional return statement that is used to return value to the caller function.

Example of a Method:

Lets understand the method by simple example that takes a parameter and returns a string value.

```
public String getName(String st)
{
    String name="ducatindia ";
    name=name+st;
    return name;
}
```

`public String getName(String st)`

 modifier return-type method-name parameter

Modifier : Modifier are access type of method. We will discuss it in detail later.

Return Type : A method may return value. Data type of value return by a method is declare in method heading.

Method name : Actual name of the method.

Parameter : Value passed to a method.

Method body : collection of statement that defines what method does.

Calling a Method

Methods are called to perform the functionality implemented in it. We can call method by its name and store the returned value into a variable.

```
String val = GetName(".com")
```

Returning Multiple values

In Java, we can return multiple values from a method by using array. We store all the values into an array that want to return and then return back it to the caller method. We must specify return-type as an array while creating an array. Lets see an example.

Example:

Below is an example in which we return an array that holds multiple values.

```
class MethodDemo2{
    static int[] total(int a, int b)
    {
        int[] s = new int[2];
```

```

s[0] = a + b;
s[1] = a - b;
    return s;
}

    public static void main(String[] args)
    {
int[] s = total(200, 70);
System.out.println("Addition = " + s[0]);
System.out.println("Subtraction = " + s[1]);
    }
}

```

Return Object from Method

In some scenario there can be need to return object of a class to the caller function. In this case, we must specify class name in the method definition.

Below is an example in which we are getting an object from the method call. It can also be used to return collection of data.

Example:

In this example, we created a method get() that returns object of Demo class.

```

class Demo{
int a;
    double b;
int c;
Demo(int m, double d, int a)
{
    a = m;
    b = d;
    c = a;
}
}
class MethodDemo4{
    static Demo get(int x, int y)
    {
        return new Demo(x * y, (double)x / y, (x + y));
    }
    public static void main(String[] args)
    {
        Demo ans = get(25, 5);
System.out.println("Multiplication = " + ans.a);
    }
}

```

```
System.out.println("Division = " + ans.b);
System.out.println("Addition = " + ans.c);
    }
}
```

Parameter Vs. Argument in a Method

While talking about method, it is important to know the difference between two terms parameter and argument.

Parameter is variable defined by a method that receives value when the method is called. Parameter are always local to the method they dont have scope outside the method. While argument is a value that is passed to a method when it is called.

You can understand it by the below image that explain parameter and argument using a program example.

```
public void sum( int x, int y )
{
    System.out.println(x+y);
}
public static void main( String[ ] args )
{
    Test b=new Test( );
    b.sum( 10, 20 );
}
```

call-by-value and call-by-reference

There are two ways to pass an argument to a method

call-by-value : In this approach copy of an argument value is pass to a method. Changes made to the argument value inside the method will have no effect on the arguments.

call-by-reference : In this reference of an argument is pass to a method. Any changes made inside the method will affect the argument value.

NOTE :However there is no concept of call-by-reference in Java. Java supports only call by value.

Example of call-by-value

Lets see an example in which we are passing argument to a method and modifying its value.

```
public class Test
{
    public void callByValue(int x)
    {
        x=100;
    }
    public static void main(String[] args)
    {
        int x=50;
        Test t = new Test();
        t.callByValue(x);    //function call
        System.out.println(x);
    }
}
```

See, in the above example, value passed to the method does not change even after modified in the method. It shows that changes made to the value was local and argument was passed as call-by-value.

Java is Strictly Pass by Value

In Java, it is very confusing whether java is pass by value or pass by reference.

When the values of parameters are copied into another variable, it is known as pass by value and when a reference of a variable is passed to a method then it is known as pass by reference.

It is one of the feature of C language where address of a variable is passed during function call and that reflect changes to original variable.

In case of Java, we there is no concept of variable address, everything is based on object. So either we can pass primitive value or object value during the method call.

Note: Java is strictly pass by value. It means during method call, values are passed not addresses.

Example:

Below is an example in which value is passed by value and changed them inside the function but outside the function changes are not reflected to the original variable values.

```
class Add
{
    int x, y;
    Add(int i, int j)
```

```

        {
            x = i;
            y = j;
        }
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Add obj = new Add(5, 10);
        // call by value
        change(obj.x, obj.y);
        System.out.println("x = "+obj.x);
        System.out.println("y = "+obj.y);

    }
    public static void change(int x, int y)
    {
        x++;
        y++;
    }
}

```

Lets take another example in which an object is passed as value to the function, in which we change its variable value and the changes reflected to the original object. It seems like pass by reference but we differentiate it. Here member of an object is changed not the object.

Example:

Below is an example in which value is passed and changes are reflected.

```

class Add
{
    int x, y;
    Add(int i, int j)
    {
        x = i;
        y = j;
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Add obj = new Add(5, 10);
        // call by value (object is passed)
    }
}

```

```

        change(obj);
        System.out.println("x = "+obj.x);
        System.out.println("y = "+obj.y);
    }
    public static void change(Add add)
    {
        add.x++;
        add.y++;
    }
}

```

Constructors in Java

A constructor is a special method that is used to initialize an object. Every class has a constructor either implicitly or explicitly.

If we don't declare a constructor in the class then JVM builds a default constructor for that class. This is known as default constructor.

A constructor has same name as the class name in which it is declared. Constructor must have no explicit return type. Constructor in Java can not be abstract, static, final or synchronized. These modifiers are not allowed for constructor.

Syntax to declare constructor

```

className (parameter-list){
    code-statements
}

```

className is the name of class, as constructor name is same as class name.

parameter-list is optional, because constructors can be parameterize and non-parameterize as well.

Constructor Example

In Java, constructor structurally looks like given in below program. A Car class has a constructor that provides values to instance variables.

```

class Car
{
    String name ;
    String model;
    Car( ) //Constructor
    {

```



```

    name = "";
    model = "";
}
}

```

Types of Constructor

Java Supports two types of constructors:

Default Constructor

Parameterized constructor

Each time a new object is created at least one constructor will be invoked.

```

Car c = new Car()    //Default constructor invoked
Car c = new Car(name); //Parameterized constructor invoked

```

Default Constructor

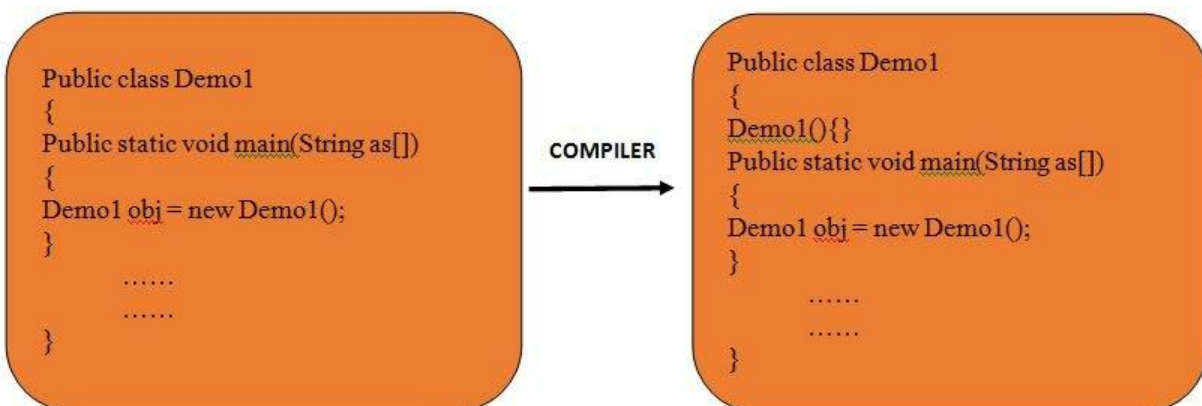
In Java, a constructor is said to be default constructor if it does not have any parameter. Default constructor can be either user defined or provided by JVM.

If a class does not contain any constructor then during runtime JVM generates a default constructor which is known as system define default constructor.

If a class contain a constructor with no parameter then it is known as default constructor defined by user. In this case JVM does not create default constructor.

The purpose of creating constructor is to initialize states of an object.

The below image shows how JVM adds a constructor to the class during runtime.



User Define Default Constructor

Constructor which is defined in the class by the programmer is known as user-defined default constructor.

Example:

In this example, we are creating a constructor that has same name as the class name.

```
class AddDemo1
{
    AddDemo1()
    {
        int a=10;
        int b=5;
        int c;
        c=a+b;
        System.out.println("*****Default Constructor*****");
        System.out.println("Total of 10 + 5 = "+c);
    }

    public static void main(String args[])
    {
        AddDemo1 obj=new AddDemo1();
    }
}
```

Constructor Overloading

Like methods, a constructor can also be overloaded. Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters. Constructor overloading is not much different than method overloading. In case of method overloading you have multiple methods with same name but different signature, whereas in Constructor overloading you have multiple constructor with different signature but only difference is that constructor doesn't have return type.

Example of constructor overloading

```
class Cricketer
{
    String name;
    String team;
    int age;
```

```

Cricketer () //default constructor.
{
    name = "";
    team = "";
    age = 0;
}
Cricketer(String n, String t, int a) //constructor overloaded
{
    name = n;
    team = t;
    age = a;
}
Cricketer (Cricketer ckt) //constructor similar to copy constructor of c++
{
    name = ckt.name;
    team = ckt.team;
    age = ckt.age;
}
public String toString()
{
    return "this is " + name + " of "+team;
}
}

```

Class test:

```

{
    public static void main (String[] args)
    {
        Cricketer c1 = new Cricketer();
        Cricketer c2 = new Cricketer("sachin", "India", 32);
        Cricketer c3 = new Cricketer(c2 );
        System.out.println(c2);
        System.out.println(c3);
        c1.name = "Virat";
        c1.team= "India";
        c1.age = 32;
        System.out.println(c1);
    }
}

```

Constructor Chaining

Constructor chaining is a process of calling one constructor from another constructor in the same class. Since constructor can only be called from another constructor, constructor chaining is used for this purpose.

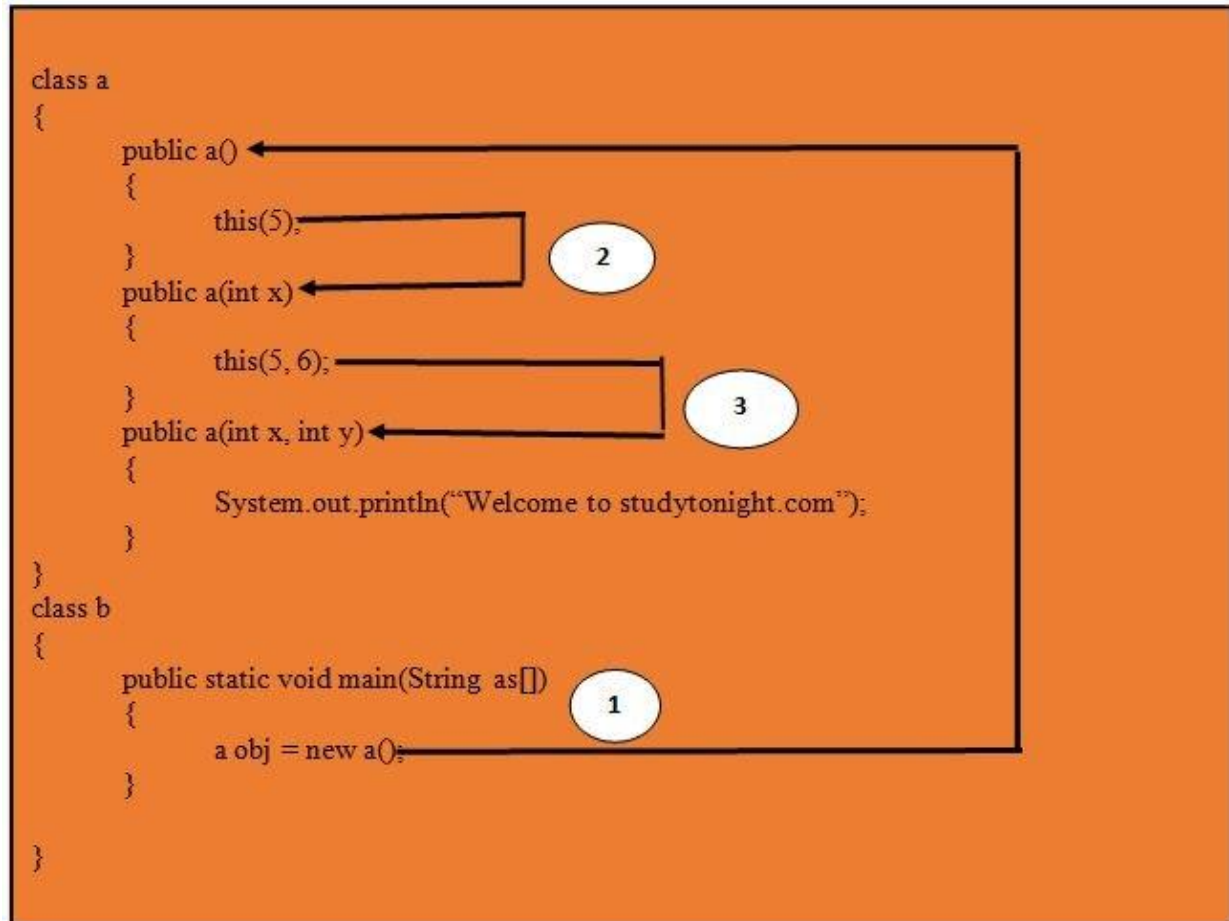
To call constructor from another constructor this keyword is used. This keyword is used to refer current object.

Lets see an example to understand constructor chaining.

```
class Test
{
    Test()
    {
        this(10);
    }
    Test(int x)
    {
        System.out.println("x="+x);
    }
    public static void main(String arg[])
    {
        Test object = new Test();
    }
}
```

Constructor chaining is used when we want to perform multiple tasks by creating a single object of the class.

In the below image, we have described the flow of constructor calling in the same class.



Example:

Lets see one more example to understand the constructor chaining. Here we have created three constructors and calling them using by using this keyword.

```
class abc
{
    public abc()
    {
        this(5);
        System.out.println("Default Constructor");
    }
    public abc(int x)
    {
        this(5, 6);
        System.out.println("Constructor with one Parameter");
        System.out.println("Value of x ==> "+x);
    }
}
```

```

    }
    public abc(int x, int y)
    {
        System.out.println("Constructor with two Parameter");
        System.out.println("Value of x and y ==> "+x+" "+y);
    }
}
class ChainingDemo1
{
    public static void main(String as[])
    {
        abcobj = new abc();
    }
}

```

Private Constructors

In Java, we can create private constructor to prevent class being instantiate. It means by declaring a private constructor, it restricts to create object of that class.

Private constructors are used to create singleton class. A class which can have only single object known as singleton class.

In private constructor, only one object can be created and the object is created within the class and also all the methods are static. An object can not be created if a private constructor is present inside a class. A class which have a private constructor and all the methods are static then it is called Utility class.

```

final class abc
{
    private abc()
    {}
    public static void add(int a, int b)
    {
        int z = a+b;
        System.out.println("Addition: "+z);
    }
    public static void sub(int x, int y)
    {
        int z = x-y;
        System.out.println("Subtraction: "+z);
    }
}

```

```
class PrivateConDemo
{
    public static void main(String as[])
    {
        abc.add(4, 5);
        abc.sub(5, 3);
    }
}
```

Access Modifiers in Java

Access modifiers are keywords in Java that are used to set accessibility. An access modifier restricts the access of a class, constructor, data member and method in another class.

Java language has four access modifier to control access level for classes and its members.

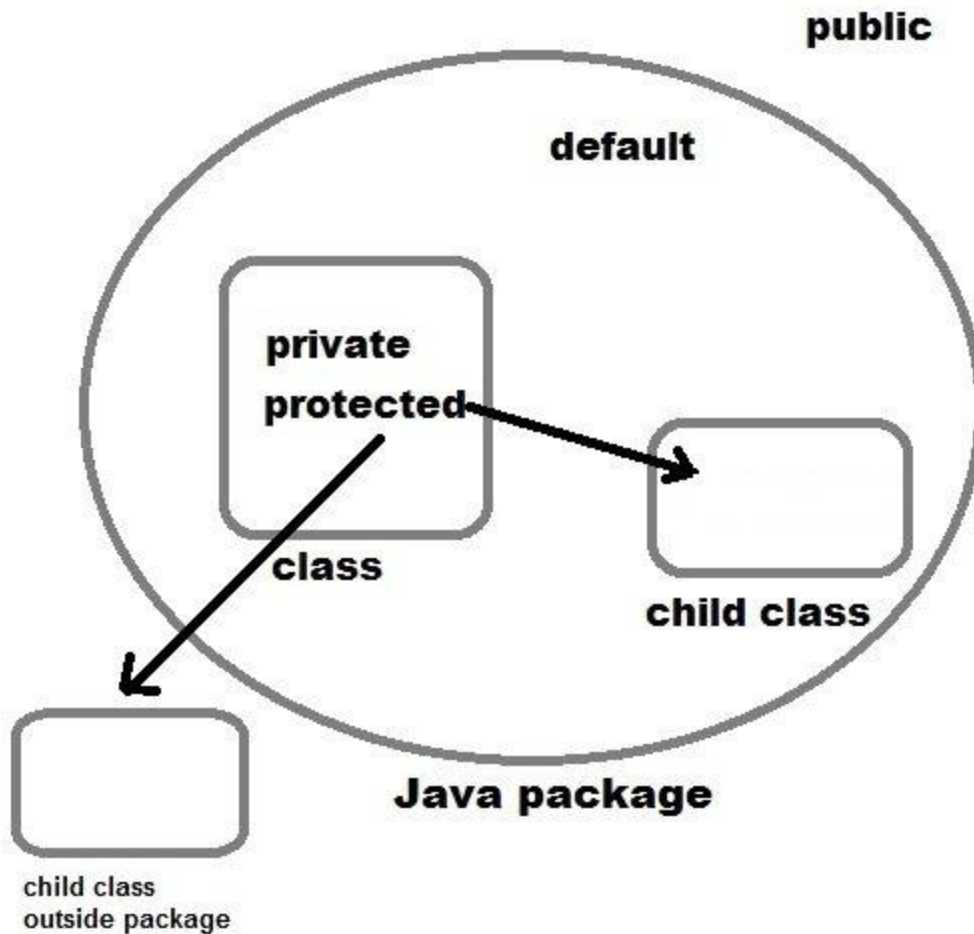
Default: Default has scope only inside the same package

Public: Public has scope that is visible everywhere

Protected: Protected has scope within the package and all sub classes

Private: Private has scope only within the classes

Java also supports many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient etc.



Default Access Modifier

If we don't specify any access modifier then it is treated as default modifier. It is used to set accessibility within the package. It means we can not access its method or class from outside the package. It is also known as package accessibility modifier.

Example:

In this example, we created a Demo class inside the package1 and another class Test by which we are accessing show() method of Demo class. We did not mention access modifier for the show() method that's why it is not accessible and reports an error during compile time.

```
package package1;
```

```
public class Demo {
```

```
    int a = 10;
```



```
// default access modifier
void show() {
    System.out.println(a);
}

}
```

```
import package1.Demo;
public class Test {

    public static void main(String[] args) {
        Demo demo = new Demo();
        demo.show(); // compile error
    }
}
```

Public Access Modifier

public access modifier is used to set public accessibility to a variable, method or a class. Any variable or method which is declared as public can be accessible from anywhere in the application.

Example:

Here, we have two class Demo and Test located in two different package. Now we want to access show method of Demo class from Test class. The method has public accessibility so it works fine. See the below example.

```
package package1;

public class Demo {

    int a = 10;
    // public access modifier
    public void show() {
        System.out.println(a);
    }

}
```

```
package package2;
```

```
import package1.Demo;
public class Test {

    public static void main(String[] args) {
        Demo demo = new Demo();
        demo.show();
    }
}
```

Protected Access Modifier

Protected modifier protects the variable, method from accessible from outside the class. It is accessible within class, and in the child class (inheritance) whether child is located in the same package or some other package.

Example:

In this example, Test class is extended by Demo and called a protected method show() which is accessible now due to inheritance.

```
package package1;

public class Demo {

    int a = 10;
    // public access modifier
    protected void show() {
        System.out.println(a);
    }

}
```

```
package package2;
import package1.Demo;
public class Test extends Demo{

    public static void main(String[] args) {
        Test test = new Test();
        test.show();
    }
}
```

Private Access Modifier

Private modifier is most restricted modifier which allows accessibility within same class only. We can set this modifier to any variable, method or even constructor as well.

Example:

In this example, we set private modifier to show() method and try to access that method from outside the class. Java does not allow to access it from outside the class.

```
class Demo {
```

```
    int a = 10;
    private void show() {
        System.out.println(a);
    }
}
```

```
public class Test {
```

```
    public static void main(String[] args) {
        Demo demo = new Demo();
        demo.show(); // compile error
    }
}
```

Non-access Modifier

Along with access modifiers, Java provides non-access modifiers as well. These modifier are used to set special properties to the variable or method.

Non-access modifiers do not change the accessibility of variable or method, but they provide special properties to them. Java provides following non-access modifiers.

- Final
- Static
- Transient
- Synchronized
- Volatile

Final Modifier

Final modifier can be used with variable, method and class. if variable is declared final then we cannot change its value. If method is declared final then it can not be overridden and if a class is declared final then we can not inherit it.

Static modifier

static modifier is used to make field static. We can use it to declare static variable, method, class etc. static can be use to declare class level variable. If a method is declared static then we don't need to have object to access that. We can use static to create nested class.

Transient modifier

When an instance variable is declared as transient, then its value doesn't persist when an object is serialized

Synchronized modifier

When a method is synchronized it can be accessed by only one thread at a time. We will discuss it in detail in Thread.

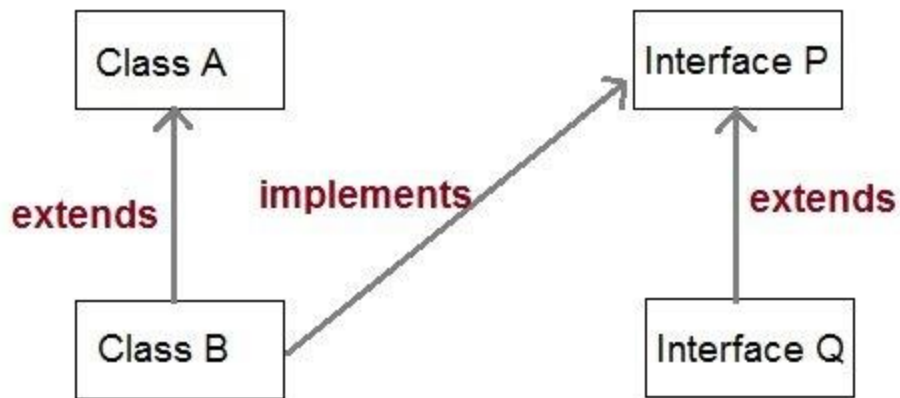
Volatile modifier

Volatile modifier tells to the compiler that the volatile variable can be changed unexpectedly by other parts of a program. Volatile variables are used in case of multi-threading program. volatile keyword cannot be used with a method or a class. It can be only used with a variable.

Inheritance (IS-A relationship) in Java

Inheritance is one of the key features of Object Oriented Programming. Inheritance provided mechanism that allowed a class to inherit property of another class. When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as Super class(Parent) and Sub class(child) in Java language.

Inheritance defines is-a relationship between a Super class and its Sub class. extends and implements keywords are used to describe inheritance in Java.



Let us see how extends keyword is used to achieve Inheritance. It shows super class and sub-class relationship.

```
class Vehicle
{
    .....
}
class Car extends Vehicle
{
    ..... //extends the property of vehicle class
}
```

Now based on above example. In OOPs term we can say that,

Vehicle is super class of Car.
Car is sub class of Vehicle.
Car IS-A Vehicle.

Purpose of Inheritance

It promotes the code reusability i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.
It promotes polymorphism by allowing method overriding.

Disadvantages of Inheritance

Main disadvantage of using inheritance is that the two classes (parent and child class) gets tightly coupled.

This means that if we change code of parent class, it will affect to all the child classes which is inheriting/deriving the parent class, and hence, it cannot be independent of each other.

Simple example of Inheritance

Before moving ahead let's take a quick example and try to understand the concept of Inheritance better,

```
class Parent
{
    public void p1()
    {
        System.out.println("Parent method");
    }
}
public class Child extends Parent {

    public void c1()
    {
        System.out.println("Child method");
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.c1(); //method of Child class
        cobj.p1(); //method of Parent class
    }
}
```

In the code above we have a class Parent which has a method p1(). We then create a new class Child which inherits the class Parent using the extends keyword and defines its own method c1(). Now by virtue of inheritance the class Child can also access the public method p1() of the class Parent.

Inheriting variables of super class

All the members of super class implicitly inherits to the child class. Member consists of instance variable and methods of the class.

Example

In this example the sub-class will be accessing the variable defined in the super class.

```
class Vehicle
{
    // variable defined
    String vehicleType;
```

```

}
public class Car extends Vehicle {

    String modelType;
    public void showDetail()
    {
        vehicleType = "Car"; //accessing Vehicle class member variable
        modelType = "Sports";
        System.out.println(modelType + " " + vehicleType);
    }
    public static void main(String[] args)
    {
        Car car = new Car();
        car.showDetail();
    }
}

```

Types of Inheritance

Java mainly supports only three types of inheritance that are listed below.

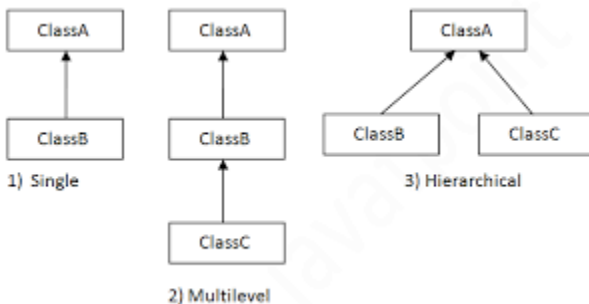
Single Inheritance

Multilevel Inheritance

Heirarchical Inheritance

NOTE: Multiple inheritance is not supported in java

We can get a quick view of type of inheritance from the below image.



Single Inheritance

When a class extends to another class then it forms single inheritance. In the below example, we have two classes in which class A extends to class B that forms single inheritance.

```

class A{

```

```

    int a = 10;
    void show() {
        System.out.println("a = "+a);
    }
}

public class B extends A{

    public static void main(String[] args) {
        B b = new B();
        b.show();

    }
}

```

Here, we can notice that show() method is declared in class A, but using child class Demo object, we can call it. That shows the inheritance between these two classes.

Multilevel Inheritance

When a class extends to another class that also extends some other class forms a multilevel inheritance. For example a class C extends to class B that also extends to class A and all the data members and methods of class A and B are now accessible in class C.

```

class A{
    int a = 10;
    void show() {
        System.out.println("a = "+a);
    }
}

class B extends A{
    int b = 10;
    void showB() {
        System.out.println("b = "+b);
    }
}

public class C extends B{

    public static void main(String[] args) {
        C c = new C();
    }
}

```



```
c.show();  
c.showB();  
}  
}
```

Hierarchical Inheritance

When a class is extended by two or more classes, it forms hierarchical inheritance. For example, class B extends to class A and class C also extends to class A in that case both B and C share properties of class A.

```
class A{  
    int a = 10;  
    void show() {  
        System.out.println("a = "+a);  
    }  
}
```

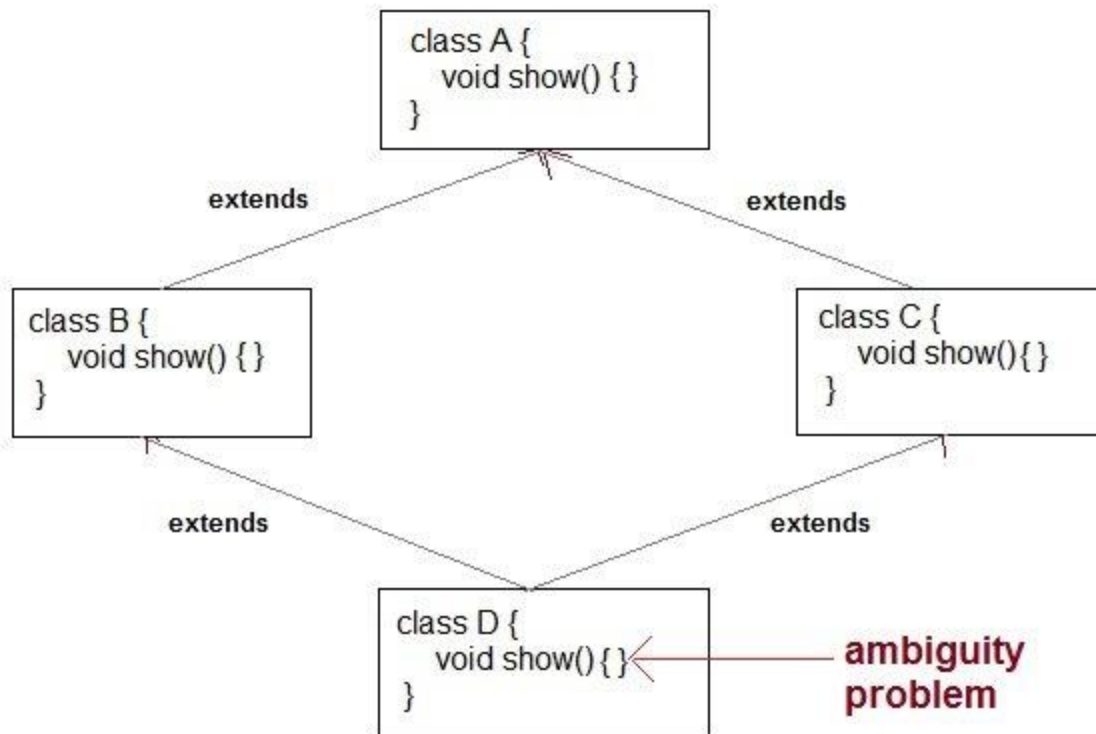
```
class B extends A{  
    int b = 10;  
    void showB() {  
        System.out.println("b = "+b);  
    }  
}
```

```
public class C extends A{  
  
    public static void main(String[] args) {  
        C c = new C();  
        c.show();  
        B b = new B();  
        b.show();  
    }  
}
```

Why multiple inheritance is not supported in Java?

To remove ambiguity.

To provide more maintainable and clear design.



super keyword

In Java, super keyword is used to refer to immediate parent class of a child class. In other words super keyword is used by a subclass whenever it need to refer to its immediate super class.

```

class Parent
{
    String name;
}
class Child extends Parent {
    String name;

    void detail()
    {
        super.name = "Parent";
        name = "Child";
    }
}

```

Example of Child class referring Parent class property using super keyword

In this example we will only focus on accessing the parent class property or variables.

```

class Parent
{
    String name;
}

public class Child extends Parent {
    String name;
    public void details()
    {
        super.name = "Parent"; //refers to parent class member
        name = "Child";
        System.out.println(super.name+" and "+name);
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.details();
    }
}

```

Example of Child class refering Parent class methods using super keyword

In this examle we will only focus on accessing the parent class methods.

```
class Parent
{
    String name;
    public void details()
    {
        name = "Parent";
        System.out.println(name);
    }
}
public class Child extends Parent {
    String name;
    public void details()
    {
        super.details();    //calling Parent class details() method
        name = "Child";
        System.out.println(name);
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.details();
    }
}
```

Example of Child class calling Parent class constructor using super keyword

In this examle we will focus on accessing the parent class constructor.

```
class Parent
{
    String name;

    public Parent(String n)
    {
        name = n;
    }
}
public class Child extends Parent {
    String name;
```

```

public Child(String n1, String n2)
{

    super(n1);    //passing argument to parent class constructor
    this.name = n2;
}
public void details()
{
    System.out.println(super.name+" and "+name);
}
public static void main(String[] args)
{
    Child cobj = new Child("Parent","Child");
    cobj.details();
}
}

```

Note: When calling the parent class constructor from the child class using super keyword, super keyword should always be the first line in the method/constructor of the child class.

Q. Can you use both this() and super() in a Constructor?

NO, because both super() and this() must be first statement inside a constructor. Hence we cannot use them together.

Aggregation (HAS-A relationship) in Java

Aggregation is a term which is used to refer one way relationship between two objects. For example, Student class can have reference of Address class but vice versa does not make sense.

In Java, aggregation represents HAS-A relationship, which means when a class contains reference of another class known to have aggregation.

The HAS-A relationship is based on usage, rather than inheritance. In other words, class A has-a relationship with class B, if class A has a reference to an instance of class B.

Lets understand it by an example and consider two classes Student and Address. Each student has own address that makes has-a relationship but address has student not makes any sense. We can understand it more clearly using Java code.

```

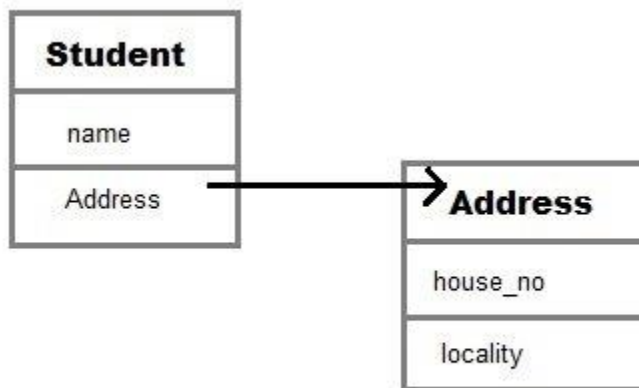
Class Address{
int street_no;
String city;
String state;
int pin;
Address(int street_no, String city, String state, int pin ){

```

```
this.street_no = street_no;  
this.city = city;  
this.state = state;  
this.pin = pin;  
}  
}
```

```
class Student  
{  
    String name;  
    Address ad;  
}
```

Here in the above code, we can see Student class has-a relationship with Address class. We have draw an image too to demonstrate relation between these two classes..



The Student class has an instance variable of type Address. As we have a variable of type Address in the Student class, it can use Address reference which is ad in this case, to invoke methods of the Address class.

Advantage of Aggregation

The main advantage of using aggregation is to maintain code re-usability. If an entity has a relation with some other entity than it can reuse code just by referring that.

Aggregation Example

Now lets understand it by using an example, here we created two classes Author and Book and Book class has a relation with Author class by having its reference. Now we are able to get all the properties of both class.

```

class Author
{
    String authorName;
    int age;
    String place;

    // Author class constructor
    Author(String name, int age, String place)
    {
        this.authorName = name;
        this.age = age;
        this.place = place;
    }
}

class Book
{
    String name;
    int price;
    // author details
    Author auther;
    Book(String n, int p, Author auther)
    {
        this.name = n;
        this.price = p;
        this.auther = auther;
    }
    public static void main(String[] args) {
        Author auther = new Author("John", 42, "USA");
        Book b = new Book("Java for Begginer", 800, auther);
        System.out.println("Book Name: "+b.name);
        System.out.println("Book Price: "+b.price);
        System.out.println("-----Auther Details-----");
        System.out.println("Auther Name: "+b.auther.authorName);
        System.out.println("Auther Age: "+b.auther.age);
        System.out.println("Auther place: "+b.auther.place);
    }
}

```

Lets take one more example to understand aggregation. Suppose we have one more class Publisher then the Book class can reuse Publisher class details by just using its reference as Author class. Lets understand it by Java code.

```
class Publisher{

    String name;
    String publisherID;
    String city;

    Publisher(String name, String publisherID, String city) {
        this.name = name;
        this.publisherID = publisherID;
        this.city = city;
    }
}
```

```
class Author
{
    String authorName;
    int age;
    String place;

    // Author class constructor
    Author(String name, int age, String place)
    {
        this.authorName = name;
        this.age = age;
        this.place = place;
    }
}
```

```
class Book
{
    String name;
    int price;
    // author details
    Author author;
    Publisher publisher;
    Book(String n, int p, Author author, Publisher publisher )
    {
        this.name = n;
        this.price = p;
        this.author = author;
        this.publisher = publisher;
    }
    public static void main(String[] args) {
        Author author = new Author("John", 42, "USA");
```



```

Publisher publisher = new Publisher("Sun Publication","JDSR-III4", "LA");
Book b = new Book("Java for Begginer", 800, auther, publisher);
System.out.println("Book Name: "+b.name);
System.out.println("Book Price: "+b.price);
System.out.println("-----Author Details-----");
System.out.println("Auther Name: "+b.auther.authorName);
System.out.println("Auther Age: "+b.auther.age);
System.out.println("Auther place: "+b.auther.place);
System.out.println("-----Publisher Details-----");
System.out.println("Publisher Name: "+b.publisher.name);
System.out.println("Publisher ID: "+b.publisher.publisherID);
System.out.println("Publisher City: "+b.publisher.city);
}
}

```

Composition in Java

Composition is a more restricted form of Aggregation. Composition can be described as when one class which includes another class, is dependent on it in such a way that it cannot functionally exist without the class which is included. For example a class Car cannot exist without Engine, as it won't be functional anymore.

Hence the word Composition which means the items something is made of and if we change the composition of things they change, similarly in Java classes, one class including another class is called a composition if the class included provides core functional meaning to the outer class.

```

class Car
{
    private Engine engine;
    Car(Engine en)
    {
        engine = en;
    }
}

```

Here by examining code, we can understand that if Car class does not have relationship with Engine class then Car does not have existence.

Composition is a design technique, not a feature of Java but we can achieve it using Java code.

Q. When to use Inheritance and Aggregation?

When you want to use some property or behaviour of any class without the requirement of modifying it or adding more functionality to it, in such case Aggregation is a better option because in case of Aggregation we are just using any external class inside our class as a variable.

Whereas when you want to use and modify some property or behaviour of any external class or may be want to add more function on top of it, its best to use Inheritance.

Method Overloading in Java

Method overloading is a concept that allows to declare multiple methods with same name but different parameters in the same class.

Java supports method overloading and always occur in the same class(unlike method overriding).

Method overloading is one of the ways through which java supports polymorphism. Polymorphism is a concept of object oriented programming that deal with multiple forms. We will cover polymorphism in separate topics later on.

Method overloading can be done by changing number of arguments or by changing the data type of arguments.

If two or more method have same name and same parameter list but differs in return type can not be overloaded.

Note: Overloaded method can have different access modifiers and it does not have any significance in method overloading.

There are two different ways of method overloading.

Different datatype of arguments

Different number of arguments

Method overloading by changing data type of arguments.

Example:

In this example, we have two sum() methods that take integer and float type arguments respectively.

Notice that in the same class we have two methods with same name but different types of parameters

```
class Calculate
{
    void sum (int a, int b)
    {
        System.out.println("sum is" +(a+b)) ;
    }
    void sum (float a, float b)
    {
        System.out.println("sum is" +(a+b));
    }
    Public static void main (String[] args)
    {
```

```

    Calculate cal = new Calculate();
    cal.sum (8,5);    //sum(int a, int b) is method is called.
    cal.sum (4.6f, 3.8f); //sum(float a, float b) is called.
}
}

```

You can see that sum() method is overloaded two times. The first takes two integer arguments, the second takes two float arguments.

Method overloading by changing no. of argument.

```

class Demo
{
    void multiply(int l, int b)
    {
        System.out.println("Result is" + (l*b));
    }
    void multiply(int l, int b, int h)
    {
        System.out.println("Result is" + (l*b*h));
    }
    public static void main(String[] args)
    {
        Demo ar = new Demo();
        ar.multiply(8,5); //multiply(int l, int b) is method is called
        ar.multiply(4,6,2); //multiply(int l, int b, int h) is called
    }
}

```

In this example the multiply() method is overloaded twice. The first method takes two arguments and the second method takes three arguments.

When an overloaded method is called Java look for match between the arguments to call the method and the its parameters. This match need not always be exact, sometime when exact match is not found, Java automatic type conversion plays a vital role.

Example of Method overloading with type promotion.

Java supports automatic type promotion, like int to long or float to double etc. In this example we are doing same and calling a function that takes one integer and second long type argument. At the time of calling we passed integer values and Java treated second argument as long type. See the below example.

```

class Demo

```

```

{
void sum(int l,long b)
{
    System.out.println("Sum is" +(l+b)) ;
}
void sum(int l, int b, int h)
{
    System.out.println("Sum is" +(l+b+h));
}
public static void main (String[] args)
{
    Demo ar = new Demo();
    ar.sum(8,5);
}
}

```

Method overloading if the order of parameters is changed

We can have two methods with same name and parameters but the order of parameters is different.

Example:

In this scenario, method overloading works but internally JVM treat it as method having different type of arguments

```

class Demo{

    public void get(String name, int id)
    {
        System.out.println("Company Name :"+ name);
        System.out.println("Company Id :"+ id);
    }

    public void get(int id, String name)
    {
        System.out.println("Company Id :"+ id);
        System.out.println("Company Name :"+ name);
    }
}

class MethodDemo8{
    public static void main (String[] args) {
        Demo obj = new Demo();
        obj.get("Cherry", 1);
        obj.get("Jhon", 2);
    }
}

```

```
}  
}
```

Overloading main Method

In Java, we can overload the main() method using different number and types of parameter but the JVM only understand the original main() method.

Example:

```
public class MethodDemo10{  
    public static void main(intargs)  
    {  
System.out.println("Main Method with int argument Executing");  
    System.out.println(args);  
    }  
    public static void main(char args)  
    {  
System.out.println("Main Method with char argument Executing");  
    System.out.println(args);  
    }  
    public static void main(Double[] args)  
    {  
System.out.println("Main Method with Double Executing");  
    System.out.println(args);  
    }  
    public static void main(String[] args)  
    {  
System.out.println("Original main Executing");  
    MethodDemo10.main(12);  
    MethodDemo10.main('c');  
    MethodDemo10.main(1236);  
    }  
}
```

Method overloading and null error

This is a general issue when working with objects, if same name methods having reference type parameters then be careful while passing arguments.

Below is an example in which you will know how a null value can cause an error when methods are overloaded.

Example:

Null value is a default value for all the reference types. It created ambiguity to JVM that reports error.

```
public class Demo
{
    public void test(Integer i)
    {
        System.out.println("test(Integer ) ");
    }
    public void test(String name)
    {
        System.out.println("test(String ) ");
    }
    public static void main(String [] args)
    {
        Demo obj = new Demo();
        obj.test(null);
    }
}
```

Reason:

The main reason for getting the compile time error in the above example is that here we have Integer and String as arguments which are not primitive data types in java and this type of argument do not accept any null value. When the null value is passed the compiler gets confused which method is to be selected as both the methods in the above example are accepting null.

However, we can solve this to pass specific reference type rather than value.

Example:

In this example, we are passing specific type argument rather than null value.

```
public class MethodDemo9
{
    public void test(Integer i)
    {
        System.out.println("Method ==> test(Integer)");
    }
    public void test(String name)
    {
        System.out.println("Method ==> test(String )");
    }
    public static void main(String [] args)
    {

```

```
MethodDemo9 obj = new MethodDemo9 ();
Integer a = null;
obj.test(a);
String b = null;
obj.test(b);
}
```

Method Overriding in Java

Method overriding is a process of overriding base class method by derived class method with more specific definition.

Method overriding performs only if two classes have is-a relationship. It means class must have inheritance. In other words, it is performed between two classes using inheritance relation.

In overriding, method of both class must have same name and equal number of parameters.

Method overriding is also referred to as runtime polymorphism because calling method is decided by JVM during runtime.

The key benefit of overriding is the ability to define method that's specific to a particular subclass type.

Rules for Method Overriding

1. Method name must be same for both parent and child classes.
2. Access modifier of child method must not be restrictive than parent class method.
3. Private, final and static methods cannot be overridden.
4. There must be an IS-A relationship between classes (inheritance).

Example of Method Overriding

Below we have simple code example with one parent class and one child class wherein the child class will override the method provided by the parent class.

```
class Animal
{
    public void eat()
    {
        System.out.println("Eat all eatables");
    }
}
```

```

class Dog extends Animal
{
    public void eat() //eat() method overridden by Dog class.
    {
        System.out.println("Dog like to eat meat");
    }

    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
    }
}

```

As you can see here Dog class gives it own implementation of eat() method. For method overriding, the method must have same name and same type signature in both parent and child class.

NOTE: Static methods cannot be overridden because, a static method is bounded with class where as instance method is bounded with object.

Example: Access modifier is more restrictive in child class

Java does not allows method overriding if child class has more restricted access modifier than parent class.

In the below example, to the child class method, we set protected which is restricted than public specified in parent class.

```

class Animal
{
    public void eat()
    {
        System.out.println("Eat all eatables");
    }
}

class Dog extends Animal
{
    protected void eat() //error
    {
        System.out.println("Dog like to eat meat");
    }

    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
    }
}

```



```
}  
}
```

Covariant return type

Since Java 5, it is possible to override a method by changing its return type. If subclass override any method by changing the return type of super class method, then the return type of overridden method must be subtype of return type declared in original method inside the super class. This is the only way by which method can be overridden by changing its return type.

Example :

```
class Animal
```

```
{  
    Animal getObj()  
    {  
        System.out.println("Animal object");  
        return new Animal();  
    }  
}
```

```
class Dog extends Animal
```

```
{  
    Dog getObj()    //Legal override after Java5 onward  
    { System.out.println("Dog object");  
      return new Dog();  
    }  
  
    public static void main(String[] args) {  
        new Dog().getObj();  
    }  
}
```

Difference between Overloading and Overriding

Method overloading and Method overriding seems to be similar concepts but they are not. Let's see some differences between both of them:

Method Overloading	Method Overriding
Parameter must be different and name must be same.	Both name and parameter must be same.
Compile time polymorphism.	Runtime polymorphism.
Increase readability of code.	Increase reusability of code.
Access specifier can be changed.	Access specifier cannot be more restrictive than original method(can be less restrictive).
It is Compiled Time Polymorphism.	It is Run Time Polymorphism.
It is performed within a class	It is performed between two classes using inheritance relation.
It is performed between two classes using inheritance relation.	It requires always inheritance.
It should have methods with the same name but a different signature.	It should have methods with same name and signature.
It can not have the same return type.	It should always have the same return type.
It can be performed using the static method	It can not be performed using the static method
It uses static binding	It uses the dynamic binding.
Access modifiers and Non-access modifiers can be changed.	Access modifiers and Non-access modifiers can not be changed.
It is code refinement technique.	It is a code replacement technique.
No keywords are used while defining the method.	Virtual keyword is used in the base class and overrides keyword is used in the derived class.
Private, static, final methods can be overloaded	Private, static, final methods can not be overloaded
No restriction is Throws Clause.	Restriction in only checked exception.
It is also known as Compile time polymorphism or static polymorphism or early binding	It is also known as Run time polymorphism or Dynamic polymorphism or Late binding

```

class OverloadingDemo{
static int add1(int x,int y){return x+y;}
static int add1(int x,int y,int z){return x+y+z;}
}

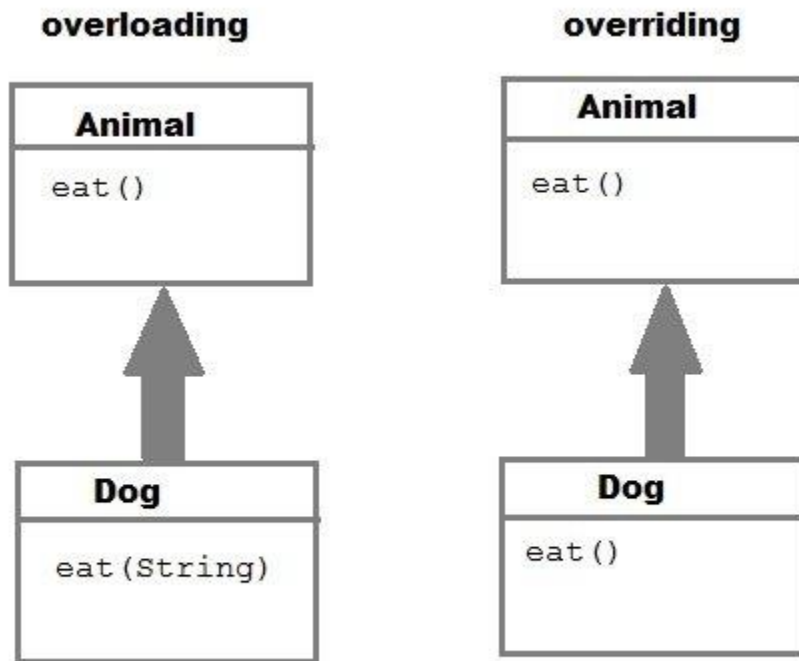
```

```

class Demo2{
void a()
{System.out.println("A");}}

```

```
class b extends c
{void a(){System.out.println("B");}}
```



Q. Can we Override static method? Explain with reasons?

No, we cannot override static method. Because static method is bound to class whereas method overriding is associated with object i.e at runtime.

Example: Overriding toString()

The toString() method of Object class is used to return string representation of an object.

Since object is super class of all java classes then we can override its string method to provide our own string presentation.

If we don't override string class and print object reference then it prints some hash code in "class_name @ hash code" format.

Below is an example of the overriding toString() method of Object class.

```
class Demo{
    private double a, b;
```

```

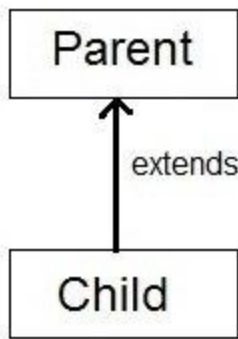
    public Demo(double a, double b)
    {
        this.a = a;
        this.b = b;
    }
    @Override
    public String toString()
    {
        return String.format(a + " + i" + b);
    }
}
public class MethodDemo11{
    public static void main(String[] args)
    {
        Demo obj1 = new Demo(25, 10);
        System.out.println(obj1);

    }
}

```

Runtime Polymorphism or Dynamic method dispatch

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.



```
Parent p = new Parent( );
```

```
Child c = new Child( );
```

```
Parent p = new Child( );
```

Upcasting

```
Child c = new Parent( );
```

incompatible type

Upcasting in Java

When Parent class reference variable refers to Child class object, it is known as Upcasting. In Java this can be done and is helpful in scenarios where multiple child classes extends one parent class. In those cases we can create a parent class reference and assign child class objects to it.

Let's take an example to understand it,

```
class Game
{
    public void type()
    {
        System.out.println("Indoor & outdoor");
    }
}
```

Class Cricket extends Game

```
{
    public void type()
    {
        System.out.println("outdoor game");
    }

    public static void main(String[] args)
    {
        Game gm = new Game();
    }
}
```

```

        Cricket ck = new Cricket();
        gm.type();
        ck.type();
        gm = ck;          //gm refers to Cricket object
        gm.type();        //calls Cricket's version of type
    }
}

```

Notice the last output. This is because of the statement, `gm = ck;`. Now `gm.type()` will call the Cricket class version of `type()` method. Because here `gm` refers to the cricket object.

Q. Difference between Static binding and Dynamic binding in Java?

Static binding in Java occurs during compile time while dynamic binding occurs during runtime. Static binding uses `type(Class)` information for binding while dynamic binding uses instance of class (`Object`) to resolve calling of method at run-time. Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

In simpler terms, Static binding means when the type of object which is invoking the method is determined at compile time by the compiler. While Dynamic binding means when the type of object which is invoking the method is determined at run time by the compiler.

Java this keyword

In Java, `this` is a keyword which is used to refer current object of a class. we can it to refer any member of the class. It means we can access any instance variable and method by using `this` keyword.

The main purpose of using `this` keyword is to solve the confusion when we have same variable name for instance and local variables.

We can use `this` keyword for the following purpose.

- `this` keyword is used to refer to current object.

- `this` is always a reference to the object on which method was invoked.

- `this` can be used to invoke current class constructor.

- `this` can be passed as an argument to another method.

Lets first understand the most general use of `this` keyword. As we said, it can be used to differentiate local and instance variables in the class.

Example:

In this example, we have three instance variables and a constructor that have three parameters with same name as instance variables. Now, we will use `this` to assign values of parameters to instance variables.

```

class Demo
{
    Double width, height, depth;
    Demo (double w, double h, double d)
    {
        this.width = w;
        this.height = h;
        this.depth = d;
    }
    public static void main(String[] args) {
        Demo d = new Demo(10,20,30);
        System.out.println("width = "+d.width);
        System.out.println("height = "+d.height);
        System.out.println("depth = "+d.depth);
    }
}

```

Here this is used to initialize member of current object. Such as, this.width refers to the variable of the current object and width only refers to the parameter received in the constructor i.e the argument passed while calling the constructor.

Calling Constructor using this keyword

We can call a constructor from inside the another function by using this keyword

Example:

In this example, we are calling a parameterized constructor from the non-parameterized constructor using the this keyword along with argument.

```

class Demo
{

    Demo ()
    {
        // Calling constructor
        this("ducatindia");
    }

    Demo(String str){

        System.out.println(str);

    }

    public static void main(String[] args) {

```

```
        Demo d = new Demo();
    }
}
```

Accessing Method using this keyword

This is another use of this keyword that allows to access method. We can access method using object reference too but if we want to use implicit object provided by Java then use this keyword.

Example:

In this example, we are accessing getName() method using this and it works fine as works with object reference. See the below example

```
class Demo
{
    public void getName()
    {
        System.out.println("ducatindia");
    }

    public void display()
    {
        this.getName();
    }

    public static void main(String[] args) {
        Demo d = new Demo();
        d.display();
    }
}
```

Return Current Object from a Method

In such scenario, where we want to return current object from a method then we can use this to solve this problem.

Example:

In this example, we created a method display that returns the object of Demo class. To return the object, we used this keyword and stored the returned object into Demo type reference variable. We used that returned object to call getName() method and it works fine.


```

class Demo
{
    public void getName()
    {
        System.out.println("ducatindia");
    }

    public Demo display()
    {
        // return current object
        return this;
    }

    public static void main(String[] args) {
        Demo d = new Demo();
        Demo d1 = d.display();
        d1.getName();
    }
}

```

Garbage Collection in Java

Java garbage collection is the process of releasing unused memory occupied by unused objects. This process is done by the JVM automatically because it is essential for memory management.

When a Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed.

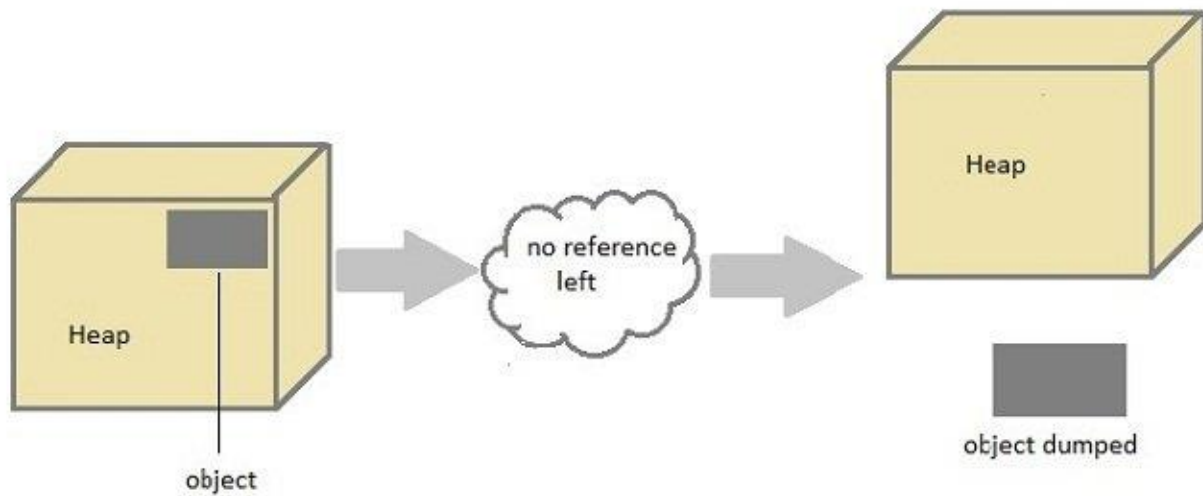
When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released. This technique is called Garbage Collection.

How Garbage Collection Works?

The garbage collection is a part of the JVM and is an automatic process done by JVM. We do not need to explicitly mark objects to be deleted. However, we can request to the JVM for garbage collection of an object but ultimately it depends on the JVM to call garbage collector.

Unlike C++ there is no explicit way to destroy object.

In the below image, you can understand that if an object does not have any reference than it will be dumped by the JVM.



Can the Garbage Collection be forced explicitly ?

No, the Garbage Collection can not be forced explicitly. We may request JVM for garbage collection by calling `System.gc()` method. But This does not guarantee that JVM will perform the garbage collection.

Advantages of Garbage Collection

Programmer doesn't need to worry about dereferencing an object.

It is done automatically by JVM.

Increases memory efficiency and decreases the chances for memory leak.

An object is able to get garbage collect if it is non-reference. We can make an object non-reference by using three ways.

1. set null to object reference which makes it able for garbage collection. For example:

```
Demo demo = new Demo();  
demo = null; // ready for garbage collection
```

2. We can non-reference an object by setting new reference to it which makes it able for garbage collection. For example

```
Demo demo = new Demo();  
Demo demo2 = new Demo();  
demo2 = demo // referring object
```

3. Anonymous object does not have any reference so if it is not in use, it is ready for the garbage collection.

finalize() method

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation finalize() method is used.

The finalize() method is called by garbage collection thread before collecting object. Its the last chance for any object to perform cleanup utility.

Signature of finalize() method

```
protected void finalize()
{
    //finalize-code
}
```

Some Important Points to Remember

It is defined in java.lang.Object class, therefore it is available to all the classes.

It is declare as protected inside Object class.

It gets called only once by a Daemon thread named GC (Garbage Collector) thread.

Request for Garbage Collection

We can request to JVM for garbage collection however, it is upto the JVM when to start the garbage collector.

Java gc() method is used to call garbage collector explicitly. However gc() method does not guarantee that JVM will perform the garbage collection. It only request the JVM for garbage collection. This method is present in System and Runtime class.

Example of gc() Method

Let's take an example and understand the functioning of the gc() method.

```
public class Test
{

    public static void main(String[] args)
    {
        Test t = new Test();
        t=null;
        System.gc();
    }
    public void finalize()
    {
```

```
        System.out.println("Garbage Collected");
    }
}
```

Static Block in Java

Static is a keyword in Java which is used to declare static stuffs. It can be used to create variable, method block or a class.

Static variable or method can be accessed without instance of a class because it belongs to class. Static members are common for all the instances of the class but non-static members are different for each instance of the class. Lets study how it works with variables and methods.

Static Variables

Static variables defined as a class member can be accessed without object of that class. Static variable is initialized once and shared among different objects of the class. All the object of the class having static variable will have the same instance of static variable.

Note: Static variable is used to represent common property of a class. It saves memory.

Example:

Suppose there are 100 employee in a company. All employee have its unique name and employee id but company name will be same all 100 employee. Here company name is the common property. So if you create a class to store employee detail, then declare company_name field as static

Below we have a simple class with one static variable, see the example.

```
class Employee
{
    int eid;
    String name;
    static String company = "ducatindia";

    public void show()
    {
        System.out.println(eid + "-" + name + "-" + company);
    }

    public static void main( String[] args )
    {
        Employee se1 = new Employee();
        se1.eid = 104;
        se1.name = "Abhijit";
    }
}
```

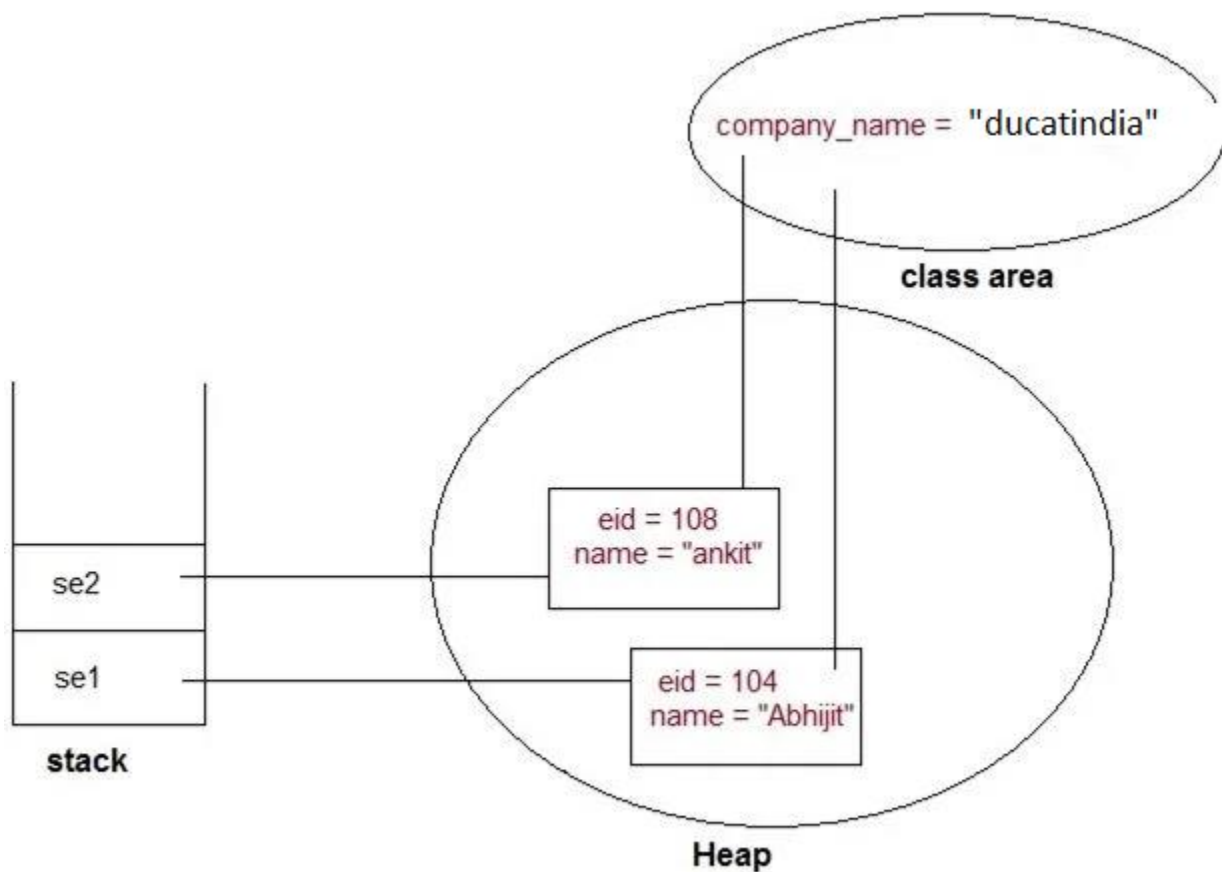
```

    se1.show();

    Employee se2 = new Employee();
    se2.eid = 108;
    se2.name = "ankit";
    se2.show();
}
}

```

We can understand it using the below image that shows different memory areas used by the program and how static variable is shared among the objects of the class.



When we have a class variable like this, defined using the static keyword, then the variable is defined only once and is used by all the instances of the class. Hence if any of the class instance modifies it then it is changed for all the other instances of the class.

Static variable vs Instance variable

Here are some differences between static variable and instance variables. Instance variables are non-static variables that store into heap and accessed through object only.

Static variable	Instance Variable
Represent common property	Represent unique property
Accessed using class name (can be accessed using object name as well)	Accessed using object
Allocated memory only once	Allocated new memory each time a new object is created

Example: static vs instance variables

Let's take an example and understand the difference.

```
public class Test
{
    static int x = 100;
    int y = 100;
    public void increment()
    {
        x++; y++;
    }
    public static void main( String[] args )
    {
        Test t1 = new Test();
        Test t2 = new Test();
        t1.increment();
        t2.increment();
        System.out.println(t2.y);
        System.out.println(Test.x); //accessed without any instance of class.
    }
}
```

See the difference in value of two variable. Static variable x shows the changes made to it by increment() method on the different object. While instance variable y show only the change made to it by increment() method on that particular instance.

Static Method in Java

A method can also be declared as static. Static methods do not need instance of its class for being accessed. main() method is the most common example of static method. main() method is declared as static because it is called before any object of the class is created.

Let's take an Example

```
class Test
{

    public static void square(int x)
    {
        System.out.println(x*x);
    }

    public static void main (String[] arg)
    {
        square(8) //static method square () is called without any instance of class.
    }
}
```

Static block in Java

Static block is used to initialize static data members. Static block executes even before main() method.

It executes when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they are programmed.

Example

Lets see an example in which we declared a static block to initialize the static variable.

```
class ST_Employee
{
    int eid;
    String name;
    static String company_name;

    static {
        company_name ="ducatindia"; //static block invoked before main() method
    }

    public void show()
    {
        System.out.println(eid+" "+name+" "+company_name);
    }
}
```

```

public static void main( String[] args )
{
    ST_Employee se1 = new ST_Employee();
    se1.eid = 104;
    se1.name = "Abhijit";
    se1.show();
}
}

```

Non-static (instance) variable cannot be referenced from a static context.

When we try to access a non-static variable from a static context like main method, java compiler throws an error message a non-static variable cannot be referenced from a static context. This is because non-static variables are related with instance of class(object) and they get created when instance of a class is created by using new operator. So if we try to access a non-static variable without any instance, compiler will complain because those variables are not yet created and they don't have any existence until an instance is created and associated with it.

Example

Let's take an example of accessing non-static variable from a static context.

```

class Test
{
    int x;
    public static void main(String[] args)
    {
        x = 10;
    }
}

```

Why main() method is static in java?

Because static methods can be called without any instance of a class and main() is called before any instance of a class is created.

Java Final Modifier

Final modifier is used to declare a field as final. It can be used with variable, method or a class.

If we declare a variable as final then it prevents its content from being modified. The variable acts like a constant. Final field must be initialized when it is declared.

If we declare a method as final then it prevents it from being overridden.

If we declare a class as final then it prevents from being inherited. We can not inherit final class in Java.

Example: Final variable

In this example, we declared a final variable and later on trying to modify its value. But final variable can not be reassigned so we get compile time error.

```
public class Test {  
  
    final int a = 10;  
    public static void main(String[] args) {  
        Test test = new Test();  
        test.a = 15; // compile error  
        System.out.println("a = "+test.a);  
    }  
}
```

Final Method

A method which is declared using final keyword known as final method. It is useful when we want to prevent a method from overridden.

Example: Final Method

In this example, we are creating a final method learn() and trying to override it but due to final keyword compiler reports an error.

```
class Ducat  
{  
    final void learn()  
    {  
        System.out.println("learning something new");  
    }  
}  
  
// concept of Inheritance  
class Student extends Ducat  
{  
    void learn()  
    {  
        System.out.println("learning something interesting");  
    }  
}
```

```

public static void main(String args[]) {
    Student object= new Student();
    object.learn();
}
}

```

This will give a compile time error because the method is declared as final and thus, it cannot be overridden. Don't get confused by the extends keyword, we will learn about this in the Inheritance tutorial which is next.

Let's take an another example, where we will have a final variable and method as well.

```

class Cloth
{
    final int MAX_PRICE = 999; //final variable
    final int MIN_PRICE = 699;
    final void display() //final method
    {
        System.out.println("Maxprice is" + MAX_PRICE );
        System.out.println("Minprice is" + MIN_PRICE);
    }
}

```

In the class above, the MAX_PRICE and MIN_PRICE variables are final hence there values cannot be changed once declared. Similarly the method display() is final which means even if some other class inherits the Cloth class, the definition of this method cannot be changed.

Final Class

A class can also be declared as final. A class declared as final cannot be inherited. The String class in java.lang package is an example of a final class.

We can create our own final class so that no other class can inherit it.

Example: Final Class

In this example, we created a final class ABC and trying to extend it from Demo class. but due to restrictions compiler reports an error. See the below example.

```

final class ABC{

```

```

        int a = 10;
        void show() {
            System.out.println("a = "+a);
        }
    }

    public class Demo extends ABC{

        public static void main(String[] args) {

            Demo demo = new Demo();

        }
    }

```

Java Blank Final Variable

Final variable that is not initialized at the time of declaration is called blank final variable. Java allows to declare a final variable without initialization but it should be initialized by the constructor only. It means we can set value for blank final variable in a constructor only.

Example:

In this example, we created a blank final variable and initialized it in a constructor which is acceptable.

```

public class Demo{
    // blank final variable
    final int a;

    Demo(){
        // initialized blank final
        a = 10;
    }

    public static void main(String[] args) {

        Demo demo = new Demo();
        System.out.println("a = "+demo.a);

    }
}

```

Static Blank Final Variable

A blank final variable declared using static keyword is called static blank final variable. It can be initialized in static block only.

Static blank final variables are used to create static constant for the class.

Example

In this example, we are creating static blank final variable which is initialized within a static block. And see, we used class name to access that variable because for accessing static variable we don't need to create object of that class.

```
public class Demo{
    // static blank final variable
    static final int a;

    static{
        // initialized static blank final
        a = 10;
    }

    public static void main(String[] args) {

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

    }
}
```

Java instanceof Operator and Downcasting

In Java, instanceof is an operator which is used to check object reference. It checks whether the reference of an object belongs to the provided type or not. It returns either true or false, if an object reference is of specified type then it return true otherwise false.

We can use instanceof operator to check whether an object reference belongs to parent class, child class, or an interface.

It's also known as type comparison operator because it compares the instance with type.

Application of instanceof

Apart from testing object type, we can use it for object downcasting. However, we can perform downcasting using typecasting but it may raise ClassCastException at runtime.

To avoid the exception, we use instanceof operator. Doing this helps to perform casting.

When we do typecast, it is always a good idea to check if the typecasting is valid or not. instanceof helps us here. We can always first check for validity using instanceof, then do typecasting.

Syntax for declaring instanceof operator is given below.

(object) instanceof (type)

object: It is an object reference variable.

type: It can be a class or an interface.

Time for an Example

Let's take an example to understand the use of instanceof operator. Here we are testing the reference type is type of Test class and it returns true.

```
public class Test
{
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t instanceof Test);
    }
}
```

Example: Interface Reference Type

Lets create an interface and test reference type using instanceof operator. This operator returns if the reference object refers to right interface type. See the below example

```
interface Callable{
    void call();
}
```

```
class Demo implements Callable {
    public void call() {
        System.out.println("Calling...");
    }
    public static void main(String[] args) {

        Callable d = new Demo();
        // Refer to a class
```

```

    System.out.println((d instanceof Demo));
    // Refer to an interface
    System.out.println((d instanceof Callable));

}
}

```

If reference variable holds null then instanceof returns false. Lets see one more example.

```

class Demo {

    public static void main(String[] args) {

        Demo d = null;
        System.out.println(d instanceof Demo);

    }
}

```

Example : Inheritance

Lets take another example to understand instanceof operator, in which we created 5 classes and some of them extends to another class. Now by creating objects we are testing reference object belongs to which class.

```

class Parent{}

class Child1 extends Parent{}

class Child2 extends Parent{}

class Demo
{
    public static void main(String[] args)
    {
        Parent p =new Parent();
        Child1 c1 = new Child1();
        Child2 c2 = new Child2();

        System.out.println(c1 instanceof Parent);    //true
        System.out.println(c2 instanceof Parent);    //true
        System.out.println(p instanceof Child1);     //false
    }
}

```

```

System.out.println(p instanceof Child2);    //false

p = c1;
System.out.println(p instanceof Child1);    //true
System.out.println(p instanceof Child2);    //false

p = c2;
System.out.println(p instanceof Child1);    //false
System.out.println(p instanceof Child2);    //true

}

}

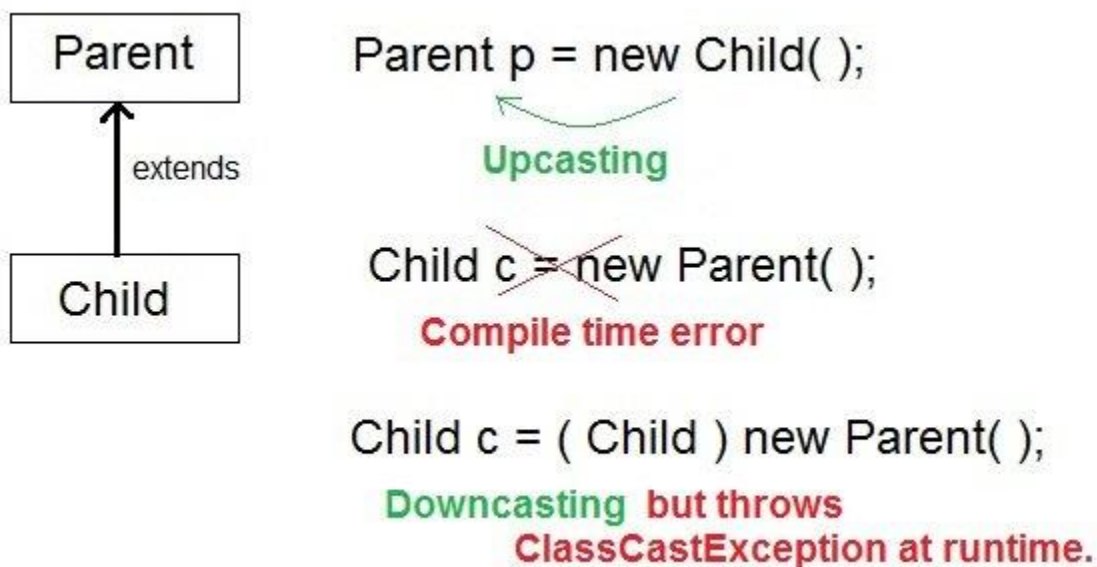
```

Downcasting in Java

Downcasting is one of the advantages of using instanceof operator. Downcasting is a process to hold object of parent class in child class reference object. It is reverse of upcasting, in which object of child is assigned to parent class reference object.

However, we can perform downcasting using typecasting but it throws `ClassCastException` at run-time. So to avoid this we use instanceof operator to perform downcasting.

You can understand whole process by the below image.



Example of downcasting with instanceof operator

```
class Parent{ }

public class Child extends Parent
{
    public void check()
    {
        System.out.println("Sucessfull Casting");
    }

    public static void show(Parent p)
    {
        if(p instanceof Child)
        {
            Child b1=(Child)p;
            b1.check();
        }
    }

    public static void main(String[] args)
    {

        Parent p=new Child();

        Child.show(p);

    }
}
```

Java Package

Package is a collection of related classes. Java uses package to group related classes, interfaces and sub-packages in any Java project.

We can assume package as a folder or a directory that is used to store similar files.

In Java, packages are used to avoid name conflicts and to control access of class, interface and enumeration etc. Using package it becomes easier to locate the related classes and it also provides a good structure for projects with hundreds of classes and other files.

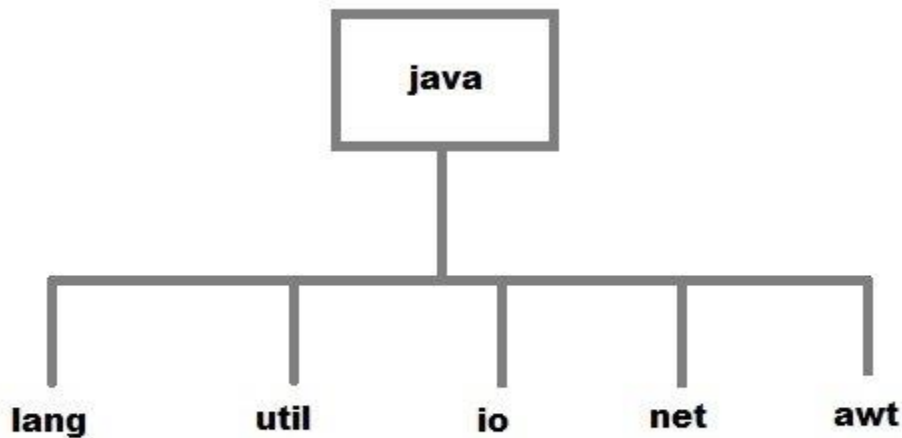
Lets understand it by a simple example, Suppose, we have some math related classes and interfaces then to collect them into a simple place, we have to create a package.

Types Of Java Package

Package can be built-in and user-defined, Java provides rich set of built-in packages in form of API that stores related classes and sub-packages.

Built-in Package: math, util, lang, i/o etc are the example of built-in packages.

User-defined-package: Java package created by user to categorize their project's classes and interface are known as user-defined packages.



How to Create a Package

Creating a package in java is quite easy, simply include a package command followed by name of the package as the first statement in java source file.

```
package mypack;  
public class employee  
{  
    String empld;  
    String name;  
}
```

The above statement will create a package with name mypack in the project directory.

Java uses file system directories to store packages. For example the .java file for any class you define to be part of mypack package must be stored in a directory called mypack.

Additional points about package:

Package statement must be first statement in the program even before the import statement.

A package is always defined as a separate folder having the same name as the package name.

Store all the classes in that package folder.

All classes of the package which we wish to access outside the package must be declared public.

All classes within the package must have the package statement as its first line.

All classes of the package must be compiled before use.

Example of Java packages

Now let's understand package creation by an example, here we created a learnjava package that stores the FirstProgram class file.

```
//save as FirstProgram.java
package learnjava;
public class FirstProgram{
    public static void main(String args[]) {
        System.out.println("Welcome to package example");
    }
}
```

How to compile Java programs inside packages?

This is just like compiling a normal java program. If you are not using any IDE, you need to follow the steps given below to successfully compile your packages:

```
javac -d . FirstProgram.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run Java package program?

To run the compiled class that we compiled using above command, we need to specify package name too. Use the below command to run the class file.

```
java learnjava.FirstProgram
```

After running the program, we will get "Welcome to package example" message to the console. You can tally that with print statement used in the program.

How to import Java Package

To import java package into a class, we need to use java import keyword which is used to access package and its classes into the java program.

Use import to access built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to any class that is present in a different package:

without import the package

import package with specified class

import package with all classes

Lets understand each one with the help of example.

Accessing package without import keyword

If you use fully qualified name to import any class into your program, then only that particular class of the package will be accessible in your program, other classes in the same package will not be accessible. For this approach, there is no need to use the import statement. But you will have to use the fully qualified name every time you are accessing the class or the interface. This is generally used when two packages have classes with same names. For example: java.util and java.sql packages contain Date class.

Example

In this example, we are creating a class A in package pack and in another class B, we are accessing it while creating object of class A.

```
//save by A.java
package pack;
public class A {
    public void msg() {
        System.out.println("Hello");
    }
}
```

```
//save by B.java
package mypack;
class B {
    public static void main(String args[]) {
        pack.A obj = new pack.A(); //using fully qualified name
        obj.msg();
    }
}
```

Import the Specific Class

Package can have many classes but sometimes we want to access only specific class in our program in that case, Java allows us to specify class name along with package name. If we use import packagename.classname statement then only the class with name classname in the package will be available for use.

Example:

In this example, we created a class Demo stored into pack package and in another class Test, we are accessing Demo class by importing package name with class name.

```
//save by Demo.java
package pack;
public class Demo {
    public void msg() {
        System.out.println("Hello");
    }
}

//save by Test.java
package mypack;
import pack.Demo;
class Test {
    public static void main(String args[]) {
        Demo obj = new Demo();
        obj.msg();
    }
}
```

Import all classes of the package

If we use packagename.* statement, then all the classes and interfaces of this package will be accessible but the classes and interface inside the sub-packages will not be available for use.

The import keyword is used to make the classes of another package accessible to the current package.

Example :

In this example, we created a class First in learnjava package that access it in another class Second by using import keyword.