Introduction:

1. Java - What, Where and Why?
2. What is Java
3. Where Java is used
4. Java Applications

Java Programming Tutorial is a widely used robust technology. Let's start learning of java from basic questions like what is java tutorial, core java, where it is used, what type of applications are created in java and why use java.

What is Java?

Java is a **programming language** and a **platform**.Java is a high level, robust, secured and object-oriented programming language.

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

Java Example

Let's have a quick look at java programming example. A detailed description of hello java example is given:

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

Where it is used?

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

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

Types of Java Applications

There are mainly 4 type of applications that can be created using java programming:

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

History of Java

1. Brief history of Java
2. Java Version History

Java history is interesting to know. The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.

James Gosling

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

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

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

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

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

Why Oak name for java language?

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

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

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

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

8) Java is an island of Indonesia where first coffee was produced (called java coffee).

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

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

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

12) JDK 1.0 released in(January 23, 1996).

Java Version History

There are many java versions that has been released. Current stable release of Java is Java SE 8.

JDK Alpha and Beta (1995), JDK 1.0 (23rd Jan, 1996), JDK 1.1 (19th Feb, 1997),

J2SE 1.2 (8th Dec, 1998), J2SE 1.3 (8th May, 2000), J2SE 1.4 (6th Feb, 2002)

J2SE 5.0 (30th Sep, 2004), Java SE 6 (11th Dec, 2006), Java SE 7 (28th July, 2011)

Java SE 8 (18th March, 2014)

Features of Java

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

Simple, Object-Oriented, Platform independent, Secured, Robust, Architecture neutral, Portable, Dynamic, Interpreted, High Performance, Multithreaded, Distributed

Simple

According to sun, Java language is simple because:

It removed many confusion and rarely-used features e.g. explicit pointers, operator overloading etc.
No need to remove unreferenced objects because there is Automatic Garbage Collection in Java.

Object-oriented

Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.

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

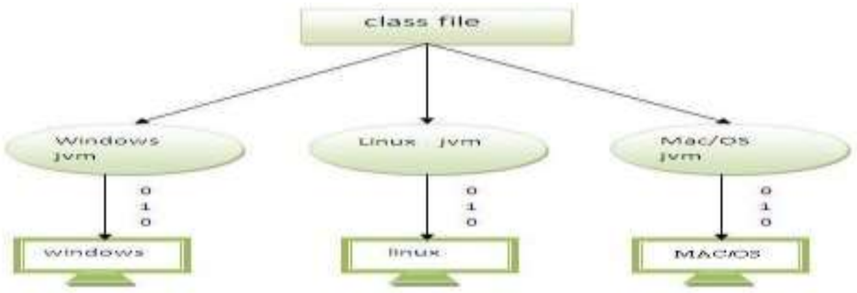
Basic concepts of OOPs are:

Object, Class, Inheritance, Polymorphism, Abstraction, Encapsulation

Platform Independent

A platform is the hardware or software environment in which a program runs.
There are two types of platforms software-based and hardware-based. Java provides software-based platform.
The Java platform differs from most other platforms in the sense that it's a software-based platform that runs on top of other hardware-based platforms.It has two components:

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

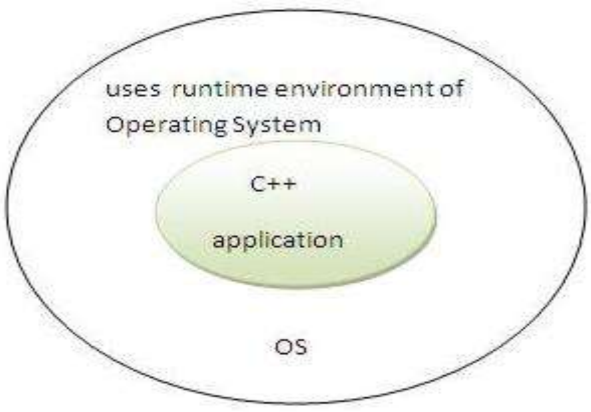


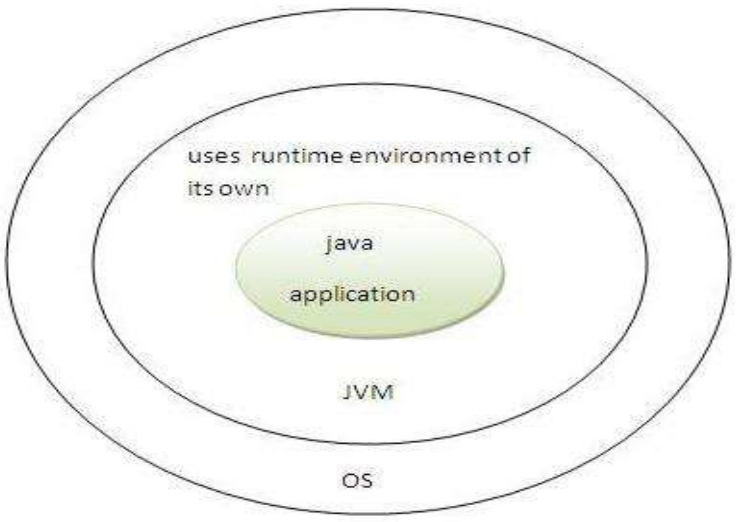
Java code can be run on multiple platforms e.g.Windows,Linux,Sun Solaris,Mac/OS etc.Java code is compiled by the compiler and converted into bytecode.This bytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

Secured

Java is secured because:

- No explicit pointer
- Programs run inside virtual machine sandbox.





- **Classloader-** adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier-** checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager-** determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL,JAAS,cryptography etc.

Robust

Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral

There is no implementation dependent features e.g. size of primitive types is set.

Portable

We may carry the java bytecode to any platform.

High-performance

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

Simple Program of Java

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

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

Requirement for Hello Java Example: For executing any java program, you need to

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

Creating hello java example

Let's create the hello java program:

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

Save this file as Simple.java

To compile: javac Simple.java

To execute: java Simple

Output: Hello Java

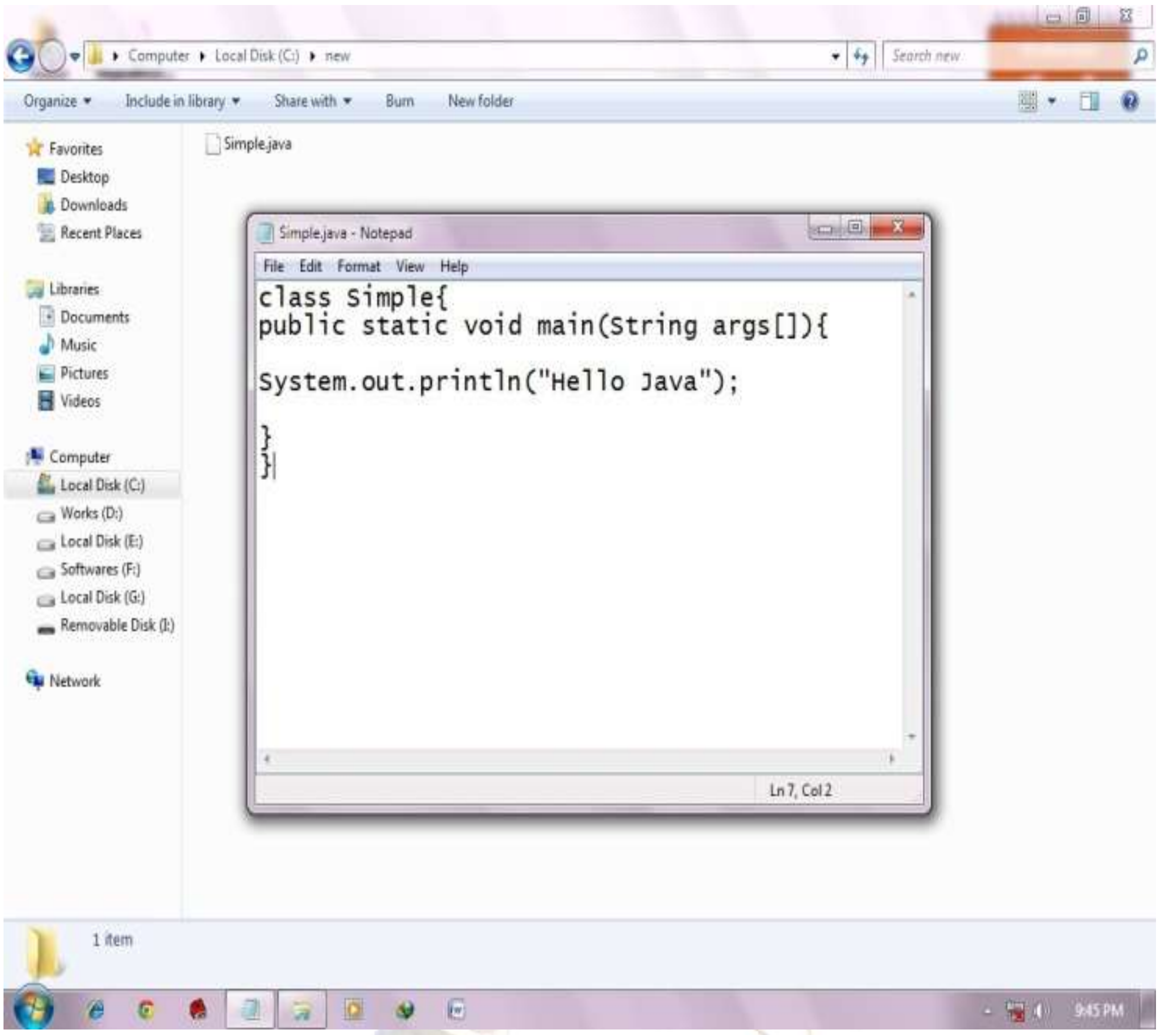
Understanding first java program

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

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
- **void** is the return type of the method, it means it doesn't return any value.
- **main** represents startup of the program.

- **String[] args** is used for command line argument. We will learn it later.
- **System.out.println()** is used print statement. We will learn about the internal working of System.out.println statement later.

To write the simple program, open notepad by **start menu -> All Programs -> Accessories -> notepad** and write simple program as displayed below:



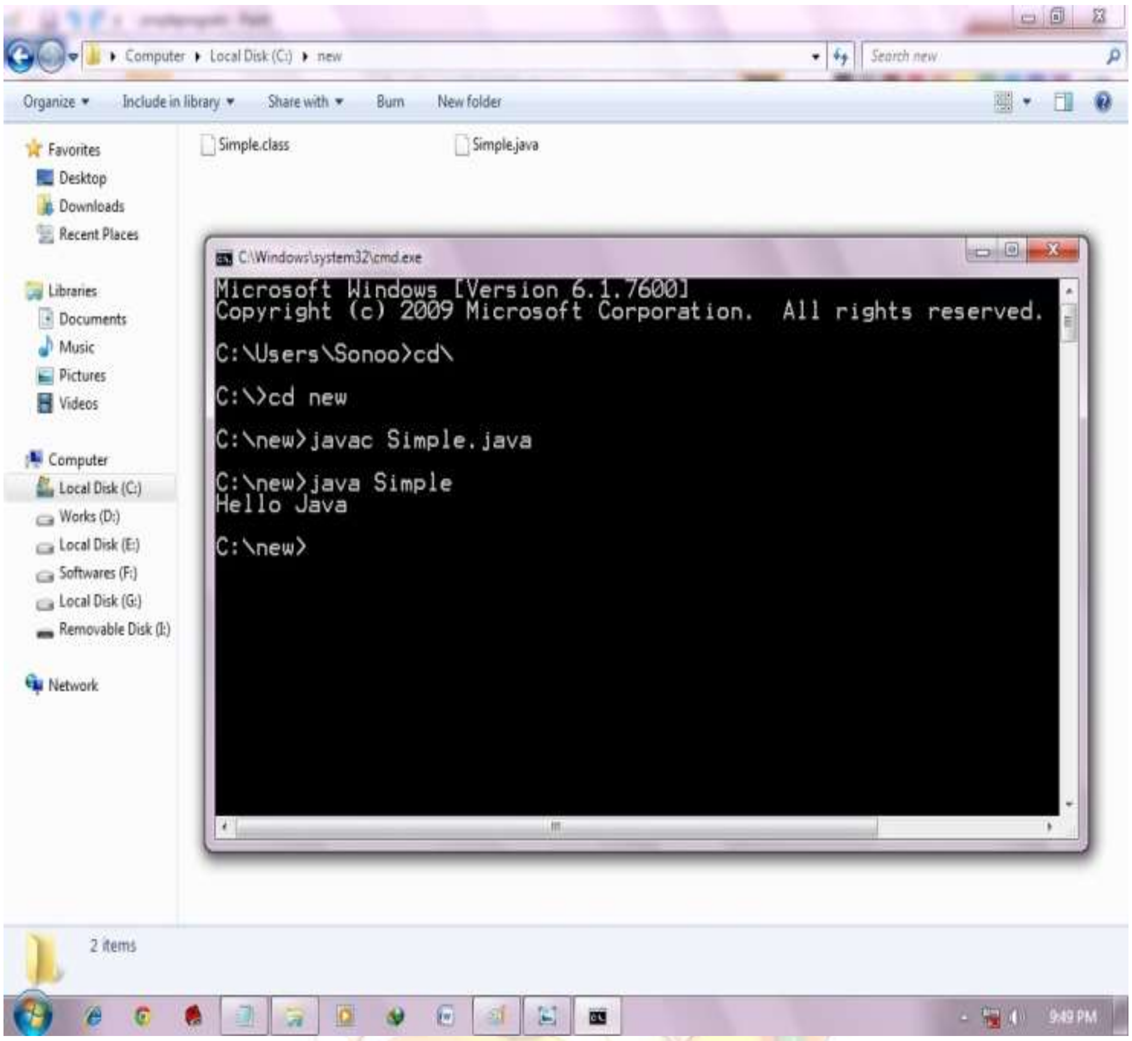
```
class Simple{
public static void main(String args[]){

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

}
}
```

As displayed in the above diagram, write the simple program of java in notepad and saved it as Simple.java.

To compile and run this program, you need to open command prompt by **start menu -> All Programs -> Accessories -> command prompt**.



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

C:\Users\Sonoo>cd\

C:\>cd new

C:\new>javac Simple.java

C:\new>java Simple
Hello Java

C:\new>
```

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

To compile: javac Simple.java

To execute: java Simple

How many ways can we write a java program?

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

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

Let's see the simple code of main method.

1) Static public void main(String args[])

2) Subscript notation in java array can be used after type, before variable or after variable.

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

```
1. public static void main(String[] args)
2. public static void main(String []args)
3. public static void main(String args[])
```

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

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

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

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

Let's see the simple code.

```
1. class A{
2. static public void main(String... args){
3. System.out.println("hello java4");
4. } };
```

Valid java main method signature:

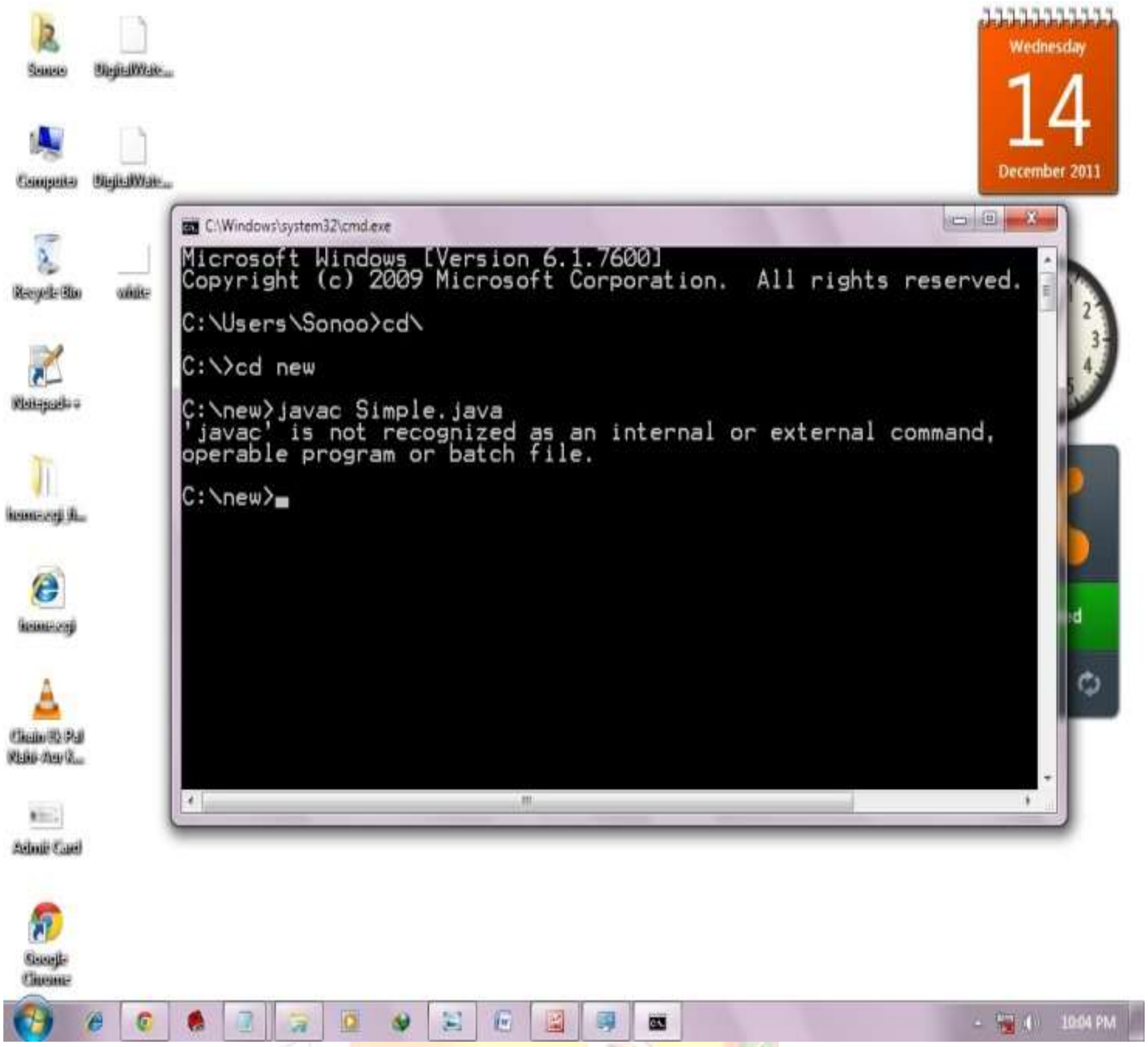
```
1. public static void main(String[] args)
2. public static void main(String []args)
3. public static void main(String args[])
4. public static void main(String... args)
5. static public void main(String[] args)
6. public static final void main(String[] args)
7. final public static void main(String[] args)
8. final strictfp public static void main(String[] args)
```

Invalid java main method signature:

```
1. public void main(String[] args)
2. static void main(String[] args)
3. public void static main(String[] args)
4. abstract public static void main(String[] args)
```

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

If there occurs a problem like displayed in the below figure, you need to set path. Since DOS doesn't know javac or java, we need to set path. Path is not required in such a case if you save your program inside the jdk/bin folder. But its good approach to set path. Click here for How to set path in java.



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

C:\Users\Sonoo>cd\

C:\>cd new

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

C:\new>
```

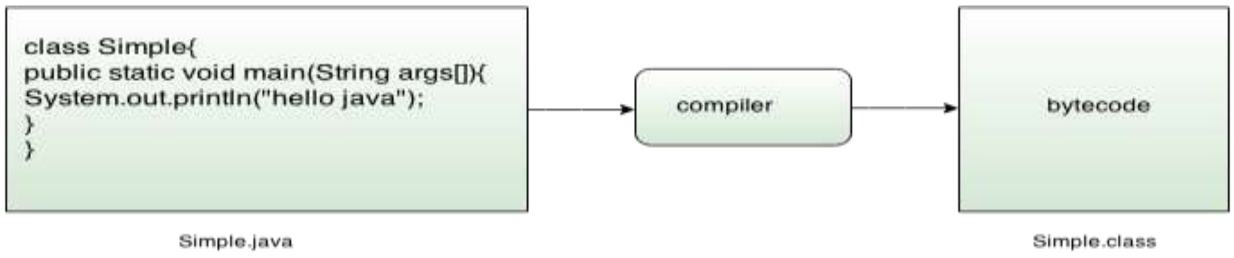
Internal Details of Hello Java Program

1. Internal Details of Hello Java

In the previous page, we have learned about the first program, how to compile and how to run the first java program. Here, we are going to learn, what happens while compiling and running the java program. Moreover, we will see some question based on the first program.

What happens at compile time?

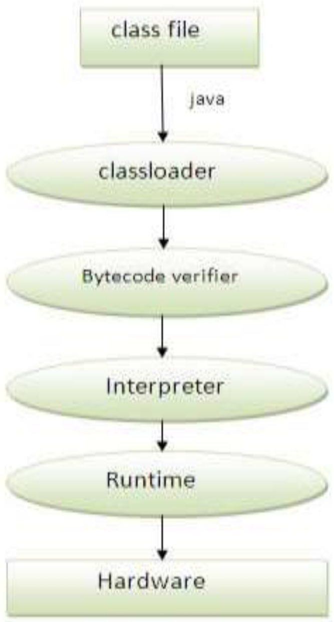
At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.



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

What happens at runtime?

At runtime, following steps are performed:



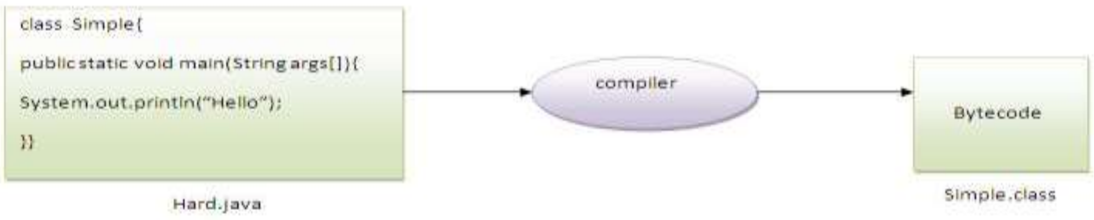
Classloader: is the subsystem of JVM that is used to load class files.

Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects.

Interpreter: read bytecode stream then execute the instructions.

Q)Can you save a java source file by other name than the class name?

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

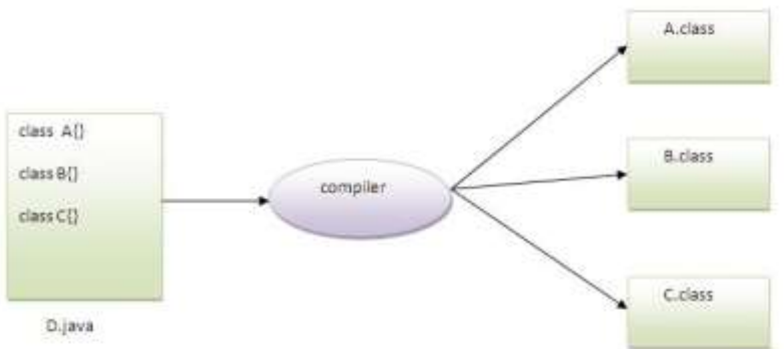

class Simple{
public static void main(String args[]){
System.out.println("Hello");
}}
Hard.java
compiler
Bytecode
Simple.class

To compile: javac Hard.java

To execute: java Simple

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

Yes, like the figure given below illustrates:



How to set path in Java:

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

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

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

But If you are having your java file outside the jdk/bin folder, it is necessary to set path of JDK.

There are 2 ways to set java path:

1. temporary
2. permanent

1) How to set Temporary Path of JDK in Windows

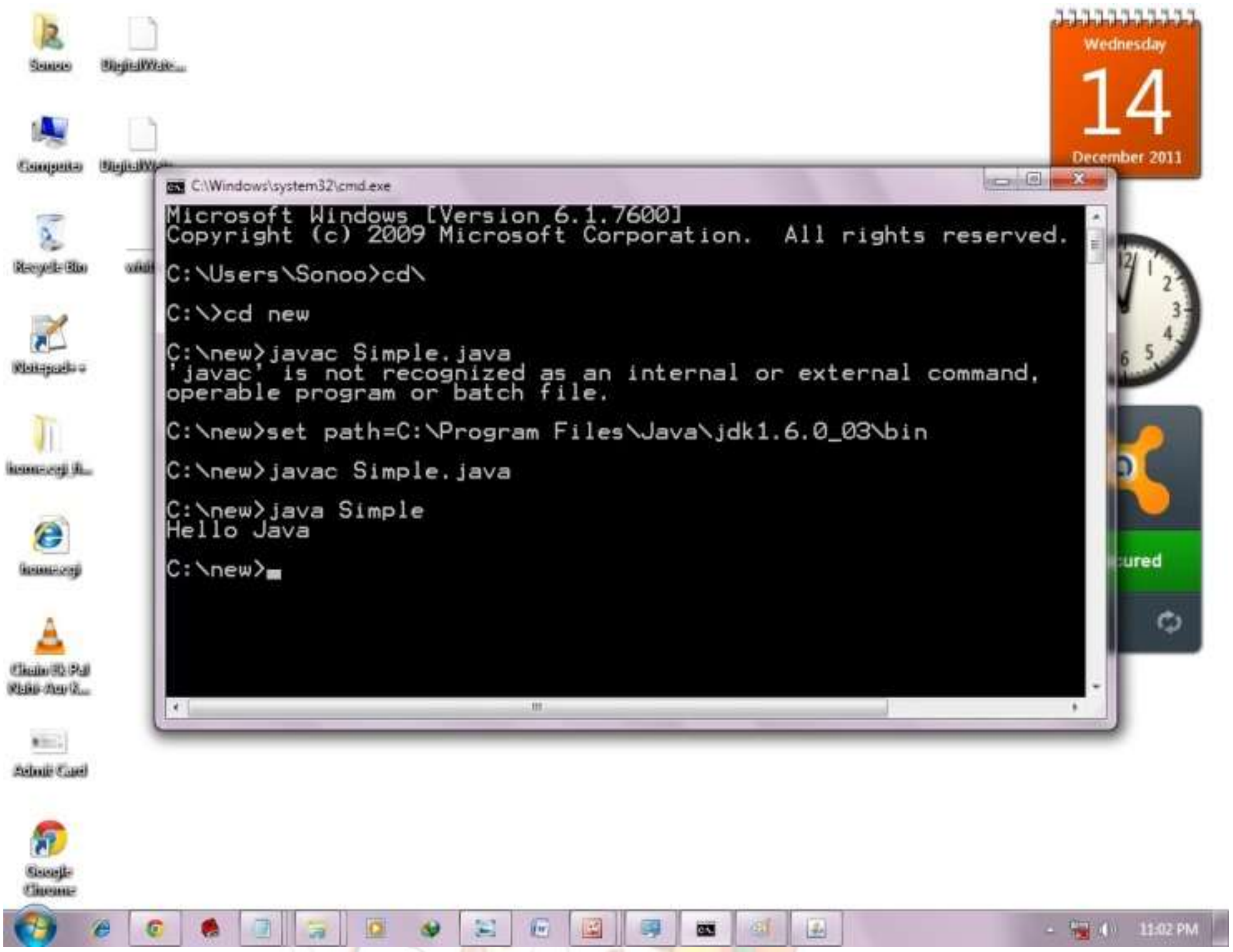
To set the temporary path of JDK, you need to follow following steps:

- Open command prompt
- copy the path of jdk/bin directory
- write in command prompt: set path=copied_path

For Example:

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

Let's see it in the figure given below:



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

C:\Users\Sonoo>cd\

C:\>cd new

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

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

C:\new>javac Simple.java

C:\new>java Simple
Hello Java

C:\new>
```

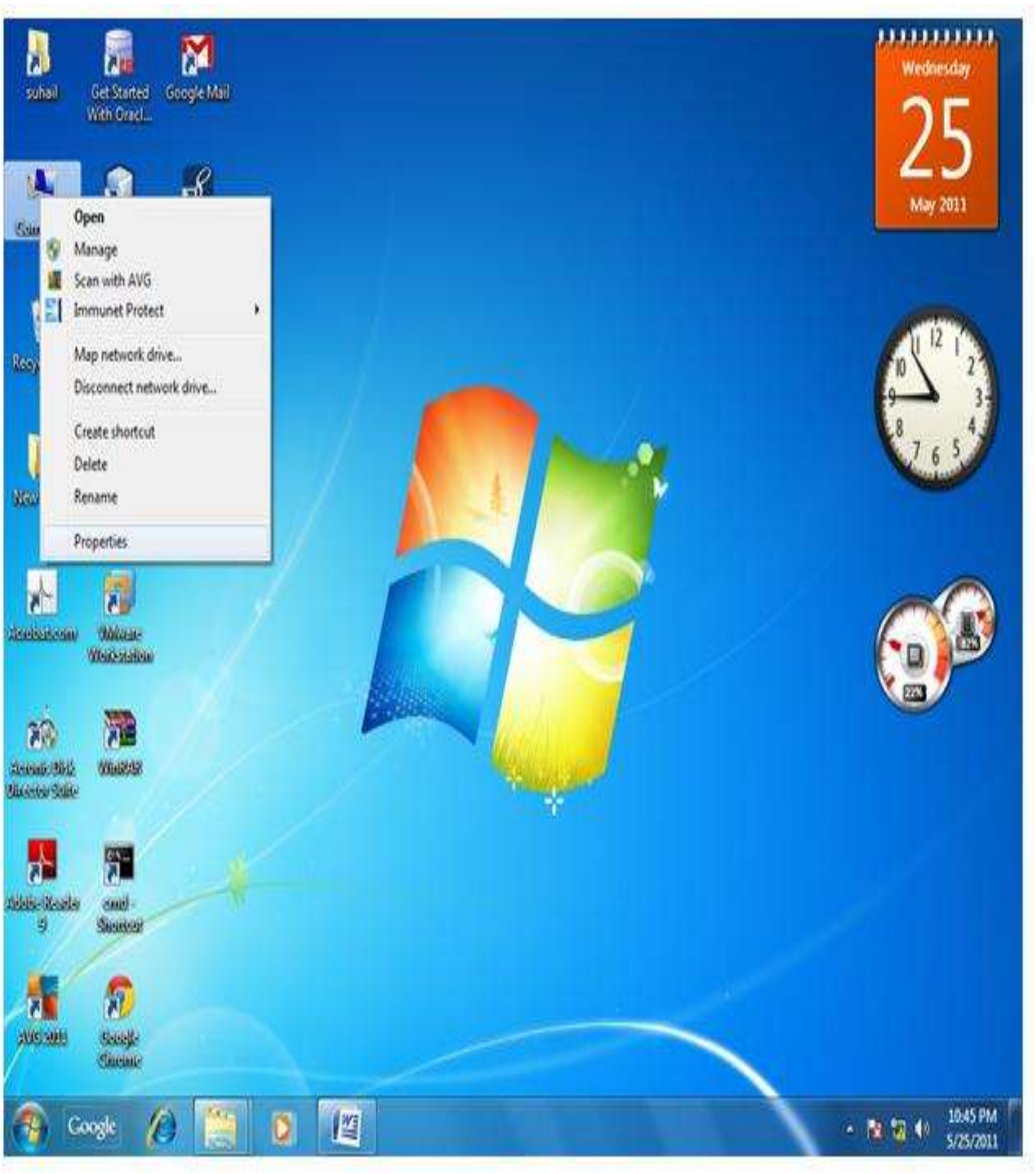
2) How to set Permanent Path of JDK in Windows

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

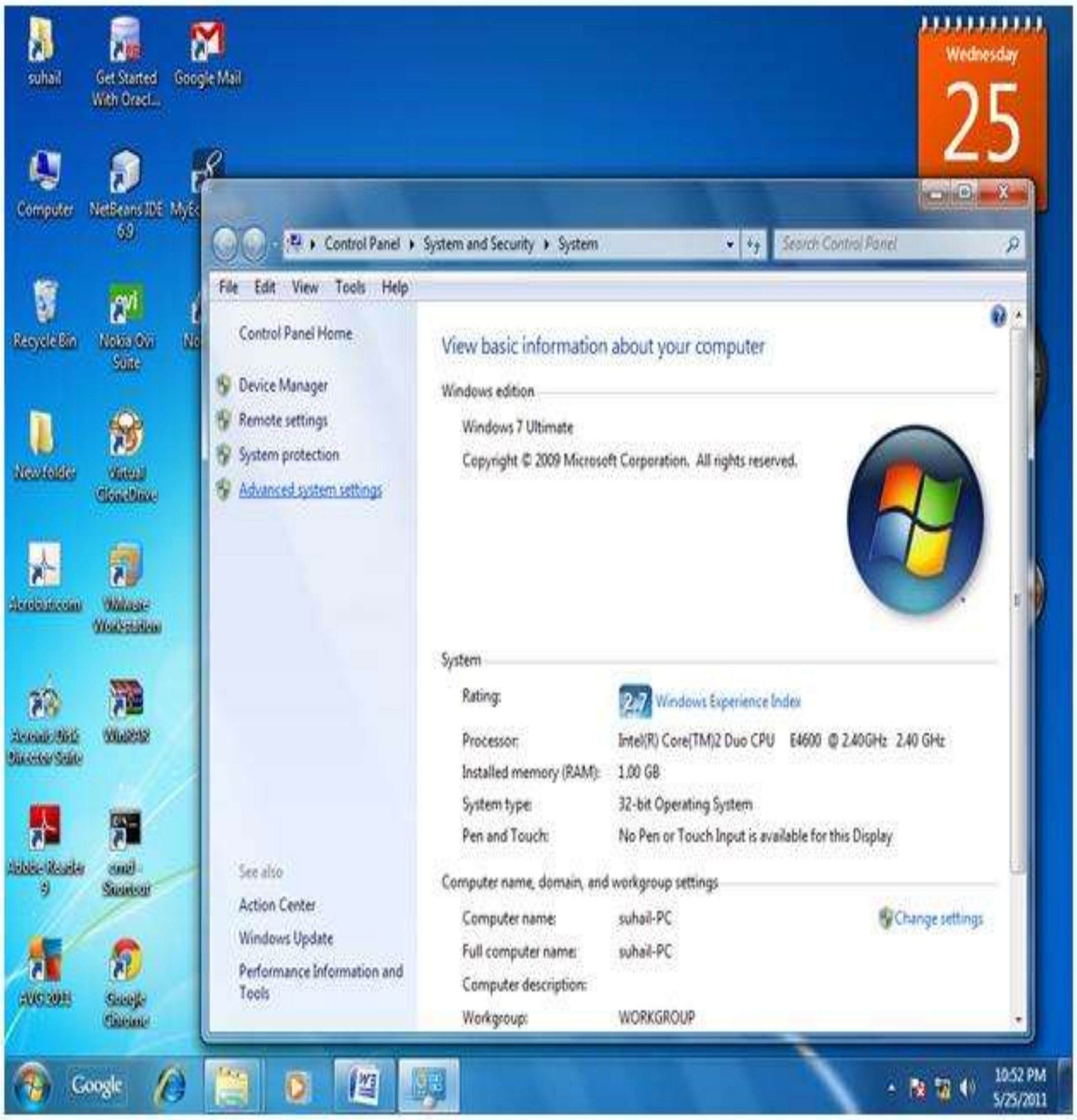
- Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

For Example:

1)Go to MyComputer properties



2)click on advanced tab



View basic information about your computer

Windows edition

Windows 7 Ultimate

Copyright © 2009 Microsoft Corporation. All rights reserved.

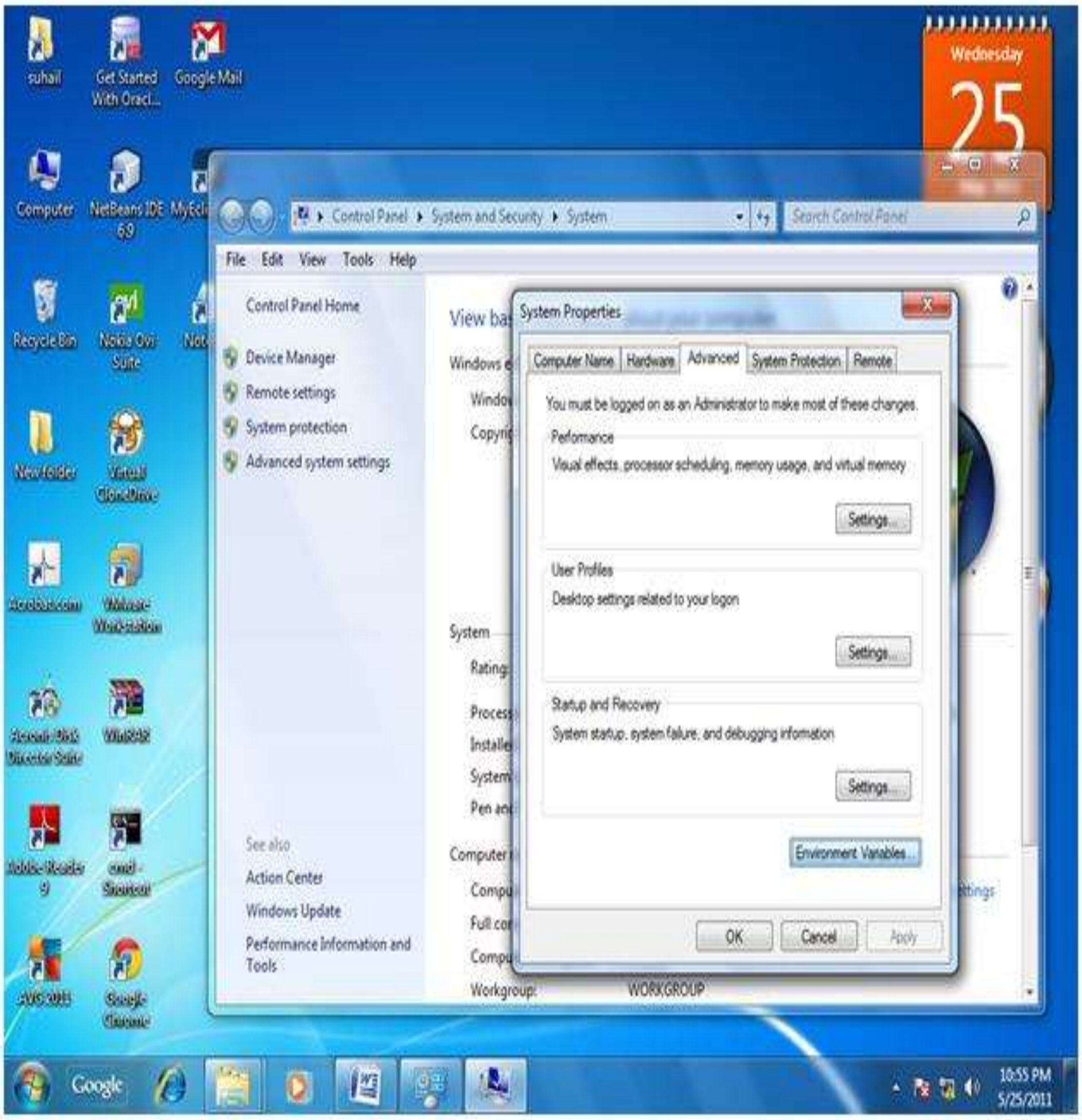
System

Rating:	2.7 Windows Experience Index
Processor:	Intel(R) Core(TM)2 Duo CPU E4600 @ 2.40GHz 2.40 GHz
Installed memory (RAM):	1.00 GB
System type:	32-bit Operating System
Pen and Touch:	No Pen or Touch Input is available for this Display

Computer name, domain, and workgroup settings

Computer name:	suhail-PC
Full computer name:	suhail-PC
Computer description:	
Workgroup:	WORKGROUP

3)click on environment variables



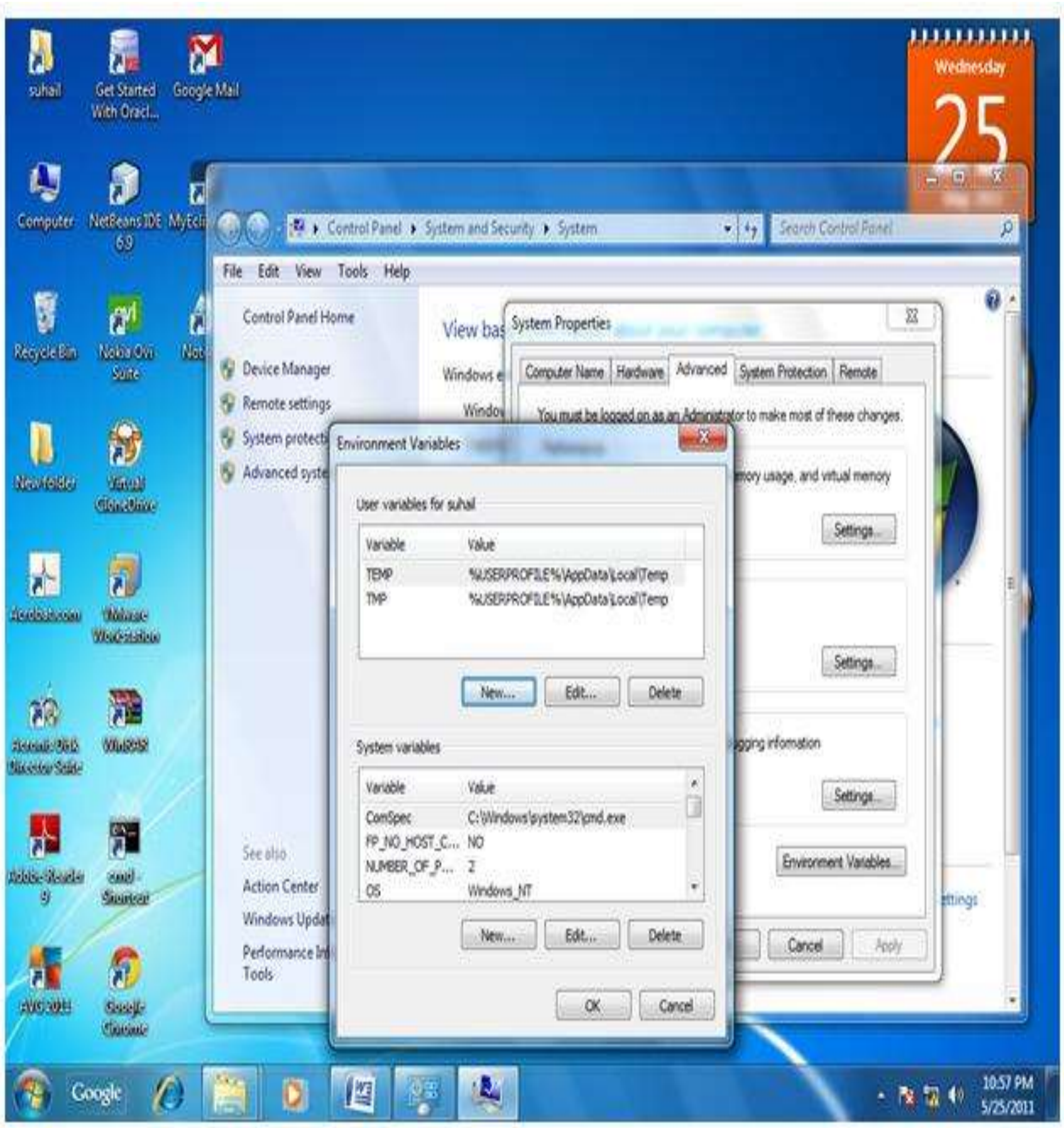
You must be logged on as an Administrator to make most of these changes.

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

User Profiles
Desktop settings related to your logon

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

4)click on new tab of user variables



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

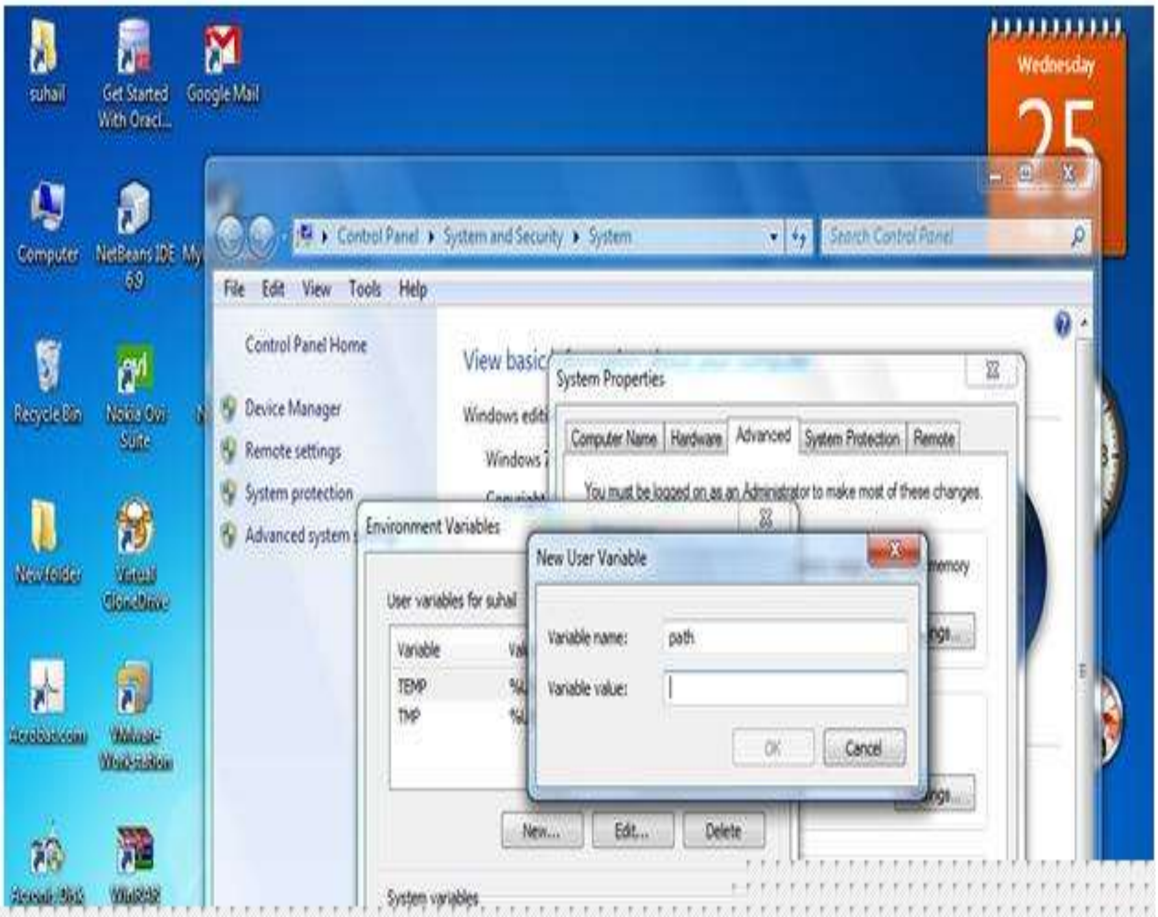
User variables for suhail

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

System variables

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

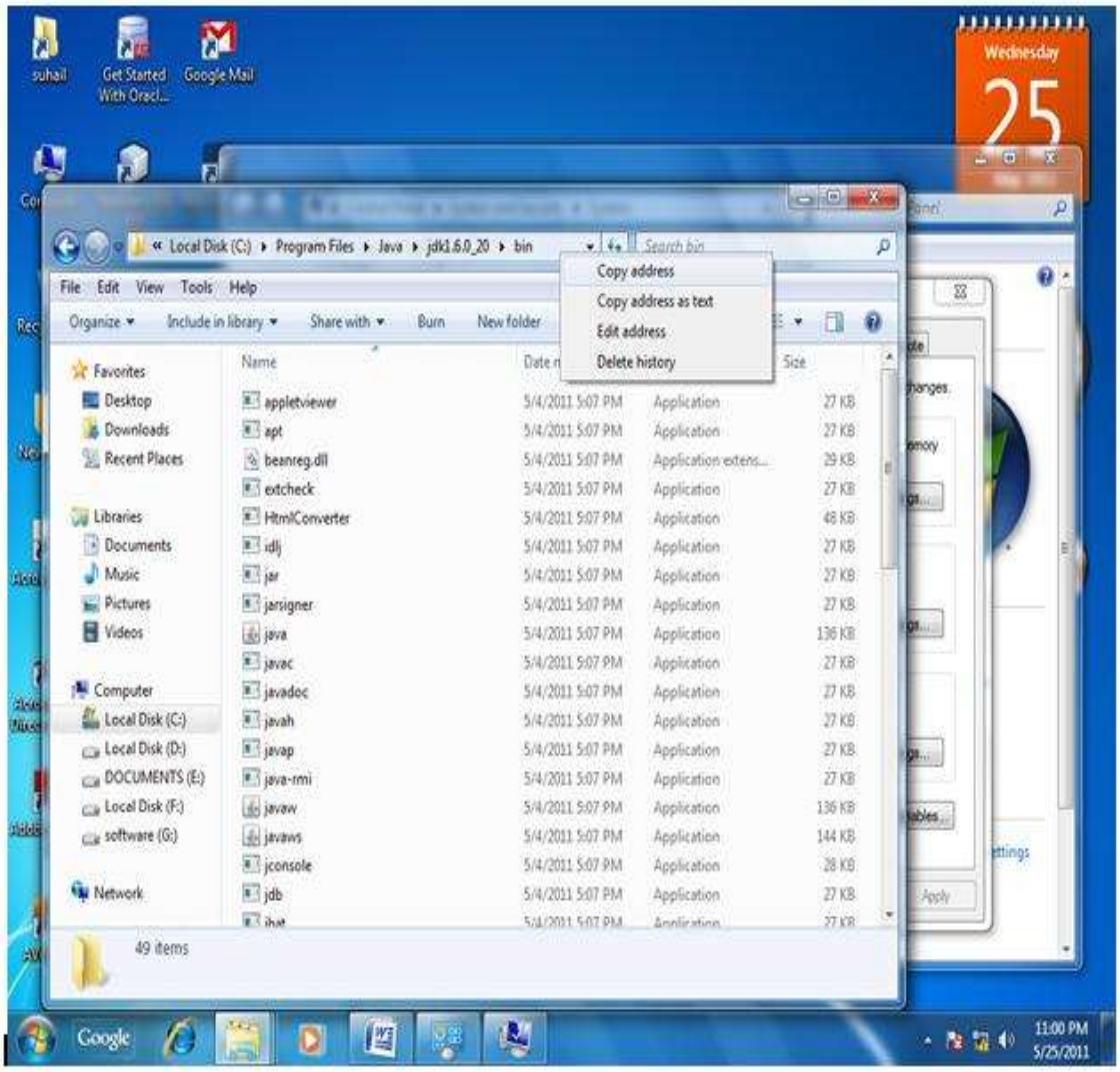
5)write path in variable name



Variable name: path

Variable value:

6)Copy the path of bin folder

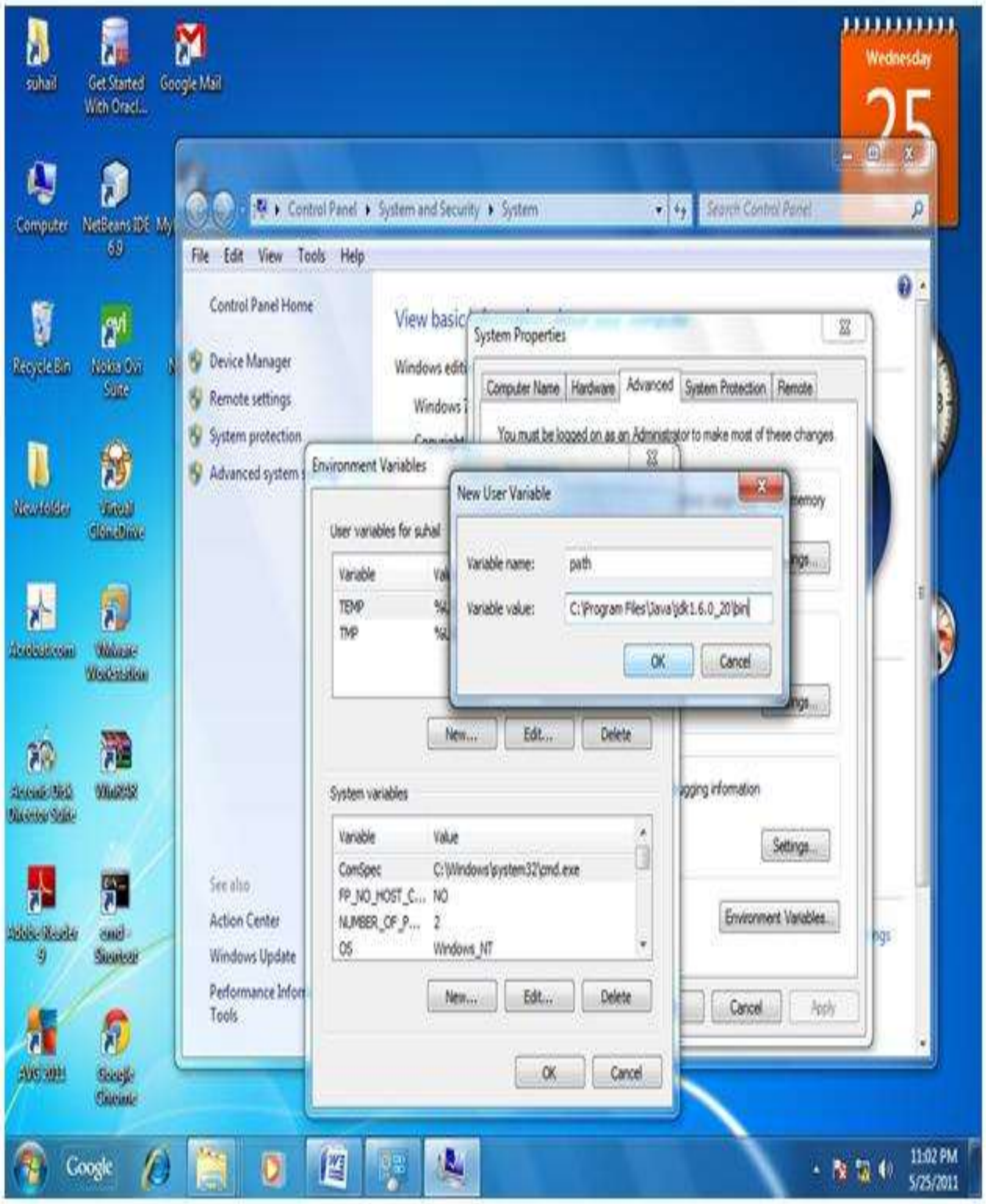


« Local Disk (C:) › Program Files › Java › jdk1.6.0_20 › bin

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

49 items

7)paste path of bin folder in variable value



New User Variable

Variable name: path

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

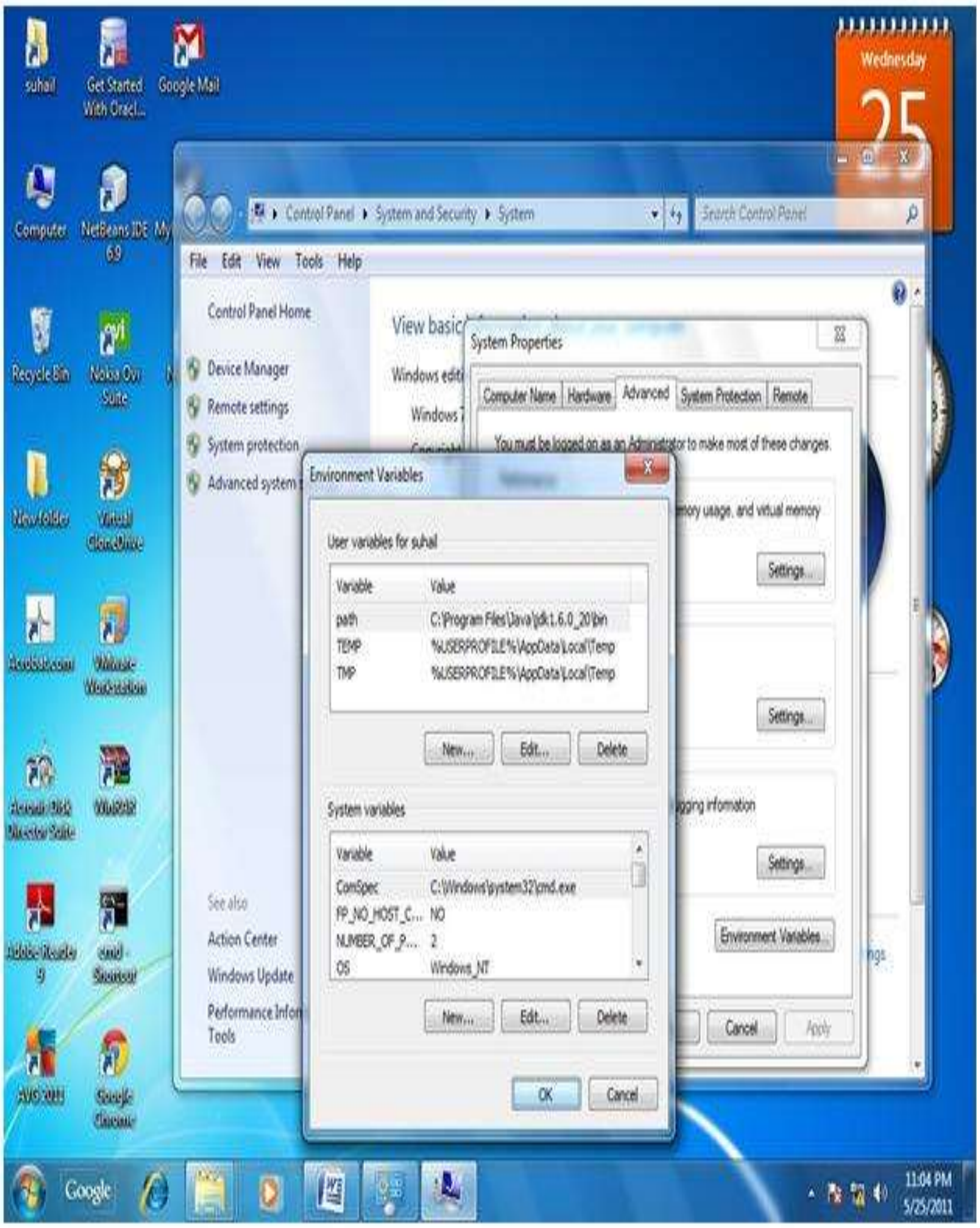
User variables for suhail

Variable	Value
TEMP	%…
TMP	%…

System variables

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

8)click on ok button


Environment Variables
User variables for suhail
New... Edit... Delete
System variables
New... Edit... Delete
OK Cancel
System Properties
Computer Name Hardware Advanced System Protection Remote
Control Panel ▸ System and Security ▸ System

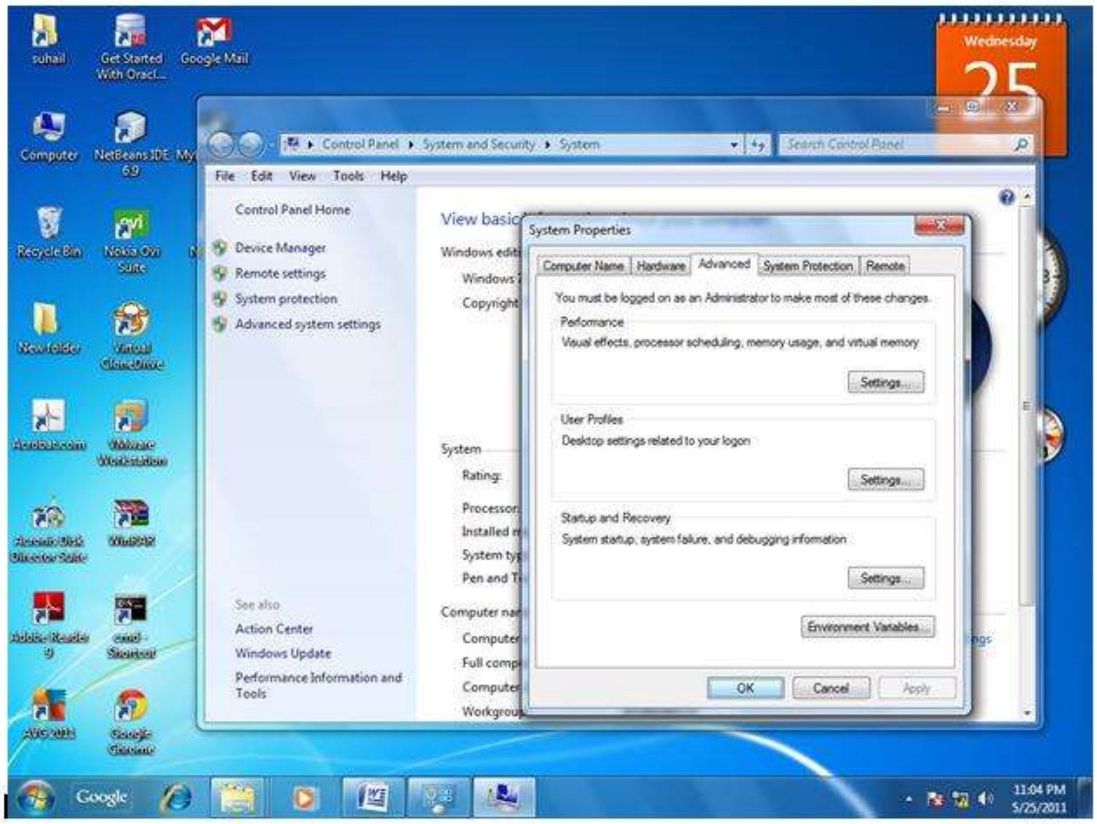
User variables for suhail

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

System variables

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

9)click on ok button



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

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

User Profiles
Desktop settings related to your logon

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

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

Setting Java Path in Linux OS

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

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

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

Difference between JDK, JRE and JVM?

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

Understanding the difference between JDK, JRE and JVM is important in Java. We are having brief overview of JVM here.

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

JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

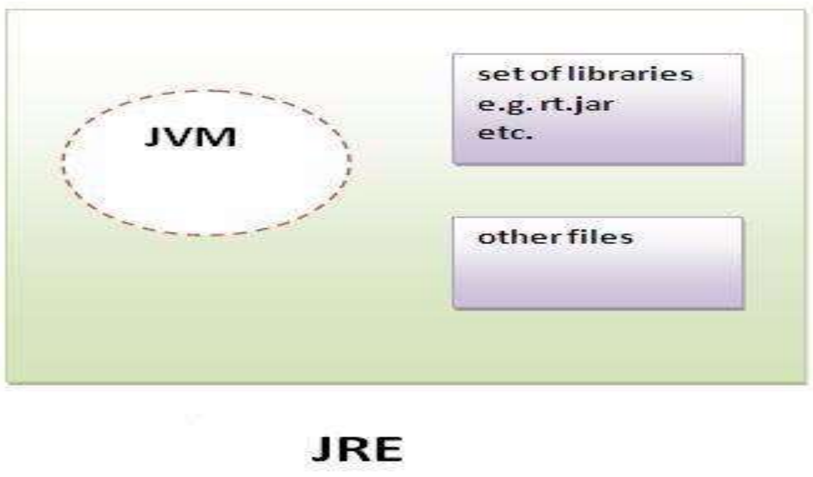
The JVM performs following main tasks:

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

JRE

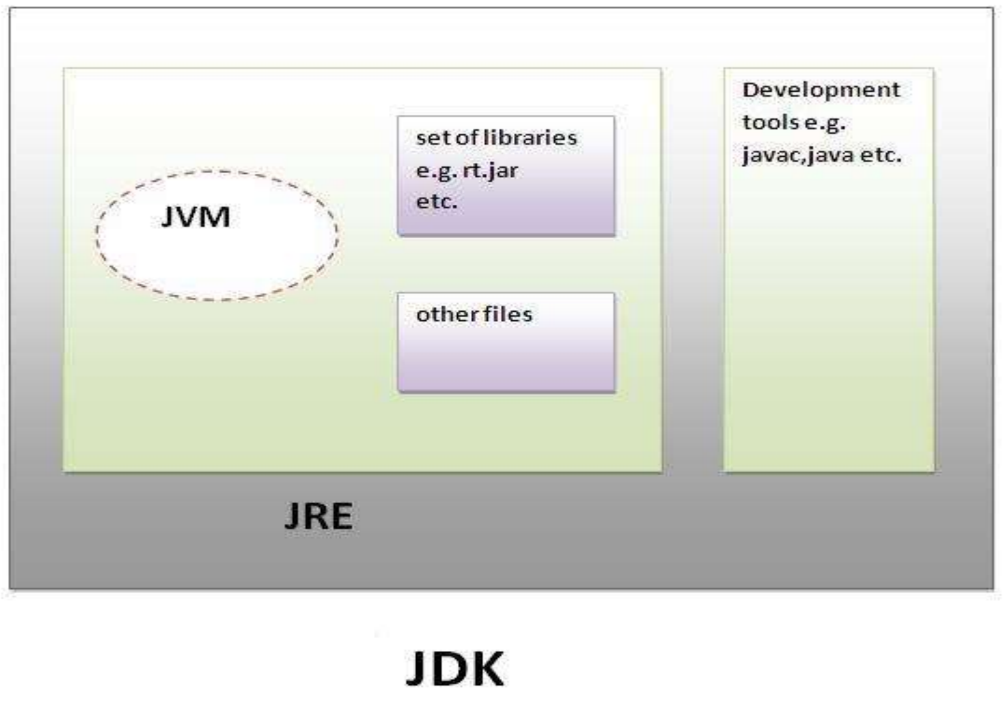
JRE is an acronym for Java Runtime Environment.It is used to provide runtime environment.It is the implementation of JVM.It physically exists.It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



JDK

JDK is an acronym for Java Development Kit.It physically exists.It contains JRE + development tools.


JVM
set of libraries e.g. rt.jar etc.
other files
Development tools e.g. javac,java etc.
JRE
JDK

JVM (Java Virtual Machine)

1. Java Virtual Machine
2. Internal Architecture of JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e.JVM is plateform dependent).

What is JVM?

It is:

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

What it does?

The JVM performs following operation:

- Loads code
- Verifies code

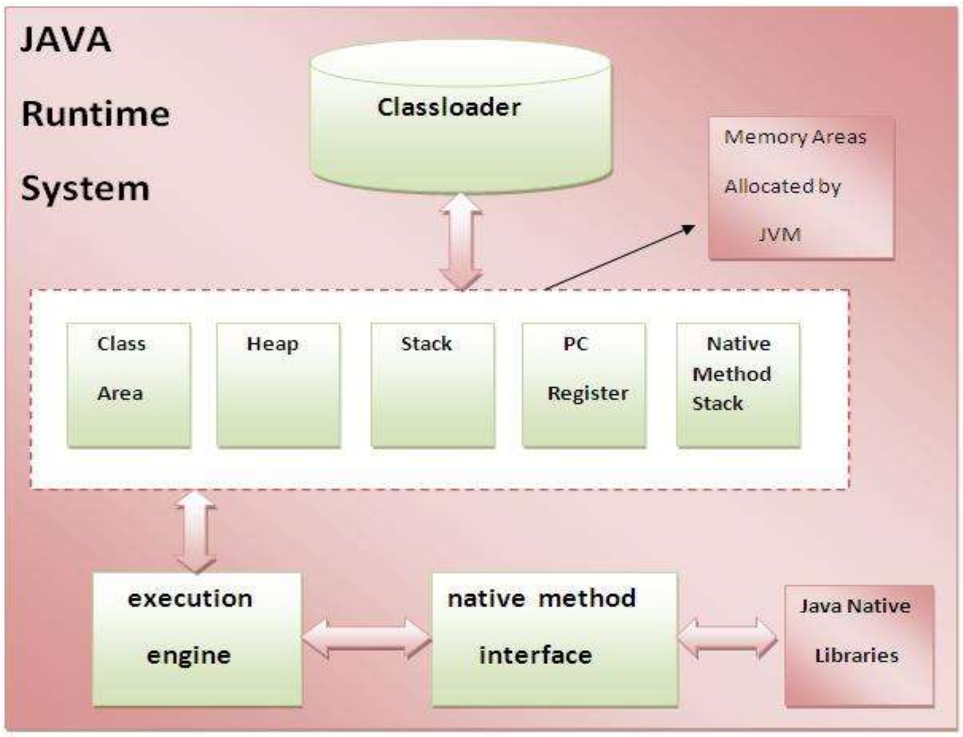
- Executes code
- Provides runtime environment

JVM provides definitions for the:

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

Internal Architecture of JVM

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



1) Classloader:

Classloader is a subsystem of JVM that is used to load class files.

2) Class(Method) Area:

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

3) Heap:

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

4) Stack:

Java Stack stores frames.It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register:

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

6) Native Method Stack:

It contains all the native methods used in the application.

7) Execution Engine:

It contains:

1) A virtual processor

2) Interpreter: Read bytecode stream then execute the instructions.

3) Just-In-Time(JIT) compiler: It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.Here the term ?compiler? refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

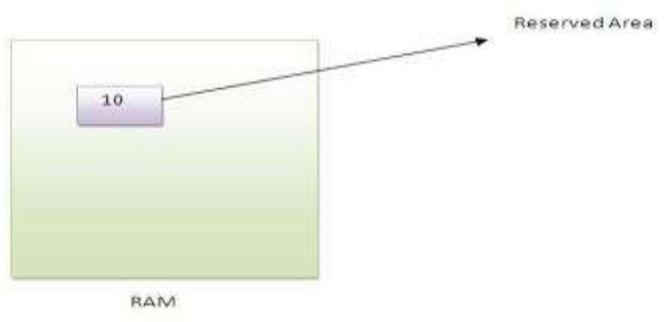
Variable and Datatype in Java

1. Variable
2. Types of Variable
3. Data Types in Java

In this page, we will learn about the variable and java data types. Variable is a name of memory location. There are three types of variables: local, instance and static. There are two types of datatypes in java, primitive and non-primitive.

Variable

Variable is name of reserved area allocated in memory.

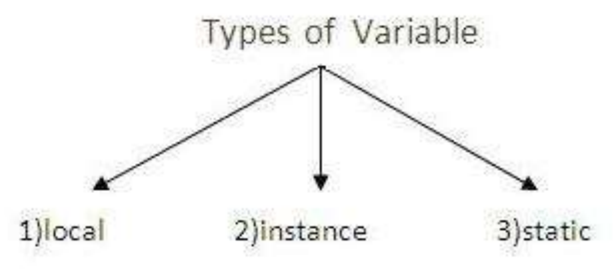


```
1. int data=10;//Here data is variable
```

Types of Variable

There are three types of variables in java

- local variable
- instance variable
- static variable



Local Variable

A variable that is declared inside the method is called local variable.

Instance Variable

A variable that is declared inside the class but outside the method is called instance variable . It is not declared as static.

Static variable

A variable that is declared as static is called static variable. It cannot be local.

We will have detailed learning of these variables in next chapters.

Example to understand the types of variables

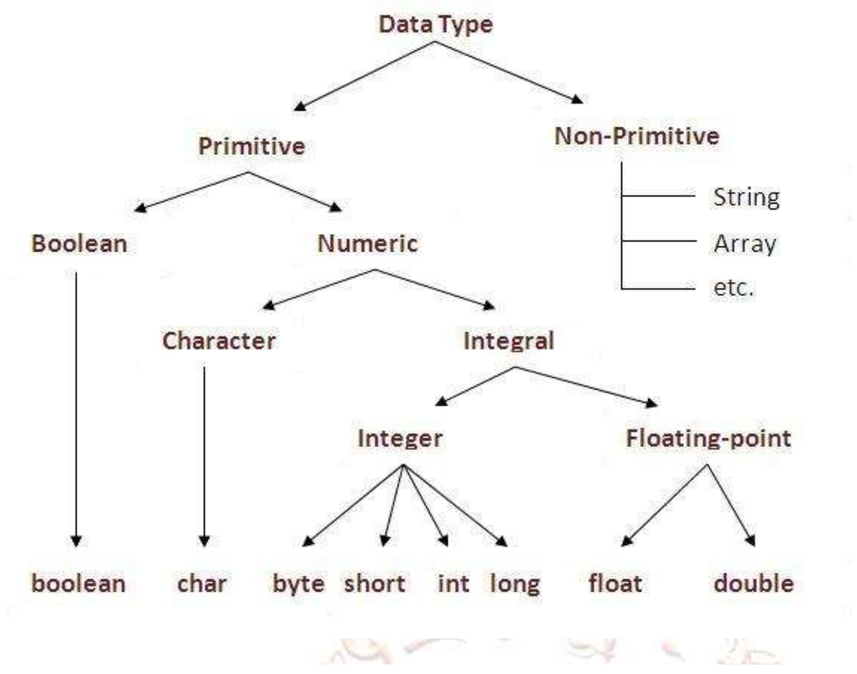
```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```

Data Types in Java

In java, there are two types of data types

- primitive data types

- non-primitive data types



Data Type	Default Value	Default size
Boolean	false	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte
Short	0	2 byte
Int	0	4 byte
Long	0L	8 byte
Float	0.0f	4 byte
Double	0.0d	8 byte

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

because java uses unicode system rather than ASCII code system. \u0000 is the lowest range of unicode system.To get detail about Unicode see below.

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.

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.

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

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

lowest value:\u0000

highest value:\uFFFF

Operators in java:

Operator in java is a symbol that is used to perform operations. There are many types of operators in java such as unary operator, arithmetic operator, relational operator, shift operator, bitwise operator, ternary operator and assignment operator.

Operators	**Precedence**
Postfix	*expr++ expr--*
Unary	*++expr --expr +expr -expr* ~ !
Multiplicative	* / %
Additive	+ -
Shift	<<>>>>>
Relational	<><= >= instanceof
Equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|

logical AND	&&
logical OR	||
Ternary	? :
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Programs:

Java programs are frequently asked in the interview. These programs can be asked from control statements, array, string, oops etc. Let's see the list of java programs.

1) Fibonacci series

Write a java program to print fibonacci series without using recursion and using recursion.

Input: 10

Output: 0 1 1 2 3 5 8 13 21 34

2) Prime number

Write a java program to check prime number.

Input: 44**Output:** not prime number

Input: 7**Output:** prime number

3) Palindrome number

Write a java program to check palindrome number.

Input: 329**Output:** not palindrome number

Input: 12321**Output:** palindrome number

4) Factorial number

Write a java program to print factorial of a number.

Input: 5 **Output:** 120

Input: 6 **Output:** 720

5) Armstrong number

Write a java program to check Armstrong number.

Input: 153 **Output:** Armstrong number

Input: 22 **Output:** not Armstrong number

Java OOPs Concepts

In this page, we will learn about basics of OOPs. Object Oriented Programming is a paradigm that provides many concepts such as**inheritance**, **data binding**, **polymorphism** etc.

Simula is considered as the first object-oriented programming language. The programming paradigm where everything is represented as an object, is known as truly object-oriented programming language.

Smalltalk is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object: Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class: Collection of objects **is called class. It is a logical entity.**

Inheritance: When one object acquires all the properties and behaviours of parent object **i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.**

Polymorphism: When one task is performed by different ways **i.e. known as polymorphism. For example: to convense the customer differently, to draw something e.g. shape or rectangle etc.**

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

Abstraction: Hiding internal details and showing functionality **is known as abstraction. For example: phone call, we don't know the internal processing.**

In java, we use abstract class and interface to achieve abstraction.

Encapsulation: Binding (or wrapping) code and data together into a single unit is known as encapsulation**. For example: capsule, it is wrapped with different medicines.**

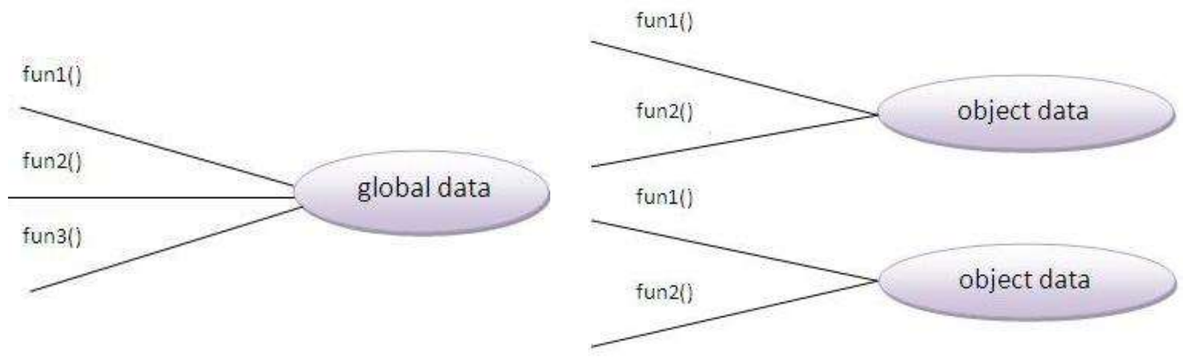
A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Advantage of OOPs over Procedure-oriented programming language

1)OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

2)OOPs provides data hiding whereas in Procedure-oriented prgramming language a global data can be accessed from anywhere.

3)OOPs provides ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.



What is difference between object-oriented programming language and object-based programming language?

Object based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object based programming languages.

What we will learn in OOPs Concepts?

- Advantage of OOPs, Naming Convention, Object and class, Method overloading
- Constructor, static keyword, this keyword with 6 usage, Inheritance, Aggregation
- Method Overriding, Covariant Return Type, super keyword, Instance Initializer block
- final keyword, Abstract class, Interface, Runtime Polymorphism,
- Static and Dynamic Binding, Downcasting with instanceof operator, Package
- Access Modifiers, Encapsulation, Object Cloning

Java Naming conventions:

Java **naming convention** is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc.

But, it is not forced to follow. So, it is known as convention not rule.

All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

Advantage of naming conventions in java:

By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that **less time** is spent to figure out what the code does.

CamelCase in java naming conventions:

Java follows camelcase syntax for naming the class, interface, method and variable.

If name is combined with two words, second word will start with uppercase letter always e.g. actionPerformed(), firstName, ActionEvent, ActionListener etc.

Object and Class in Java

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

In this page, we will learn about java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

Object is the physical as well as logical entity whereas class is the logical entity only.

Object in Java

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tengible and intengible). The example of integible object is banking system.

An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But,it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

Class in Java

A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**
- **class and interface**

Syntax to declare a class:

```
class <class_name>{
    data member;
    method;
}
```

Simple Example of Object and Class

In this example, we have created a Student class that have two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.

```
class Student1{
 int id;//data member (also instance variable)
 String name;//data member(also instance variable)
  public static void main(String args[]){
  Student1 s1=new Student1();//creating an object of Student
 System.out.println(s1.id);
 System.out.println(s1.name);
}
}
```

```
Output:0 null
```

Instance variable in Java

A variable that is created inside the class but outside the method, is known as instance variable.Instance variable doesn't get memory at compile time.It gets memory at runtime when object(instance) is created.That is why, it is known as instance variable.

Method in Java

In java, a method is like function i.e. used to expose behaviour of an object.

Advantage of Method

- Code Reusability
- Code Optimization

new keyword

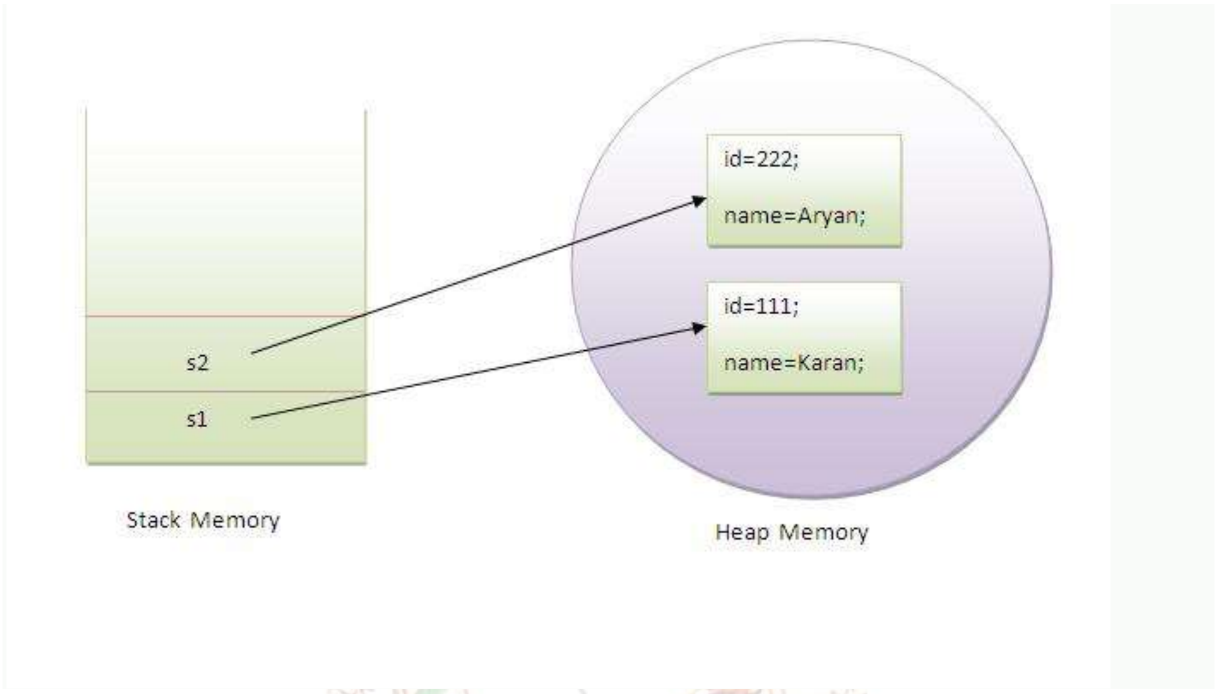
The new keyword is used to allocate memory at runtime.

Example of Object and class that maintains the records of students:

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method on it. Here, we are displaying the state (data) of the objects by invoking the displayInformation method.

```
class Student2{
 int rollno;
 String name;
  void insertRecord(int r, String n){  //method
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}//method
 public static void main(String args[]){
  Student2 s1=new Student2();
  Student2 s2=new Student2();
 s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 } }
```

Output:111 Kara   222 Aryan



As you see in the above figure, object gets the memory in Heap area and reference variable refers to the object allocated in the Heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

Another Example of Object and Class

There is given another example that maintains the records of Rectangle class. Its exaplanation is same as in the above Student class example.

```
class Rectangle{
 int length;
 int width;
 void insert(int l,int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
} }
```

```
Output:55      45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By factory () method etc.

We will learn, these ways to create the object later.

Annonymous object

Annonymous simply means nameless.An object that have no reference is known as annonymous object.

If you have to use an object only once, annonymous object is a good approach.

```
class Calculation{
 void fact(int  n){
 int fact=1;
 for(int i=1;i<=n;i++){
  fact=fact*i;
 }
 System.out.println("factorial is "+fact);
}

public static void main(String args[]){
 new Calculation().fact(5);//calling method with annonymous object
}
}
```

```
Output:Factorial is 120
```

Creating multiple objects by one type only:

We can create multiple objects by one type only as we do in case of primitives.

```
Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
```

Let's see the example:

```
class Rectangle{
 int length;
 int width;
   void insert(int l,int w){
  length=l;
  width=w;
 }
   void calculateArea(){System.out.println(length*width);}
   public static void main(String args[]){
  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
  r1.insert(11,5);
  r2.insert(3,15);
   r1.calculateArea();
  r2.calculateArea();
} }
```

```
Output:55
       45
```

Method Overloading in Java

1. Different ways to overload the method
2. By changing the no. of arguments
3. By changing the datatype
4. Why method overloading is not possible by changing the return type
5. Can we overload the main method
6. method overloading with Type Promotion

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs. So, we perform method overloading to figure out the program quickly.

Advantage of method overloading?

Method overloading **increases the readability of the program**.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In java, Methood Overloading is not possible by changing the return type of the method.

---

1)Example of Method Overloading by changing the no. of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

```
class Calculation{
  void sum(int a,int b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}
  public static void main(String args[]){
  Calculation obj=new Calculation();
  obj.sum(10,10,10);
  obj.sum(20,20);
 } }
```

```
Output:30      40
```

---

2)Example of Method Overloading by changing data type of argument

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two double arguments.

```
class Calculation2{
  void sum(int a,int b){System.out.println(a+b);}
  void sum(double a,double b){System.out.println(a+b);}
  public static void main(String args[]){
  Calculation2 obj=new Calculation2();
  obj.sum(10.5,10.5);
  obj.sum(20,20);
  }}
```

```
Output:21.0     40
```

Que) Why Method Overloaing is not possible by changing the return type of method?

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity. Let's see how ambiguity may occur:

because there was problem:

```
class Calculation3{
  int sum(int a,int b){System.out.println(a+b);}
  double sum(int a,int b){System.out.println(a+b);}
  public static void main(String args[]){
  Calculation3 obj=new Calculation3();
  int result=obj.sum(20,20); //Compile Time Error
  } }
```

int result=obj.sum(20,20); //Here how can java determine which sum() method should be called

---

Can we overload main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. Let's see the simple example:

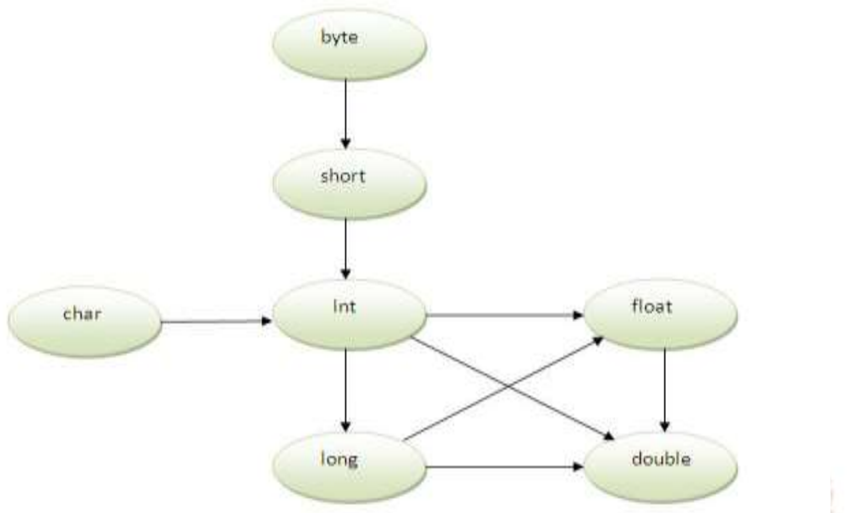
```
class Overloading1{
  public static void main(int a){
  System.out.println(a);
  }
  public static void main(String args[]){
  System.out.println("main() method invoked");
  main(10);
  }
}
```

```
Output:main() method invoked

     10
```

Method Overloading and TypePromotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```java
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}
   public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalculation1();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);
  } }
```

```
Output:40     60
```

Example of Method Overloading with TypePromotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```java
class OverloadingCalculation2{
  void sum(int a,int b){System.out.println("int arg method invoked");}
  void sum(long a,long b){System.out.println("long arg method invoked");}
  public static void main(String args[]){
  OverloadingCalculation2 obj=new OverloadingCalculation2();
  obj.sum(20,20);//now int arg sum() method gets invoked
  } }
```

```
Output:int arg method invoked
```

Example of Method Overloading with TypePromotion in case ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
class OverloadingCalculation3{
  void sum(int a,long b){System.out.println("a method invoked");}
  void sum(long a,int b){System.out.println("b method invoked");}
   public static void main(String args[]){
  OverloadingCalculation3 obj=new OverloadingCalculation3();
  obj.sum(20,20);//now ambiguity
  } }
```

```
Output:Compile Time Error
```

Constructor in Java

Constructor in java is a *special type of method* that is used to initialize the object.

Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

There are basically two rules defined for the constructor.

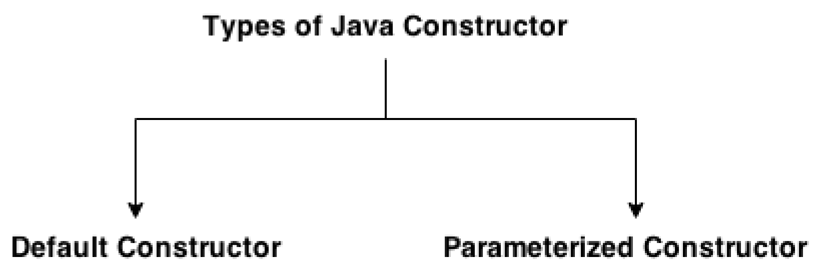
1. Constructor name must be same as its class name

2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor that have no parameter is known as default constructor.

Syntax of default constructor:

```
<class_name>(){}
```

Example of default constructor

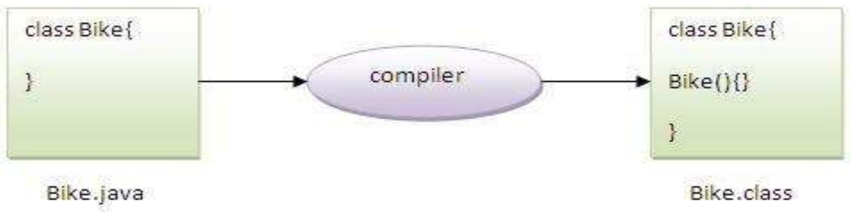
In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
} }
```

Output:

```
Bike is created
```

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



Q) What is the purpose of default constructor?

Default constructor provides the default values to the object like 0, null etc. depending on the type.

Example of default constructor that displays the default values

```
class Student3{

int id;

String name;


void display(){System.out.println(id+" "+name);}


public static void main(String args[]){

Student3 s1=new Student3();

Student3 s2=new Student3();

s1.display();
s2.display();
} }
```

Output:

```
0 null  0 null
```

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor.Here 0 and null values are provided by default constructor.

Java parameterized constructor

A constructor that have parameters is known as parameterized constructor.

Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
class Student4{
int id;
String name;
Student4(int i,String n){
id = i;
name = n;
}
void display(){System.out.println(id+" "+name);}
public static void main(String args[]){
Student4 s1 = new Student4(111,"Karan");
Student4 s2 = new Student4(222,"Aryan");
s1.display();
s2.display();
} }
```

Output:

```
111 Karan     222 Aryan
```

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
class Student5{
    int id;
    String name;
    int age;
    Student5(int i,String n){
    id = i;
    name = n;
    }
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}
      public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   } }
```

Output:

```
111 Karan 0

222 Aryan 25
```

Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name

Java Copy Constructor

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

```
class Student6{
    int id;
    String name;
    Student6(int i,String n){
    id = i;
    name = n;
    }

    Student6(Student6 s){
    id = s.id;
```

```
   name =s.name;
   }
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   Student6 s1 = new Student6(111,"Karan");
   Student6 s2 = new Student6(s1);
   s1.display();
   s2.display();
  }
}
```

Output:

```
111 Karan
111 Karan
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
class Student7{
   int id;
   String name;
   Student7(int i,String n){
   id = i;
   name = n;
   }
   Student7(){}
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   Student7 s1 = new Student7(111,"Karan");
   Student7 s2 = new Student7();
   s2.id=s1.id;
   s2.name=s1.name;
   s1.display();
17.    s2.display();
18.   } }
```

Output:

```
111 Karan  111 Karan
```

Q) Does constructor return any value?

Ans:yes, that is current class instance (You cannot use return type yet it returns a value).

Q) Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

Java static keyword

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

---

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program **memory efficient** (i.e it saves memory).

Understanding problem without static variable

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

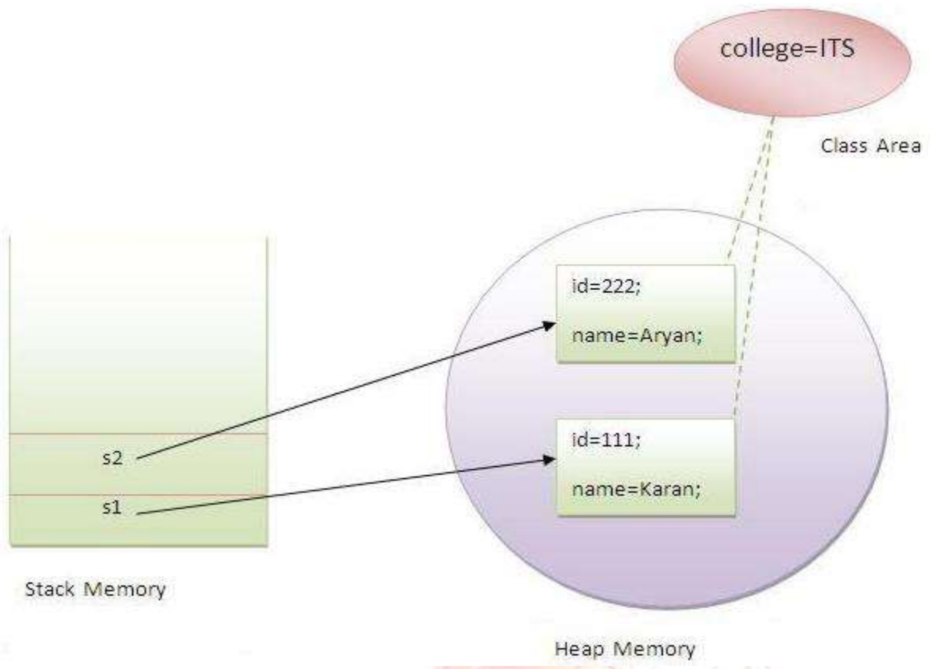
Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.All student have its unique rollno and name so instance data member is good.Here, college refers to the common property of all objects.If we make it static,this field will get memory only once.

Java static property is shared to all objects.

Example of static variable

```
//Program of static variable
 class Student8{
  int rollno;
  String name;
  static String college ="ITS";
     Student8(int r,String n){
  rollno = r;
  name = n;
  }
 void display (){System.out.println(rollno+" "+name+" "+college);}
  public static void main(String args[]){
 Student8 s1 = new Student8(111,"Karan");
 Student8 s2 = new Student8(222,"Aryan");
   s1.display();
 s2.display();
 }
}
```

```
Output:111 Karan ITS
       222 Aryan ITS
```



Program of counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the count variable.

```
class Counter{
int count=0;//will get memory when instance is created
  Counter(){
count++;
System.out.println(count);
}
  public static void main(String args[]){
 Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
 } }
```

```
Output:1
       1
       1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
class Counter2{
static int count=0;//will get memory only once and retain its value
 Counter2(){
count++;
System.out.println(count);
}
  public static void main(String args[]){
  Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
 } }
```

```
Output:1      2      3
```

---

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

```
//Program of changing the common property of all objects(static field).
class Student9{
     int rollno;
     String name;
     static String college = "ITS";
     static void change(){
     college = "BBDIT";
     }
     Student9(int r, String n){
     rollno = r;
     name = n;
     }
     void display (){System.out.println(rollno+" "+name+" "+college);}
    public static void main(String args[]){
  Student9.change();
    Student9 s1 = new Student9 (111,"Karan");
  Student9 s2 = new Student9 (222,"Aryan");
  Student9 s3 = new Student9 (333,"Sonoo");
    s1.display();
  s2.display();
  s3.display();
  } }
```

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Another example of static method that performs normal calculation

```
//Program to get cube of a given number by static method
class Calculate{
  static int cube(int x){
  return x*x*x;
  }      public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);    } }
```

```
Output:125
```

Restrictions for static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
class A{   int a=40;//non static
   public static void main(String args[]){
  System.out.println(a);
 } }
```

```
Output:Compile Time Error
```

Q) why java main method is static?

Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

---

3) Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block

```java
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

```
Output:static block is invoked
       Hello main
```

Q) Can we execute a program without main() method?

Ans) Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

```java
class A3{
  static{
  System.out.println("static block is invoked");
  System.exit(0);
  }
}
```

```
Output:static block is invoked (if not JDK7)
```

In JDK7 and above, output will be:

```
Output:Error: Main method not found in class A3, please define the main method as:
public static void main(String[] args)
```

this keyword in java

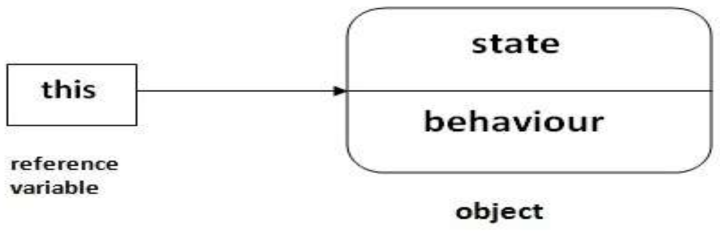
There can be a lot of usage of **java this keyword**. In java, this is a**reference variable** that refers to the current object.

Usage of java this keyword:

Here is given the 6 usage of java this keyword.

1. this keyword can be used to refer current class instance variable.
2. this() can be used to invoke current class constructor.
3. this keyword can be used to invoke current class method (implicitly)
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this keyword can also be used to return the current class instance.

Suggestion: If you are beginner to java, lookup only two usage of this keyword.



1) The this keyword can be used to refer current class instance variable.

If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
class Student10{
    int id;
    String name;
      Student10(int id,String name){
    id = id;      name = name;      }
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
    Student10 s1 = new Student10(111,"Karan");
    Student10 s2 = new Student10(321,"Aryan");
    s1.display();      s2.display();      } }
```

```
Output:0 null       0 null
```

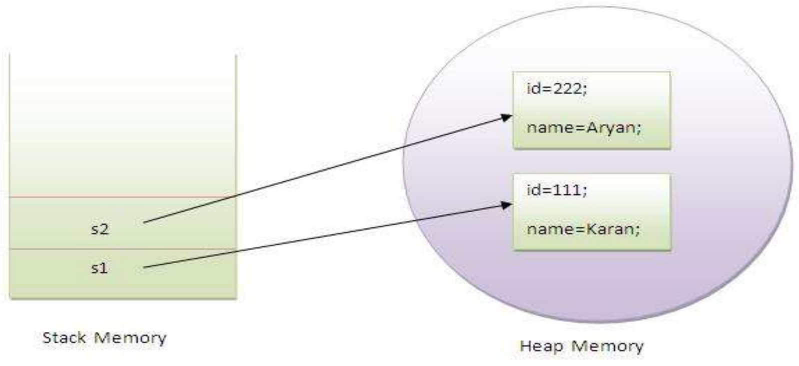
In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

```
//example of this keyword
class Student11{      int id;      String name;
    Student11(int id,String name){      this.id = id;      this.name = name;      }
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
    Student11 s1 = new Student11(111,"Karan");
    Student11 s2 = new Student11(222,"Aryan");
    s1.display();      s2.display();
} }
```

```
Output111 Karan

     222 Aryan
```



If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

```
class Student12{      int id;      String name;
Student12(int i,String n){      id = i;      name = n;      }
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
    Student12 e1 = new Student12(111,"karan");
    Student12 e2 = new Student12(222,"Aryan");
    e1.display();      e2.display();  }  }
```

```
Output:111 Karan       222 Aryan
```

2) this() can be used to invoked current class constructor.

The this() constructor call can be used to invoke the current class constructor (constructor chaining). This approach is better if you have many constructors in the class and want to reuse that constructor.

```
//Program of this() constructor call (constructor chaining)
  class Student13{      int id;      String name;
    Student13(){System.out.println("default constructor is invoked");}
    Student13(int id,String name){
    this ();//it is used to invoked current class constructor.
    this.id = id;
    this.name = name;
    }
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
    Student13 e1 = new Student13(111,"karan");
    Student13 e2 = new Student13(222,"Aryan");
    e1.display();      e2.display();
  } }
```

```
Output:
       default constructor is invoked
       default constructor is invoked
       111 Karan       222 Aryan
```

Where to use this() constructor call?

The this() constructor call should be used to reuse the constructor in the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
class Student14{     int id;     String name;     String city;
   Student14(int id,String name){     this.id = id;     this.name = name;     }
   Student14(int id,String name,String city){
   this(id,name);//now no need to initialize id and name
   this.city=city;     }
   void display(){System.out.println(id+" "+name+" "+city);}
   public static void main(String args[]){
   Student14 e1 = new Student14(111,"karan");
   Student14 e2 = new Student14(222,"Aryan","delhi");
   e1.display();     e2.display();     } }
```

```
Output:111 Karan null     222 Aryan delhi
```

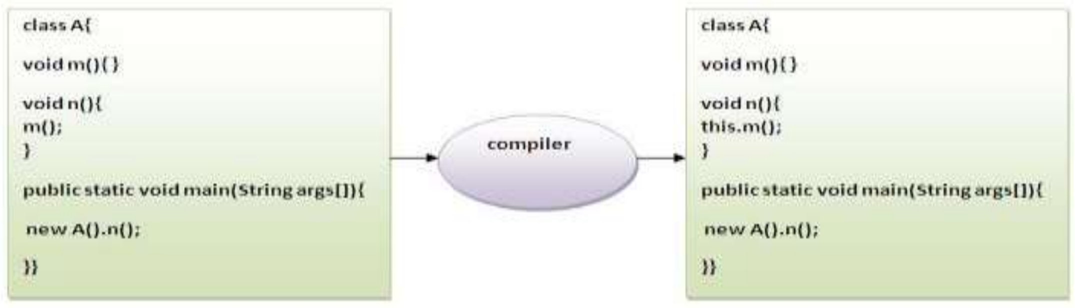
Rule: Call to this() must be the first statement in constructor.

```
class Student15{     int id;     String name;
   Student15(){System.out.println("default constructor is invoked");}
   Student15(int id,String name){     id = id;     name = name;
   this ();//must be the first statement     }
   void display(){System.out.println(id+" "+name);}
   public static void main(String args[]){
   Student15 e1 = new Student15(111,"karan");
   Student15 e2 = new Student15(222,"Aryan");
   e1.display();     e2.display();     } }
```

```
Output:Compile Time Error
```

3)The this keyword can be used to invoke current class method (implicitly).

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



```
class A{
void m(){}
void n(){
m();
}
public static void main(String args[]){
new A().n();
}}
```

```
class A{
void m(){}
void n(){
this.m();
}
public static void main(String args[]){
new A().n();
}}
```

```
class S{
  void m(){    System.out.println("method is invoked");    }
  void n(){    this.m();//no need because compiler does it for you.    }
  void p(){    n();//complier will add this to invoke n() method as this.n()    }
  public static void main(String args[]){    S s1 = new S();   s1.p();    }  }
```

```
Output:method is invoked
```

4) this keyword can be passed as an argument in the method.

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class S2{
  void m(S2 obj){    System.out.println("method is invoked");    }
  void p(){    m(this);    }
      public static void main(String args[]){    S2 s1 = new S2();    s1.p();    }  }
```

```
Output:method is invoked
```

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one.

5) The this keyword can be passed as argument in the constructor call.

We can pass this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
class B{    A4 obj;    B(A4 obj){      this.obj=obj;    }
  void display(){      System.out.println(obj.data);//using data member of A4 class    }  }
  class A4{    int data=10;    A4(){     B b=new B(this);     b.display();    }
  public static void main(String args[]){     A4 a=new A4();    }
}
```

```
Output:10
```

6) The this keyword can be used to return current class instance.

We can return this keyword as statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
1. return_type method_name(){  return this;  }
```

Example of this keyword that you return as a statement from the method

```
1. class A{  A getA(){  return this;  }
2. void msg(){System.out.println("Hello java");}  }
3. class Test1{  public static void main(String args[]){  new A().getA().msg();  } }
```

```
Output:Hello java
```

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

```
1. class A5{  void m(){  System.out.println(this);//prints same reference ID  }
2.   public static void main(String args[]){  A5 obj=new A5();
3. System.out.println(obj);//prints the reference ID
4.   obj.m();  }  }
```

```
Output:A5@22b3ea59
       A5@22b3ea59
```

Inheritance in Java

1. Inheritance, Types of Inheritance
2. Why multiple inheritance is not possible in java in case of class?

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Syntax of Java Inheritance

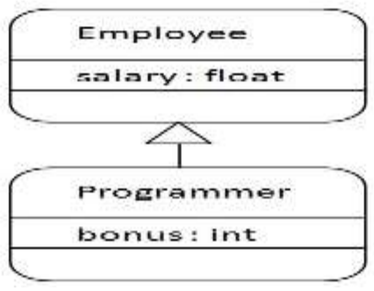
```
1. class Subclass-name extends Superclass-name
2. {
```

```
3.     //methods and fields
4.  }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class.

In the terminology of Java, a class that is inherited is called a super class. The new class is called a subclass.

Understanding the simple example of inheritance



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is **Programmer IS-A Employee**.It means that Programmer is a type of Employee.

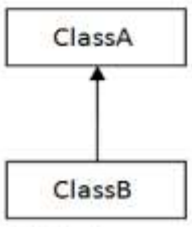
```
1.  class Employee{   float salary=40000;  }
2.  class Programmer extends Employee{   int bonus=10000;
3.   public static void main(String args[]){
4.     Programmer p=new Programmer();
5.     System.out.println("Programmer salary is:"+p.salary);
6.     System.out.println("Bonus of Programmer is:"+p.bonus);
7.  } }
```

```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

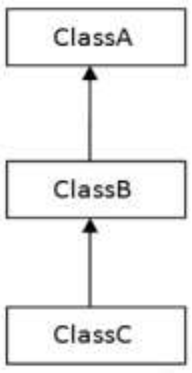
In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

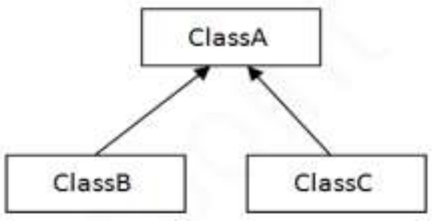
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



1) Single



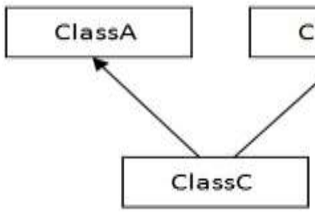
2) Multilevel



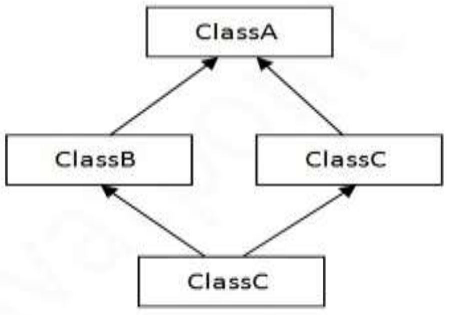
3) Hierarchical

Note: Multiple inheritance is not supported in java through class.

When a class extends multiple classes i.e. known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

.

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
class A{  void msg(){System.out.println("Hello");}  }
class B{  void msg(){System.out.println("Welcome");}  }
class C extends A,B{//suppose if it were
Public Static void main(String args[]){
  C obj=new C();     obj.msg();//Now which msg() method would be invoked?
}}
```

Compile Time Error

Aggregation in Java:

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

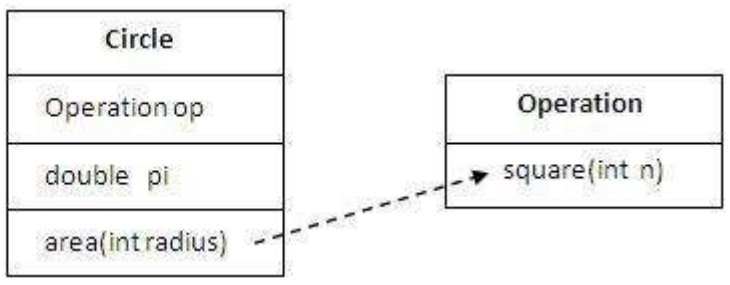
```
class Employee{  int id;  String name;  Address address;//Address is a class
...
}
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

- For Code Reusability.

Simple Example of Aggregation



In this example, we have created the reference of Operation class in the Circle class.

```
class Operation{
 int square(int n){    return n*n;   }  }
  class Circle{   Operation op;//aggregation   double pi=3.14;
     double area(int radius){     op=new Operation();
   int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
   return pi*rsquare;   }
  public static void main(String args[]){
   Circle c=new Circle();
   double result=c.area(5);
   System.out.println(result);
 } }
```

```
Output:78.5
```

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Understanding meaningful example of Aggregation

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

Address.java

```java
public class Address {  String city,state,country;
public Address(String city, String state, String country) {
    this.city = city;      this.state = state;      this.country = country;  }}
```

Emp.java

```java
public class Emp {  int id;  String name;  Address address;
public Emp(int id, String name,Address address) {
    this.id = id;      this.name = name;      this.address=address;
}
void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}
public static void main(String[] args) {
Address address1=new Address("gzb","UP","india");
Address address2=new Address("gno","UP","india");
Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);
e.display();  e2.display();  }}
```

```
Output:111 varun      gzb UP india      112 arun      gno UP india
```

Method Overriding in Java

1. Understanding problem without method overriding
2. Can we override the static method
3. method overloading vs method overriding

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
class Vehicle{    void run(){System.out.println("Vehicle is running");}  }
class Bike extends Vehicle{   public static void main(String args[]){    Bike obj = new Bike();
  obj.run();    }}
```

```
Output:Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

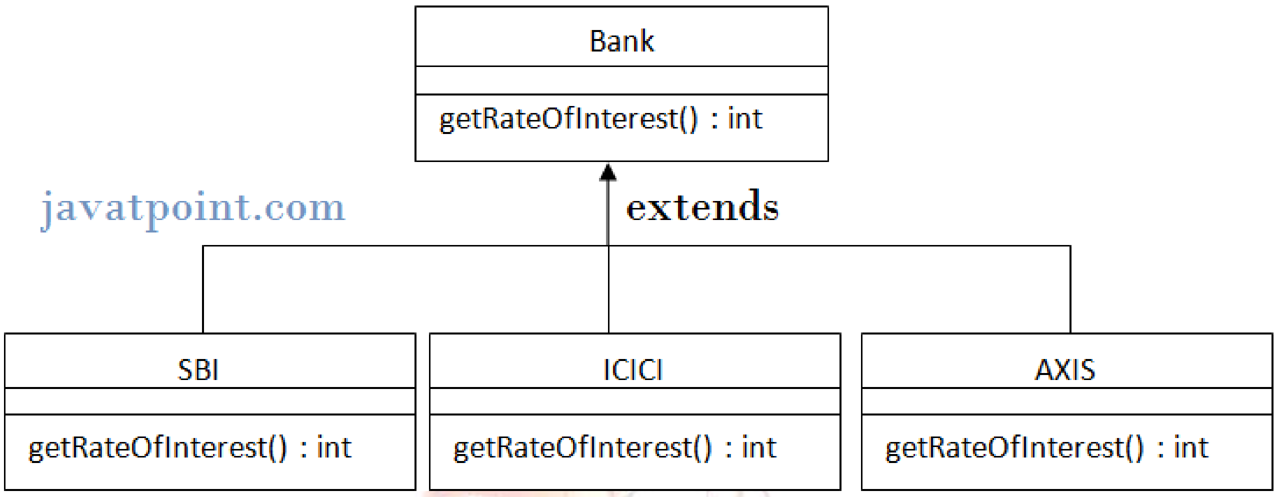
In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
class Vehicle{
void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
void run(){System.out.println("Bike is running safely");}
public static void main(String args[]){
Bike2 obj = new Bike2();
obj.run();
}
```

```
Output:Bike is running safely
```

Real example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.


Bank
getRateOfInterest() : int
javatpoint.com
extends
SBI
getRateOfInterest() : int
ICICI
getRateOfInterest() : int
AXIS
getRateOfInterest() : int

```
class Bank{  int getRateOfInterest(){return 0;}  }
class SBI extends Bank{  int getRateOfInterest(){return 8;}  }
  class ICICI extends Bank{  int getRateOfInterest(){return 7;}  }
class AXIS extends Bank{  int getRateOfInterest(){return 9;}  }
  class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

Output:

```
SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9
```

Q)Can we override static method?

No, static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Q)Why we cannot override static method?

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

Q)Can we override java main method?

No, because main is a static method.

Q)Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Java Method Overloading example:

```
class OverloadingExample{  static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
```

Java Method Overriding example

```
class Animal{  void eat(){System.out.println("eating...");}  }
class Dog extends Animal{  void eat(){System.out.println("eating bread...");}
}
```

Access Modifiers in java:

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

Private, default, protected, public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

---

1) private access modifier

The private access modifier is accessible only within class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A{  private int data=40;
private void msg(){System.out.println("Hello java");}  }
  public class Simple{   public static void main(String args[]){     A obj=new A();
   System.out.println(obj.data);//Compile Time Error     obj.msg();//Compile Time Error
   } }
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{  private A(){}//private constructor  void msg(){System.out.println("Hello java");}  }
public class Simple{   public static void main(String args[]){     A obj=new A();//Compile Time Error
 } }
```

Note: A class cannot be private or protected except nested class.

---

2) default access modifier

If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{    void msg(){System.out.println("Hello");}  }
```

```
//save by B.java
package mypack;
import pack.*;
class B{    public static void main(String args[]){     A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error    }  }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

---

3) protected access modifier

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A{  protected void msg(){System.out.println("Hello");}  }
```

```
//save by B.java
package mypack;
import pack.*;
class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
```

```
 } }
```

```
Output:Hello
```

4) public access modifier

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java
  package pack;
public class A{  public void msg(){System.out.println("Hello");}  }
```

```
//save by B.java
 package mypack;
import pack.*;
  class B{    public static void main(String args[]){     A obj = new A();     obj.msg();    }
}
```

```
Output:Hello
```

Understanding all java access modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Let's understand the access modifiers by a simple table.

Java access modifiers with method overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
class A{  protected void msg(){System.out.println("Hello java");}  }
  public class Simple extends A{  void msg(){System.out.println("Hello java");}}//C.T.Error
 public static void main(String args[]){     Simple obj=new Simple();
obj.msg();    } }
```

The default modifier is more restrictive than protected. That is why there is compile time error.

Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

Simple example of Covariant Return Type

```
class A{ A get(){return this;}  }
  class B1 extends A{  B1 get(){return this;}
void message(){System.out.println("welcome to covariant return type");}
 public static void main(String args[]){
new B1().get().message();  }  }
```

```
Output:welcome to covariant return type
```

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

super keyword in java:

The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

Usage of java super Keyword

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. super is used to invoke immediate parent class method.

1) super is used to refer immediate parent class instance variable.

Problem without super keyword

```
class Vehicle{    int speed=50;  }
class Bike3 extends Vehicle{    int speed=100;    void display(){
   System.out.println(speed);//will print speed of Bike
  }
  public static void main(String args[]){
   Bike3 b=new Bike3();
   b.display();  }  }
```

```
Output:100
```

In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is refered by instance bydefault, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable.

Solution by super keyword

```
//example of super keyword
class Vehicle{    int speed=50;  }
  class Bike4 extends Vehicle{    int speed=100;
   void display(){     System.out.println(super.speed);//will print speed of Vehicle now     }
  public static void main(String args[]){     Bike4 b=new Bike4();     b.display();
  }}
```

```
Output:50
```

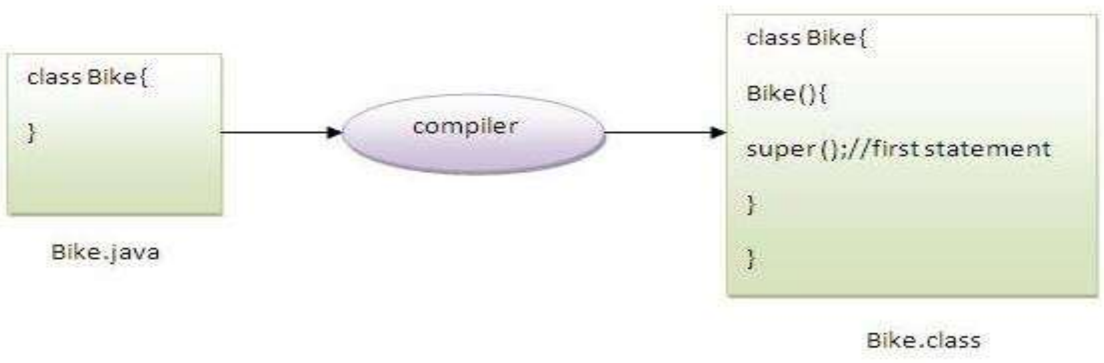
2) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor as given below:

```
class Vehicle{    Vehicle(){System.out.println("Vehicle is created");}  }
  class Bike5 extends Vehicle{    Bike5(){     super();//will invoke parent class constructor
   System.out.println("Bike is created");    }
  public static void main(String args[]){
   Bike5 b=new Bike5();     }  }
```

```
Output:Vehicle is created
       Bike is created
```

Note: super() is added in each class constructor automatically by compiler.



As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement.If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

Another example of super keyword where super() is provided by the compiler implicitly.

```
1. class Vehicle{    Vehicle(){System.out.println("Vehicle is created");}  }
2. class Bike6 extends Vehicle{    int speed;    Bike6(int speed){      this.speed=speed;
3.    System.out.println(speed);    }
4.   public static void main(String args[]){     Bike6 b=new Bike6(10);  }  }
```

```
Output:Vehicle is created
       10
```

3) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used in case subclass contains the same method as parent class as in the example given below:

```
1. class Person{  void message(){System.out.println("welcome");}  }
2.   class Student16 extends Person{  void message(){System.out.println("welcome to java");}
3.  void display(){  message();//will invoke current class message() method
4. super.message();//will invoke parent class message() method
5. }
6. public static void main(String args[]){
7. Student16 s=new Student16();
8. s.display();  }  }
```

```
Output:welcome to java
       welcome
```

In the above example Student and Person both classes have message() method if we call message() method from Student class, it will call the message() method of Student class not of Person class because priority is given to local.

In case there is no method in subclass as parent, there is no need to use super. In the example given below message() method is invoked from Student class but Student class does not have message() method, so you can directly call message() method.

Program in case super is not required

```
1. class Person{  void message(){System.out.println("welcome");}  }
2.  class Student17 extends Person{    void display(){  message();//will invoke parent class messa
ge() method  }
3.   public static void main(String args[]){  Student17 s=new Student17();  s.display();
4. }  }
```

```
Output:welcome
```

Instance initializer block:

Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created.

The initialization of the instance variable can be directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Que) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

```
class Bike{     int speed=100;  }
```

Why use instance initializer block?

Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

Example of instance initializer block

Let's see the simple example of instance initializer block the performs initialization.

```
class Bike7{     int speed;
    Bike7(){System.out.println("speed is "+speed);}
      {speed=100;}
     public static void main(String args[]){
    Bike7 b1=new Bike7();
    Bike7 b2=new Bike7();
    }     }
```

```
Output:speed is 100
       speed is 100
```

There are three places in java where you can perform operations:

1. method
2. constructor
3. block

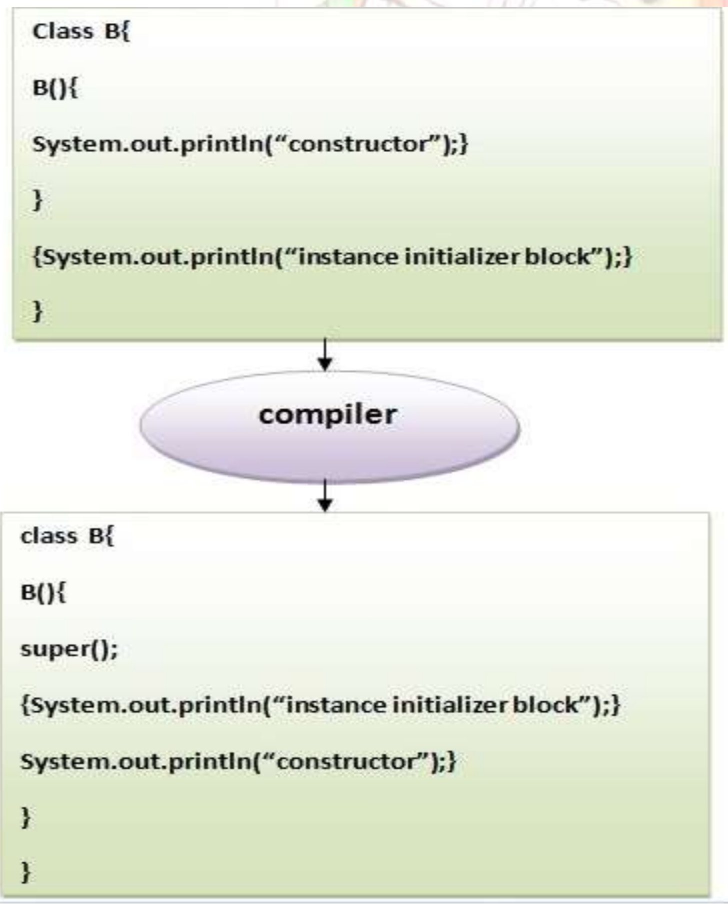
What is invoked firstly instance initializer block or constructor?

```
class Bike8{     int speed;
   Bike8(){System.out.println("constructor is invoked");}
     {System.out.println("instance initializer block invoked");}
        public static void main(String args[]){
   Bike8 b1=new Bike8();
   Bike8 b2=new Bike8();
   }
}
```

```
Output:instance initializer block invoked
     constructor is invoked
     instance initializer block invoked
     constructor is invoked
```

In the above example, it seems that instance initializer block is firstly invoked but NO. Instance intializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below:

Note: The java compiler copies the code of instance initializer block in every constructor.



```
Class B{
B(){
System.out.println("constructor");}
}
{System.out.println("instance initializer block");}
}
```

```
class B{
B(){
super();
{System.out.println("instance initializer block");}
System.out.println("constructor");}
}
}
```

Rules for instance initializer block :

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

Program of instance initializer block that is invoked after super()

```
class A{
A(){  System.out.println("parent class constructor invoked");  }  }
class B2 extends A{  B2(){  super();  System.out.println("child class constructor invoked");  }
 {System.out.println("instance initializer block is invoked");}
  public static void main(String args[]){  B2 b=new B2();  }  }
```

```
Output:parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked
```

Another example of instance block

```
class A{  A(){  System.out.println("parent class constructor invoked");  }  }
  class B3 extends A{  B3(){  super();  System.out.println("child class constructor invoked");
}
B3(int a){  super();  System.out.println("child class constructor invoked "+a);  }
 {System.out.println("instance initializer block is invoked");}
  public static void main(String args[]){
B3 b1=new B3();  B3 b2=new B3(10);  }  }
```

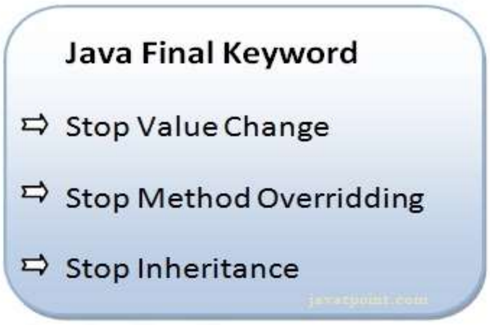
```
Output:parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked
       parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked 10
```

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

Variable   method   class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.



1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{   final int speedlimit=90;//final variable   void run(){    speedlimit=400;   }
 public static void main(String args[]){   Bike9 obj=new  Bike9();  obj.run();   }
}//end of class
```

```
Output:Compile Time Error
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{   final void run(){System.out.println("running");}  }
 class Honda extends Bike{     void run(){System.out.println("running safely with 100kmph");}
        public static void main(String args[]){     Honda honda= new Honda();
   honda.run();     } }
```

```
Output:Compile Time Error
```

---

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}
  class Honda1 extends Bike{    void run(){System.out.println("running safely with 100kmph");}
      public static void main(String args[]){    Honda1 honda= new Honda();    honda.run();
  } }
```

```
Output:Compile Time Error
```

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{   final void run(){System.out.println("running...");}  }
class Honda2 extends Bike{    public static void main(String args[]){     new Honda2().run();
   } }
```

```
Output:running...
```

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
class Student{  int id;  String name;  final String PAN_CARD_NUMBER;
...  }
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
class Bike10{    final int speedlimit;//blank final variable
      Bike10(){    speedlimit=70;    System.out.println(speedlimit);   }
    public static void main(String args[]){      new Bike10();   }  }
```

```
Output:70
```

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
class A{    static final int data;//static blank final variable    static{ data=50;}
  public static void main(String args[]){      System.out.println(A.data);   }  }
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{    int cube(final int n){      n=n+2;//can't be changed as n is final
   n*n*n;    }
  public static void main(String args[]){      Bike11 b=new Bike11();
    b.cube(5);   }  }
```

```
Output:Compile Time Error
```

Q) Can we declare a constructor final?

No, because constructor is never inherited.

Polymorphism in Java:

Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

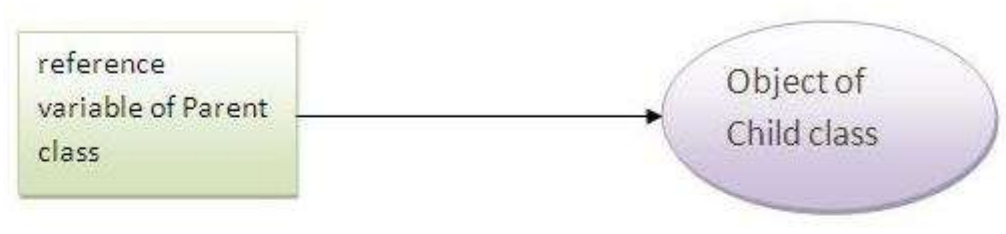
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



```
1.  class A{}
2.  class B extends A{}
1.  A a=new B();//upcasting
```

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.

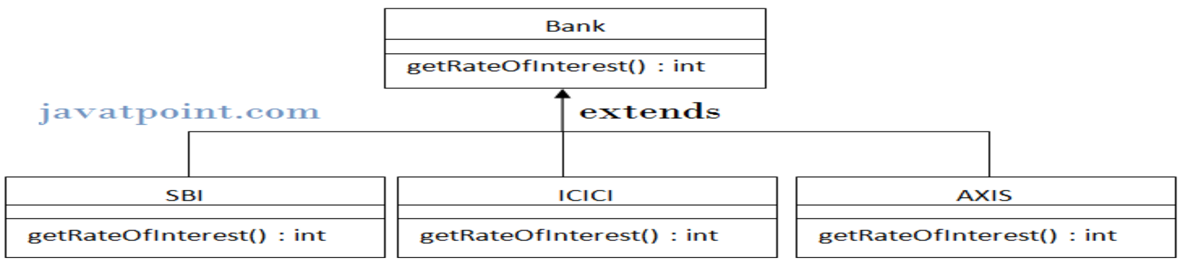
Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
1.  class Bike{    void run(){System.out.println("running");}  }
2.  class Splender extends Bike{    void run(){System.out.println("running safely with 60km");}
3.      public static void main(String args[]){     Bike b = new Splender();//upcasting
4.      b.run();    }  }
```

```
Output:running safely with 60km.
```

Real example of Java Runtime Polymorphism

Consider a scenario, Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.


Bank
getRateOfInterest() : int
javatpoint.com
extends
SBI
getRateOfInterest() : int
ICICI
getRateOfInterest() : int
AXIS
getRateOfInterest() : int

Note: It is also given in method overriding but there was no upcasting.

```
class Bank{  int getRateOfInterest(){return 0;}}
class SBI extends Bank{  int getRateOfInterest(){return 8;}  }
class ICICI extends Bank{  int getRateOfInterest(){return 7;}  }
class AXIS extends Bank{  int getRateOfInterest(){return 9;}  }
  class Test3{  public static void main(String args[]){  Bank b1=new SBI();
Bank b2=new ICICI();  Bank b3=new AXIS();
System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest());
} }
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Java Runtime Polymorphism with data member

Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

```
class Bike{  int speedlimit=90;  }
class Honda3 extends Bike{  int speedlimit=150;
 public static void main(String args[]){    Bike obj=new Honda3();
  System.out.println(obj.speedlimit);//90  }
```

```
Output:90
```

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
class Animal{  void eat(){System.out.println("eating");}  }
  class Dog extends Animal{  void eat(){System.out.println("eating fruits");}  }
  class BabyDog extends Dog{  void eat(){System.out.println("drinking milk");}
  public static void main(String args[]){
```

```
Animal a1,a2,a3;
a1=new Animal();  a2=new Dog();  a3=new BabyDog();
a1.eat();  a2.eat();  a3.eat();
}  }
```

```
Output: eating
        eating fruits
        drinking Milk
```

Try for Output

```
class Animal{  void eat(){System.out.println("animal is eating...");}  }
  class Dog extends Animal{  void eat(){System.out.println("dog is eating...");}  }
  class BabyDog1 extends Dog{
public static void main(String args[]){  Animal a=new BabyDog1();  a.eat();  }}
```

```
Output: Dog is eating
```

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. static binding (also known as early binding).
2. dynamic binding (also known as late binding).

Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

```
int data=30;  Here data variable is a type of int.
```

2) References have a type

```
class Dog{  public static void main(String args[]){   Dog d1;//Here d1 is a type of Dog   }  }
```

3) Objects have a type

An object is an instance of particular java class,but it is also an instance of its superclass.

```
class Animal{}
class Dog extends Animal{   public static void main(String args[]){    Dog d1=new Dog();
 } }
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```
class Dog{   private void eat(){System.out.println("dog is eating...");}
  public static void main(String args[]){    Dog d1=new Dog();
 d1.eat();   } }
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
class Animal{   void eat(){System.out.println("animal is eating...");}  }
 class Dog extends Animal{   void eat(){System.out.println("dog is eating...");}
  public static void main(String args[]){    Animal a=new Dog();
 a.eat();   } }
```

```
Output:dog is eating...
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

Java instanceof:

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple);//true
 } }
```

```
Output:true
```

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

Another example of java instanceof operator

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal
  public static void main(String args[]){
 Dog1 d=new Dog1();
 System.out.println(d instanceof Animal);//true
 } }
```

```
Output:true
```

---

instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
  System.out.println(d instanceof Dog2);//false
 } }
```

```
Output:false
```

Downcasting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

```
Dog d=new Animal();//Compilation error
```

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

```
Dog d=(Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime
```

Possibility of downcasting with instanceof

Let's see the example, where downcasting is possible by instanceof operator.

```
class Animal { }
  class Dog3 extends Animal {
  static void method(Animal a) {
    if(a instanceof Dog3){
       Dog3 d=(Dog3)a;//downcasting
       System.out.println("ok downcasting performed");
    }   }
     public static void main (String [] args) {
    Animal a=new Dog3();     Dog3.method(a);    }
     }
```

```
Output:ok downcasting performed
```

Downcasting without the use of java instanceof

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

```
class Animal { }
class Dog4 extends Animal {
  static void method(Animal a) {
       Dog4 d=(Dog4)a;//downcasting
       System.out.println("ok downcasting performed");
  }
   public static void main (String [] args) {
    Animal a=new Dog4();     Dog4.method(a);
  } }
```

```
Output:ok downcasting performed
```

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

```
Animal a=new Animal();  Dog.method(a);
//Now ClassCastException but not in case of instanceof operator
```

Understanding Real use of instanceof in java

Let's see the real use of instanceof keyword by the example given below.

```
interface Printable{}
class A implements Printable{  public void a(){System.out.println("a method");}  }
class B implements Printable{  public void b(){System.out.println("b method");}  }
  class Call{  void invoke(Printable p){//upcasting
if(p instanceof A){  A a=(A)p;//Downcasting   a.a();  }
if(p instanceof B){  B b=(B)p;//Downcasting   b.b();  }
  }
}//end of Call class
class Test4{  public static void main(String args[]){  Printable p=new B();
Call c=new Call();  c.invoke(p); }  }
```

```
Output: b method
```

Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Before learning java abstract class, let's understand the abstraction in java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

```
abstract class A{}
```

abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Example abstract method

```java
abstract void printStatus();//no body and abstract
```

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```java
abstract class Bike{   abstract void run();  }
class Honda4 extends Bike{  void run(){System.out.println("running safely..");}
  public static void main(String args[]){
 Bike obj = new Honda4();   obj.run();  }  }
```

```
running safely..
```

Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.

A **factory method** is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```java
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
  class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
  //In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape() met
hod
s.draw();
}  }
```

```
drawing circle
```

Another example of abstract class in java

File: TestBank.java

```
 abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{    int getRateOfInterest(){return 7;}  }
class PNB extends Bank{    int getRateOfInterest(){return 7;}    }
   class TestBank{    public static void main(String args[]){
Bank b=new SBI();//if object is PNB, method of PNB will be invoked
int interest=b.getRateOfInterest();
System.out.println("Rate of Interest is: "+interest+" %");
}}
```

```
Rate of Interest is: 7 %
```

Abstract class having constructor, data member, methods etc.

An abstract class can have data member, abstract method, method body, constructor and even main() method.

File: TestAbstraction2.java

```
//example of abstract class that have method body
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }
 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 } }
```

```
bike is created
running safely..
gear changed
```

Rule: If there is any abstract method in a class, that class must be abstract.

```
class Bike12{  abstract void run();  }
```

```
compile time error
```

Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
interface A{  void a();  void b();  void c();  void d();  }
 abstract class B implements A{  public void c(){System.out.println("I am C");}  }
 class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}
  class Test5{  public static void main(String args[]){  A a=new M();
a.a();  a.b();  a.c();  a.d();  }}
```

```
Output:I am a
    I am b
    I am c
    I am d
```

Interface in Java

1. Interface
2. Example of Interface

An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.

The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

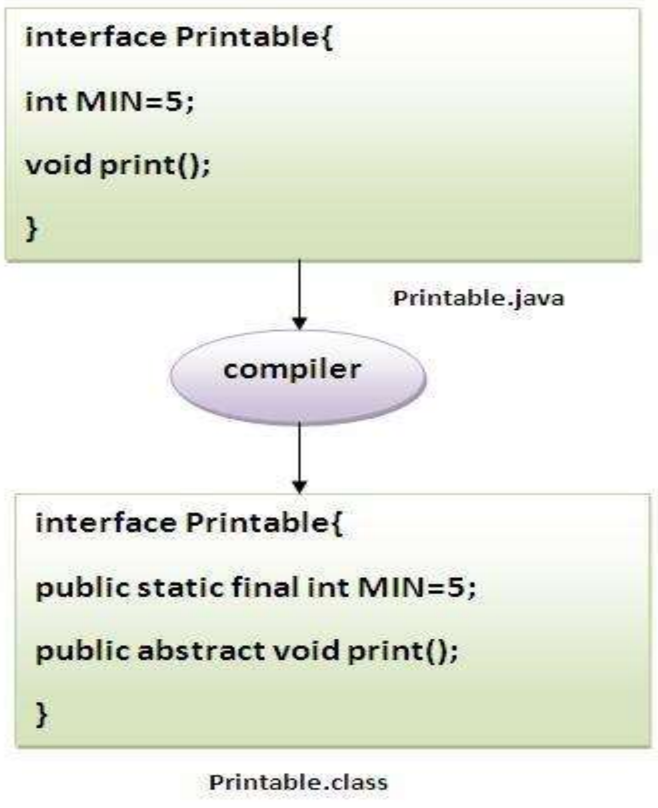
Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

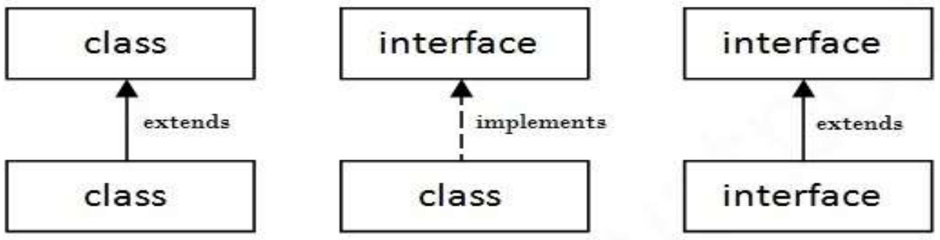
The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

In other words, Interface fields are public, static and final bydefault, and methods are public and abstract.



Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Simple example of Java interface

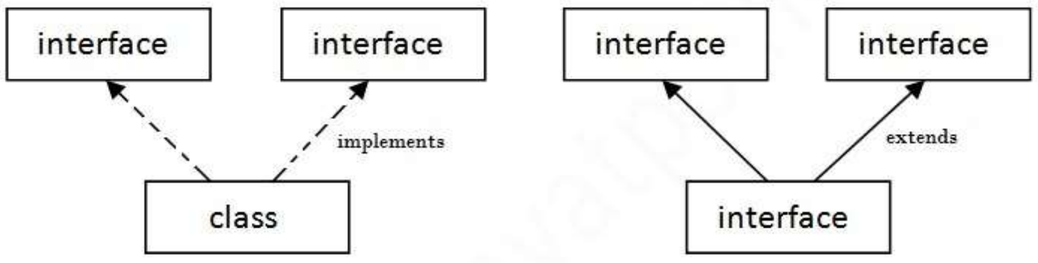
In this example, Printable interface have only one method, its implementation is provided in the A class.

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){  A6 obj = new A6();  obj.print();   }  }
```

Output:Hello

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{  void print();  }
  interface Showable{  void show();  }
  class A7 implements Printable,Showable{  public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
  public static void main(String args[]){  A7 obj = new A7(); obj.print();
obj.show();   }  }
```

```
Output:Hello       Welcome
```

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```
interface Printable{  void print();  }
  interface Showable{  void print();  }
  class testinterface1 implements Printable,Showable{
public void print(){System.out.println("Hello");}
  public static void main(String args[]){
testinterface1 obj = new testinterface1();
obj.print();   }  }
```

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class A, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable{  void print();  }
interface Showable extends Printable{  void show();  }
class Testinterface2 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
Testinterface2 obj = new Testinterface2();
obj.print();
9. obj.show();  } }
```

```
Hello
Welcome
```

Q) What is marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written?
public interface Serializable{
}
```

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

```
interface printable{  void print();  interface MessagePrintable{    void msg();  } }
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract**methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.

6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
//Creating interface that has 4 methods
interface A{  void a();//bydefault, public and abstract  void b();  void c();  void d();  }
  //Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A{  public void c(){System.out.println("I am C");}  }
  //Creating subclass of abstract class, now we need to provide the implementation of rest of the methods
class M extends B{  public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}  public void d(){System.out.println("I am d");}
}
  //Creating a test class that calls the methods of A interface
class Test5{  public static void main(String args[]){  A a=new M();
a.a();  a.b();  a.c();  a.d();  }}
```

Output:

```
I am a

I am b

I am c

I am d
```

Java Package:

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

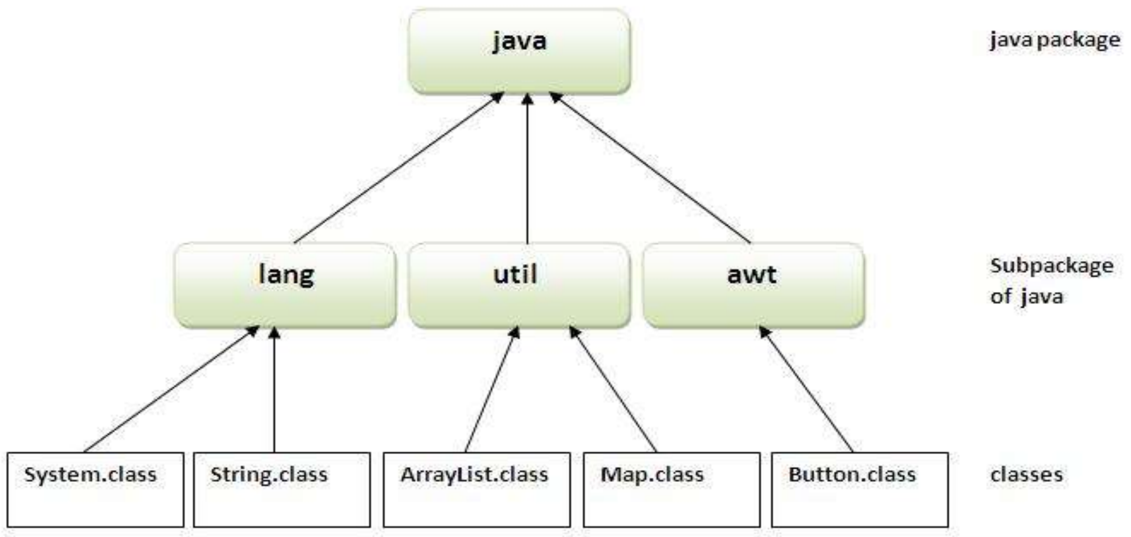
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  } }
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For **example**

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), 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

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

```
Output:Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
 package pack;
```

```
public class A{    public void msg(){System.out.println("Hello");}  }
```

```
//save by B.java
 package mypack;
import pack.*;
  class B{    public static void main(String args[]){     A obj = new A();     obj.msg();    }  }
```

```
Output:Hello
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java

package pack;
public class A{    public void msg(){System.out.println("Hello");}  }
```

```
//save by B.java
package mypack;
import pack.A;
class B{    public static void main(String args[]){     A obj = new A();     obj.msg();    }
}
```

```
Output:Hello
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
  package pack;
public class A{    public void msg(){System.out.println("Hello");}  }
```

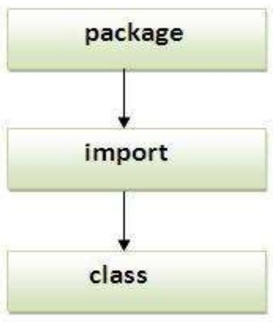
```
//save by B.java
  package mypack;
class B{    public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
5.    obj.msg();    } }
```

```
Output:Hello
```

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

```
1.  package com.javatpoint.core;
2.  class Simple{    public static void main(String args[]){
3.    System.out.println("Hello subpackage");    } }
```

To Compile: javac -d . Simple.java

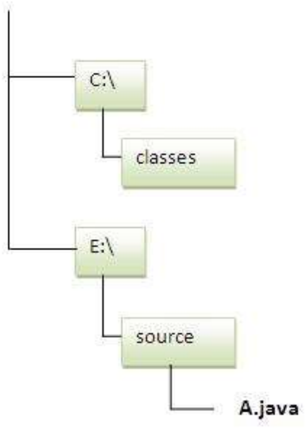
To Run: java com.javatpoint.core.Simple

```
Output:Hello subpackage
```

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive.

For example:



```
//save as Simple.java
  package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   } }
```

To Compile:

```
e:\sources> javac -d c:\classes Simple.java
```

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

```
e:\sources> set classpath=c:\classes;.;
```

```
e:\sources> java mypack.Simple
```

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

```
e:\sources> java -classpath c:\classes mypack.Simple
```

```
Output:Welcome to package
```

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
  - By setting the classpath in the command prompt
  - By -classpath switch
- Permanent
  - By setting the classpath in the environment variables
  - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

```
//save as C.java otherwise Compilte Time Error
 class A{}
class B{}
public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
//save as A.java
 package javatpoint;
public class A{}
```

```
//save as B.java
  package javatpoint;
public class B{}
```

What is static import feature of Java5?

Static Import:

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Advantage of static import:

- Less coding is required if you have access any static member of a class oftenly.

Disadvantage of static import:

- If you overuse the static import feature, it makes the program unreadable and unmaintainable.

Simple Example of static import

```
import static java.lang.System.*;
class StaticImportExample{
  public static void main(String args[]){
   out.println("Hello");//Now no need of System.out
   out.println("Java");
 }  }
```

```
Output:Hello      Java
```

What is the difference between import and static import?

The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

Encapsulation in Java

Encapsulation in java is a *process of wrapping code and data together into a single unit*, for example capsule i.e. mixed of several medicines.

We can create a fully encapsulated class in java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of fully encapsulated class.

Advantage of Encapsulation in java

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

Simple example of encapsulation in java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

```
//save as Student.java
package com.javatpoint;
public class Student{  private String name;
  public String getName(){  return name;  }
public void setName(String name){  this.name=name  }  }
```

```
//save as Test.java
```

```
2. package com.javatpoint;
3. class Test{
4. public static void main(String[] args){  Student s=new Student();
5. s.setname("vijay");  System.out.println(s.getName());  } }
```

```
Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output: vijay
```

Object class in Java

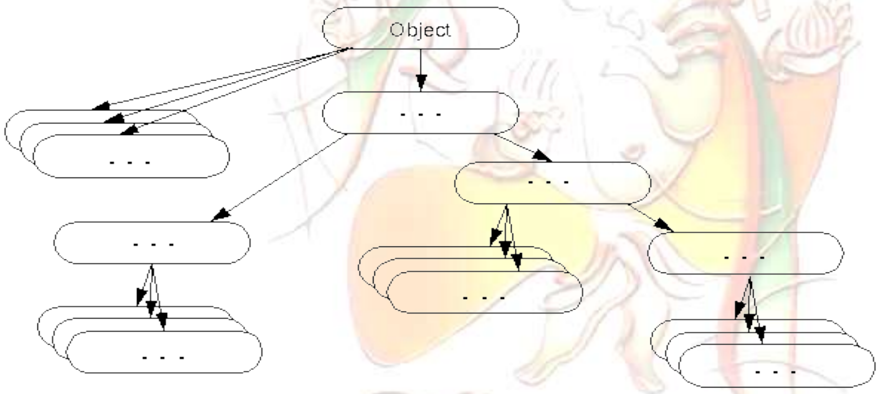
The **Object class** is the parent class of all the classes in java bydefault. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

```
1. Object obj=getObject();//we don't what object would be returned from this method
```

The Object class provides some common behaviours to all the objects such as object can be compared, object can be cloned, object can be notified etc.



Methods of Object class

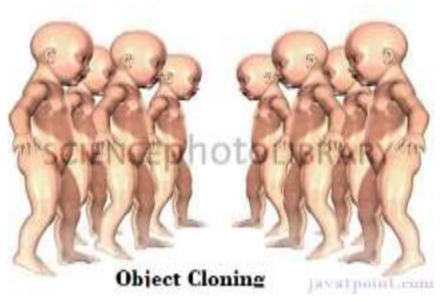
The Object class provides many methods. They are as follows:

Method	**Description**
public final ClassgetClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.

public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified miliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

We will have the detailed learning of these methods in next chapters.

Object Cloning in Java



The **object cloning** is a way to create exact copy of an object. For this purpose, clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

```
1. protected Object clone() throws CloneNotSupportedException
```

Why use clone() method ?

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.

Advantage of Object cloning

Less processing task.

Example of clone() method (Object cloning)

Let's see the simple example of object cloning

```
class Student18 implements Cloneable{
int rollno;  String name;
Student18(int rollno,String name){  this.rollno=rollno; this.name=name;  }
  public Object clone()throws CloneNotSupportedException{  return super.clone();  }
public static void main(String args[]){
try{  Student18 s1=new Student18(101,"amit");
Student18 s2=(Student18)s1.clone();    System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);
}catch(CloneNotSupportedException c){}
} }
```

```
Output:101 amit
       101 amit
```

As you can see in the above example, both reference variables have the same value. Thus, the clone() copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

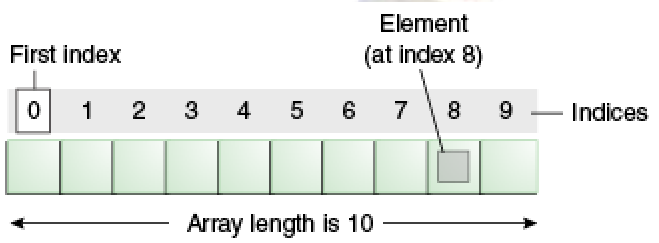
If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

Java Array

Normally, array is a collection of similar type of elements that have contiguous memory location.

Java array is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.



Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in java

Syntax to Declare an Array in java

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in java

```
arrayRefVar=new datatype[size];
```

Example of single dimensional java array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

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

int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization  a[1]=20;  a[2]=70;  a[3]=40;  a[4]=50;
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

```
Output: 10      20      70      40      50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
class Testarray1{
```

```java
public static void main(String args[]){
 int a[]={33,3,4,5};//declaration, instantiation and initialization
  //printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);   }}
```

```
Output:33      3      4      5
```

Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

```java
class Testarray2{
static void min(int arr[]){  int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
System.out.println(min);  }
  public static void main(String args[]){
  int a[]={33,3,4,5};
min(a);//passing array to method
  }}
```

```
Output:3
```

Multidimensional array in java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java

```java
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in java

```java
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in java

```java
arr[0][0]=1;   arr[0][1]=2;  arr[0][2]=3; arr[1][0]=4;  arr[1][1]=5;
```

```
arr[1][2]=6;   arr[2][0]=7;  arr[2][1]=8;  arr[2][2]=9;
```

Example of Multidimensional java array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
class Testarray3{
public static void main(String args[]){
  //declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
 //printing 2D array
for(int i=0;i<3;i++){   for(int j=0;j<3;j++){     System.out.print(arr[i][j]+" ");   }
 System.out.println();  }
}}
```

```
Output:1 2 3      2 4 5      4 4 5
```

What is class name of java array?

In java, array is an object. For array object, an proxy class is created whose name can be obtained by getClass().getName() method on the object.

```
class Testarray4{
public static void main(String args[]){
  int arr[]={4,4,5};
  Class c=arr.getClass();  String name=c.getName();
 System.out.println(name);
}}
```

```
Output:I
```

Copying a java array

We can copy an array to another by the arraycopy method of System class.

Syntax of arraycopy method

```
public static void arraycopy(
Object src, int srcPos,Object dest, int destPos, int length
)
```

Example of arraycopy method

```
class TestArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));     } }
```

```
Output:caffein
```

Addition of 2 matrices in java

Let's see a simple example that adds two matrices.

```
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};  int b[][]={{1,3,4},{3,4,5}};
  //creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];
//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){  c[i][j]=a[i][j]+b[i][j];  System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}  }}
```

```
Output:2 6 8
       6 8 10
```

Call by Value and Call by Reference in Java

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Example of call by value in java

In case of call by value original value is not changed. Let's take a simple example:

```
class Operation{   int data=50;     void change(int data){   data=data+100;//changes will be in the local variable only   }
 public static void main(String args[]){
   Operation op=new Operation();
   System.out.println("before change "+op.data);
   op.change(500);
6.     System.out.println("after change "+op.data);   }}
```

```
Output:before change 50

       after change 50
```

Another Example of call by value in java

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

```
1.  class Operation2{   int data=50;
2.   void change(Operation2 op){   op.data=op.data+100;//changes will be in the instance variable
3.   }
4.   public static void main(String args[]){     Operation2 op=new Operation2();
5.       System.out.println("before change "+op.data);
6.     op.change(op);//passing object     System.out.println("after change "+op.data);
7.     }  }
```

```
Output:before change 50

       after change 150
```

strictfp keyword

The strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

Legal code for strictfp keyword

The strictfp keyword can be applied on methods, classes and interfaces.

```
1.  strictfp class A{}//strictfp applied on class
1.  strictfp interface M{}//strictfp applied on interface
1.  class A{  void m(){}//strictfp applied on method  }
```

Illegal code for strictfp keyword

The strictfp keyword can be applied on abstract methods, variables or constructors.

```
1.  class B{  strictfp abstract void m();//Illegal combination of modifiers  }
1.  class B{  strictfp int data=10;//modifier strictfp not allowed here }
1.  class B{  strictfp B(){}//modifier strictfp not allowed here  }
```

Creating API Document | javadoc tool

We can create document api in java by the help of **javadoc** tool. In the java file, we must use the documentation comment /**... */ to post information for the class, method, constructor, fields etc.

Let's see the simple class that contains documentation comment.

```
package com.abc;
/** This class is a user-defined class that contains one methods cube.*/
public class M{
/** The cube method prints cube of the given number */
public static void  cube(int n){System.out.println(n*n*n);}
}
```

To create the document API, you need to use the javadoc tool followed by java file name. There is no need to compile the javafile.

On the command prompt, you need to write:

```
javadoc M.java
```

to generate the document api. Now, there will be created a lot of html files. Open the index.html file to get the information about the classes.

Java Command Line Arguments

1. Command Line Argument
2. Simple example of command-line argument
3. Example of command-line argument that prints all the values

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
class CommandLineExample{
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);  }  }
```

```
compile by > javac CommandLineExample.java
run by > java CommandLineExample sonoo
```

```
Output: Your first argument is: sonoo
```

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

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

compile by > javac A.java

run by > java A sonoo jaiswal 1 3 abc

```
Output: sonoo
     jaiswal
     1
     3
     abc
```

Difference between object and class

There are many differences between object and class. A list of differences between object and class are given below:

No.	Object	Class
1)	Object is an **instance** of a class.	Class is a **blueprint or template**from which objects are created.
2)	Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
3)	Object is a **physical** entity.	Class is a **logical** entity.
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using **class keyword** e.g. class Student{}
5)	Object is created **many times** as per requirement.	Class is declared **once**.
6)	Object **allocates memory when it is created**.	Class **doesn't allocated memory when it is created**.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Java Method Overloading example

```
class OverloadingExample{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
```

Java Method Overriding example

```
class Animal{  void eat(){System.out.println("eating...");}  }
class Dog extends Animal{  void eat(){System.out.println("eating bread...");}  }
```

New Features in Java

There are many new features that have been added in java. There are major enhancement made in Java5, Java6 and Java7 like **auto-boxing**, **generics**, **var-args**, **java annotations**, **enum**, **premain method** etc.

Most of the interviewers ask questions from this chapter.

Do You Know ?

1. How to create generic class and generic method in java ?
2. What is annotation and how to create custom annotation ?
3. What is the advantage of assertion and where we should not use it ?
4. What is variable argument and what rules are defined for variable argument ?
5. What is the difference between import and static import ?
6. How autoboxing is applied in method overloading. Which concept beats autoboxing ?
7. What is enum type and how to specify specific value to the enum constants ?

J2SE 4 Features

The important feature of J2SE 4 is assertions. It is used for testing.

- Assertion (Java 4)

J2SE 5 Features

The important features of J2SE 5 are generics and assertions. Others are auto-boxing, enum, var-args, static import, for-each loop (enhanced for loop etc.

- For-each loop (Java 5)
- Varargs (Java 5)
- Static Import (Java 5)
- Autoboxing and Unboxing (Java 5)
- Enum (Java 5)
- Covariant Return Type (Java 5)
- Annotation (Java 5)
- Generics (Java 5)

JavaSE 6 Features

The important feature of JavaSE 6 is premain method (also known as instrumentation).

- Instrumentation (premain method) (Java 6)

JavaSE 7 Features

The important features of JavaSE 7 are try with resource, catching multiple exceptions etc.

- String in switch statement (Java 7)
- Binary Literals (Java 7)
- The try-with-resources (Java 7)
- Caching Multiple Exceptions by single catch (Java 7)
- Underscores in Numeric Literals (Java 7)

Assertion:

Assertion is a statement in java. It can be used to test your assumptions about the program.

While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose.

Advantage of Assertion:

It provides an effective way to detect and correct programming errors.

Syntax of using Assertion:

There are two ways to use assertion. First way is:

```
1. assert expression;
```

and second way is:

```
assert expression1 : expression2;
```

Simple Example of Assertion in java:

```
import java.util.Scanner;

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

  Scanner scanner = new Scanner( System.in );
  System.out.print("Enter ur age ");

  int value = scanner.nextInt();
  assert value>=18:" Not valid";

  System.out.println("value is "+value);
 }
}


```

If you use assertion, It will not run simply because assertion is disabled by default. To enable the assertion, **-ea** or **-enableassertions** switch of java must be used.

Compile it by: **javac AssertionExample.java**

Run it by: **java -ea AssertionExample**

```
Output: Enter ur age 11
      Exception in thread "main" java.lang.AssertionError: Not valid
```

Where not to use Assertion:

There are some situations where assertion should be avoid to use. They are:

1. According to Sun Specification, assertion should not be used to check arguments in the public methods because it should result in appropriate runtime exception e.g. IllegalArgumentException, NullPointerException etc.
2. Do not use assertion, if you don't want any error in any situation.

For-each loop (Advanced or Enhanced For loop):

The for-each loop introduced in Java5. It is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

Advantage of for-each loop:

- It makes the code more readable.
- It elimnates the possibility of programming errors.

Syntax of for-each loop:

```
for(data_type variable : array | collection){}
```

Simple Example of for-each loop for traversing the array elements:

```java

class ForEachExample1{
  public static void main(String args[]){
   int arr[]={12,13,14,44};

   for(int i:arr){
     System.out.println(i);
   }

 }
}

```

```
Output:12
       13
       14
       44
```

Simple Example of for-each loop for traversing the collection elements:

```java
import java.util.*;
class ForEachExample2{
  public static void main(String args[]){
   ArrayList<String> list=new ArrayList<String>();
   list.add("vimal");
   list.add("sonoo");
   list.add("ratan");

   for(String s:list){
     System.out.println(s);
   }

 }
}

```

```
Output:vimal
       sonoo
       ratan
```

Static Import:

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Advantage of static import:

- Less coding is required if you have access any static member of a class oftenly.

Disadvantage of static import:

- If you overuse the static import feature, it makes the program unreadable and unmaintainable.

Simple Example of static import

```java
import static java.lang.System.*;
```

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

   out.println("Hello");//Now no need of System.out
   out.println("Java");

 }
}

```

```
Output:Hello
       Java
```

What is the difference between import and static import?

The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

Autoboxing and Unboxing:

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

Advantage of Autoboxing and Unboxing:

No need of conversion between primitives and Wrappers manually so less coding is required.

Simple Example of Autoboxing in java:

```java

class BoxingExample1{
  public static void main(String args[]){
    int a=50;
        Integer a2=new Integer(a);//Boxing

        Integer a3=5;//Boxing

        System.out.println(a2+" "+a3);
 }
}

```

```
Output:50 5
```

Simple Example of Unboxing in java:

The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing. Let's see the example of unboxing:

```java

class UnboxingExample1{
  public static void main(String args[]){
    Integer i=new Integer(50);
        int a=i;

        System.out.println(a);
 }
}
```

```
Output:50
```

Autoboxing and Unboxing with comparison operators

Autoboxing can be performed with comparison operators. Let's see the example of boxing with comparison operator:

```

class UnboxingExample2{
  public static void main(String args[]){
    Integer i=new Integer(50);

        if(i<100){            //unboxing internally
        System.out.println(i);
        }
  }
}

```

```
Output:50
```

Autoboxing and Unboxing with method overloading

In method overloading, boxing and unboxing can be performed. There are some rules for method overloading with boxing:

- **Widening beats boxing**
- **Widening beats varargs**
- **Boxing beats varargs**

1) Example of Autoboxing where widening beats boxing

If there is possibility of widening and boxing, widening beats boxing.

```

class Boxing1{
  static void m(int i){System.out.println("int");}
  static void m(Integer i){System.out.println("Integer");}

  public static void main(String args[]){
   short s=30;
   m(s);
  }
}

```

```
Output:int
```

2) Example of Autoboxing where widening beats varargs

If there is possibility of widening and varargs, widening beats var-args.

```

class Boxing2{
  static void m(int i, int i2){System.out.println("int int");}
  static void m(Integer... i){System.out.println("Integer...");}

  public static void main(String args[]){
   short s1=30,s2=40;
   m(s1,s2);
  }
}
```

```
Output:int int
```

3) Example of Autoboxing where boxing beats varargs

Let's see the program where boxing beats variable argument:

```

class Boxing3{
  static void m(Integer i){System.out.println("Integer");}
  static void m(Integer... i){System.out.println("Integer...");}

  public static void main(String args[]){
   int a=30;
   m(a);
  }
}

```

```
Output:Integer
```

Method overloading with Widening and Boxing

Widening and Boxing can't be performed as given below:

```

class Boxing4{
  static void m(Long l){System.out.println("Long");}

  public static void main(String args[]){
   int a=30;
   m(a);
  }
}

```

```
Output:Compile Time Error
```

Enum

An enum is a data type which contains fixed set of constants. It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The enum constants are static and final implicitely. It is available from Java 5. Enums can be thought of as classes that have fixed set of constants.

Points to remember for Enum:

1. enum improves type safety
2. enum can be easily used in switch
3. enum can be traversed
4. enum can have fields, constructors and methods
5. enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Simple Example of enum in java:

```

class EnumExample1{

public enum Season { WINTER, SPRING, SUMMER, FALL }

public static void main(String[] args) {
for (Season s : Season.values())
System.out.println(s);

}}

```

```
Output:WINTER
       SPRING
       SUMMER
       FALL
```

What is the purpose of values() method in enum?

The java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

Internal code generated by the compiler for the above example of enum type

The java compiler internally creates a static and final class that extends the Enum class as shown in the below example:

```
public static final class EnumExample1$Season extends Enum
{
 private EnumExample1$Season(String s, int i)
   {         super(s, i);      }

   public static EnumExample1$Season[] values()
   {         return (EnumExample1$Season[])$VALUES.clone();      }

   public static EnumExample1$Season valueOf(String s)
   {         return (EnumExample1$Season)Enum.valueOf(EnumExample1$Season, s);      }

   public static final EnumExample1$Season WINTER;
   public static final EnumExample1$Season SPRING;
   public static final EnumExample1$Season SUMMER;
   public static final EnumExample1$Season FALL;
   private static final EnumExample1$Season $VALUES[];

   static
   {
       WINTER = new EnumExample1$Season("WINTER", 0);
       SPRING = new EnumExample1$Season("SPRING", 1);
       SUMMER = new EnumExample1$Season("SUMMER", 2);
       FALL = new EnumExample1$Season("FALL", 3);
       $VALUES = (new EnumExample1$Season[] {
           WINTER, SPRING, SUMMER, FALL
       });
   }

}
```

Defining enum:

The enum can be defined within or outside the class because it is similar to a class.

Example of enum that is defined outside the class:

```

enum Season { WINTER, SPRING, SUMMER, FALL }

class EnumExample2{
public static void main(String[] args) {

Season s=Season.WINTER;
System.out.println(s);

}}

```

```
Output:WINTER
```

Example of enum that is defined within the class:

```
class EnumExample3{
enum Season { WINTER, SPRING, SUMMER, FALL; }//semicolon(;) is optional here

public static void main(String[] args) {
Season s=Season.WINTER;//enum type is required to access WINTER
System.out.println(s);

}}

```

```
Output:WINTER
```

Initializing specific value to the enum constants:

The enum constants have initial value that starts from 0, 1, 2, 3 and so on. But we can initialize the specific value to the enum constants by defining fields and constructors. As specified earlier, Enum can have fields, constructors and methods.

Example of specifying initial value to the enum constants

```
class EnumExample4{
enum Season{
WINTER(5), SPRING(10), SUMMER(15), FALL(20);

private int value;
private Season(int value){
this.value=value;
}
}
public static void main(String args[]){
for (Season s : Season.values())
System.out.println(s+" "+s.value);

}}

```

```
Output:WINTER 5
       SPRING 10
       SUMMER 15
       FALL 20
```

Constructor of enum type is private if you don't declare private compiler internally have private constructor

```
enum Season{
WINTER(10),SUMMER(20);
private int value;
Season(int value){  this.value=value;  } }
```

Internal code generated by the compiler for the above example of enum type

```
final class Season extends Enum
{

   public static Season[] values()
   {
       return (Season[])$VALUES.clone();
   }

   public static Season valueOf(String s)
   {
       return (Season)Enum.valueOf(Season, s);
   }

   private Season(String s, int i, int j)
   {
       super(s, i);
       value = j;
   }

   public static final Season WINTER;
   public static final Season SUMMER;
   private int value;
   private static final Season $VALUES[];

   static
   {
       WINTER = new Season("WINTER", 0, 10);
       SUMMER = new Season("SUMMER", 1, 20);
       $VALUES = (new Season[] {
           WINTER, SUMMER
       });
   }
}
```

Can we create the instance of enum by new keyword?

No, because it contains private constructors only.

Can we have abstract method in enum?

Yes, ofcourse! we can have abstract methods and can provide the implementation of these methods.

Applying enum on switch statement

We can apply enum on switch statement as in the given example:

Example of applying enum on switch statement

```
class EnumExample5{
enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}

public static void main(String args[]){

Day day=Day.MONDAY;

switch(day){
case SUNDAY:
 System.out.println("sunday");
 break;
case MONDAY:
 System.out.println("monday");
 break;
default:
System.out.println("other day");
}

}}

```

```
Output:monday
```

Java Annotations

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in java are used to provide additional information, so it is an alternative option for XML and java marker interfaces.

First, we will learn some built-in annotations then we will move on creating and using custom annotations.

Built-In Java Annotations

There are several built-in annotations in java. Some annotations are applied to java code and some to other annotations.

Built-In Java Annotations used in java code

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations

- @Target
- @Retention
- @Inherited
- @Documented

Understanding Built-In Annotations in java

Let's understand the built-in annotations first.

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

```
class Animal{
void eatSomething(){System.out.println("eating something");}
}

class Dog extends Animal{
@Override
void eatsomething(){System.out.println("eating foods");}//should be eatSomething
}
```

```java

class TestAnnotation1{
public static void main(String args[]){
Animal a=new Dog();
a.eatSomething();
}}
```

```
Output:Comple Time Error
```

@SuppressWarnings

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

```java
import java.util.*;
class TestAnnotation2{
@SuppressWarnings("unchecked")
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("sonoo");  list.add("vimal");  list.add("ratan");

for(Object obj:list)  System.out.println(obj);
}}
```

```
Now no warning at compile time.
```

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

@Deprecated

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

```java
class A{
void m(){System.out.println("hello m");}

@Deprecated
void n(){System.out.println("hello n");}
}

class TestAnnotation3{
public static void main(String args[]){

A a=new A();
a.n();
}}
```

At Compile Time:

Note: Test.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

At Runtime:hello n

Custom Annotation

To create and use custom java annotation, visit the next page.

Java Custom Annotation

Java Custom annotations or Java User-defined annotations are easy to create and use. The *@interface* element is used to declare an annotation. For example:

```
@interface MyAnnotation{}
```

Here, MyAnnotation is the custom annotation name.

Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. It may assign a default value to the method.

Types of Annotation

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

```
@interface MyAnnotation{}
```

The @Override and @Deprecated are marker annotations.

2) Single-Value Annotation

An annotation that has one method, is called single-value annotation. For example:

```
@interface MyAnnotation{
int value();
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation{
int value() default 0;
}
```

How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.

```
@MyAnnotation(value=10)
```

The value can be anything.

3) Mulit-Value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

```
@interface MyAnnotation{
int value1();  String value2();  String value3();  }  }
```

We can provide the default value also. For example:

```
@interface MyAnnotation{  int value1() default 1;  String value2() default "";  String value3() default "xyz"
;  }
```

How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

```
@MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad")
```

Built-in Annotations used in custom annotations in java

- @Target , @Retention, @Inherited, @Documented

@Target

@Target tag is used to specify at which type, the annotation is used.

The java.lang.annotation.**ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. Let's see the constants of ElementType enum:

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

Example to specify annoation for a class

```
@Target(ElementType.TYPE)
@interface MyAnnotation{
int value1();
String value2();
}
```

Example to specify annoation for a class, methods or fields

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
@interface MyAnnotation{
int value1();
String value2();
}
```

@Retention

@Retention annotation is used to specify to what level annotation will be available.

RetentionPolicy	Availability
RetentionPolicy.SOURCE	refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	refers to the .class file, available to java compiler but not to JVM . It is included in the class file.
RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM .

Example to specify the RetentionPolicy

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyAnnotation{  int value1();  String value2();  }
```

Example of custom annotation: creating, applying and accessing annotation

Let's see the simple example of creating, applying and accessing annotation.

File: Test.java

```
//Creating annotation
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation{
int value();
}

//Applying annotation
class Hello{
@MyAnnotation(value=10)
public void sayHello(){System.out.println("hello annotation");}
}
 //Accessing annotation
class TestCustomAnnotation1{
public static void main(String args[])throws Exception{

Hello h=new Hello();
Method m=h.getClass().getMethod("sayHello");

```

```java
MyAnnotation manno=m.getAnnotation(MyAnnotation.class);
System.out.println("value is: "+manno.value());
}}
```

```
Output:value is: 10
```

How built-in annotaions are used in real scenario?

In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation. Creating and Accessing annotation is performed by the implementation provider. On behalf of the annotation, java compiler or JVM performs some additional operations.

@Inherited

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

```java
@Inherited
@interface ForEveryone { }//Now it will be available to subclass also

@interface ForEveryone { }
class Superclass{}

class Subclass extends Superclass{}
```

@Documented

The @Documented Marks the annotation for inclusion in the documentation.

Generics in Java

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```java
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```java
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Syntax to use generic collection

```
ClassOrInterface<Type>
```

Example to use Generics in java

```
ArrayList<String>
```

Full Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error

String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);

Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

```
Output:element is: jai
       rahul
       jai
```

Example of Java Generics using Map

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

```
import java.util.*;
class TestGenerics2{
public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"vijay");
map.put(4,"umesh");
map.put(2,"ankit");

//Now use Map.Entry for Set and Iterator
Set<Map.Entry<Integer,String>> set=map.entrySet();

Iterator<Map.Entry<Integer,String>> itr=set.iterator();
while(itr.hasNext()){
Map.Entry e=itr.next();//no need to typecast
System.out.println(e.getKey()+" "+e.getValue());
} }}
```

```
Output:1 vijay       2 ankit        4 umesh
```

Generic class

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

Let's see the simple example to create and use the generic class.

Creating generic class:

```
class MyGen<T>{  T obj;  void add(T obj){this.obj=obj;}  T get(){return obj;}  }
```

The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the generic class.

```
class TestGenerics3{
public static void main(String args[]){  MyGen<Integer> m=new MyGen<Integer>();  m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());  }}
```

```
Output:2
```

Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

Generic Method

Like generic class, we can create generic method that can accept any type of argument.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{
   public static < E > void printArray(E[] elements) {
        for ( E element : elements){                      System.out.println(element );           }
         System.out.println();       }
    public static void main( String args[] ) {          Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };
        System.out.println( "Printing Integer Array" );          printArray( intArray  );

       System.out.println( "Printing Character Array" );          printArray( charArray );       }
}
```

```
Output:Printing Integer Array
          10        20        30        40        50
          Printing Character Array
          J       A       V       A       T       P       O       I       N       T
```

Wildcard in Java Generics

The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.

Let's understand it by the example given below:

```
import java.util.*;
abstract class Shape{  abstract void draw();  }
class Rectangle extends Shape{  void draw(){System.out.println("drawing rectangle");}  }
class Circle extends Shape{  void draw(){System.out.println("drawing circle");} }
  class GenericTest{
//creating a method that accepts only child class of Shape
public static void drawShapes(List<? extends Shape> lists){
for(Shape s:lists){  s.draw();//calling method of Shape class by child class instance  }
}
public static void main(String args[]){
List<Rectangle> list1=new ArrayList<Rectangle>();
list1.add(new Rectangle());
  List<Circle> list2=new ArrayList<Circle>();
list2.add(new Circle());
list2.add(new Circle());
drawShapes(list1);  drawShapes(list2);  }}
```

```
drawing rectangle
drawing circle
drawing circle
```