

Core Java

(J2SE)

Read – Understand – Imagine – Think – Apply – Create – Deliver – Update

- OCJP exam oriented approach
- Complete real time approach

Index

s.no	Topic	Page no
1	Introduction to Java Programming	
2	Introduction to Object Oriented Programming	
3	Structure of Java application	
4	Class members	
5	Static members flow using JVM	
6	Non-static members flow Using JVM architecture	
7	Data types	
8	Wrapper classes	
9	Scanner & Random	
10	Command Line Arguments	
11	Access Modifiers	
12	Packages	
13	Object Oriented Programming	
14	This	
15	Inheritance	
16	Super	
17	Final modifier	
18	Abstract classes	
19	Interfaces	
20	Polymorphism	
21	Has-a relation	
22	Exception handling	
23	Multi threading	
24	Garbage collection	
25	Inner classes	
26	IO streams	
27	Working with Files	
28	Reflection	
29	Collections	
30	String Handling	
31	Applets, AWT, Swings	
32	Network programming	

Introduction to Java Programming Language

What is language?

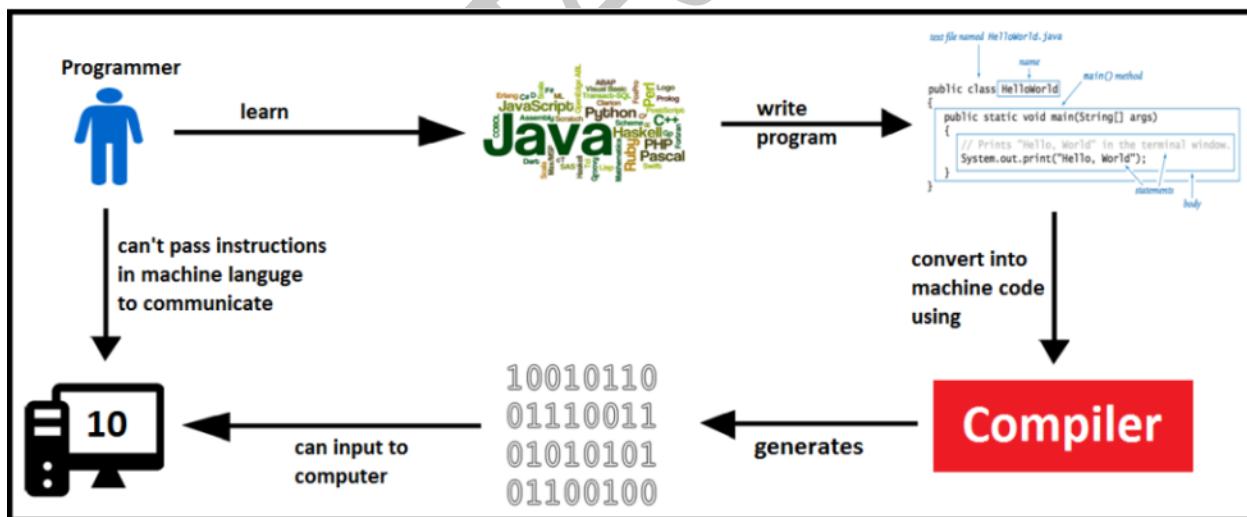
- Language is to communicate.
- Communication is the process of transformation data between objects.
- For this communication, we will use general languages such as English, Telugu, Hindi

What is Programming language?

- It is pre-defined application software.
- It is standalone application.

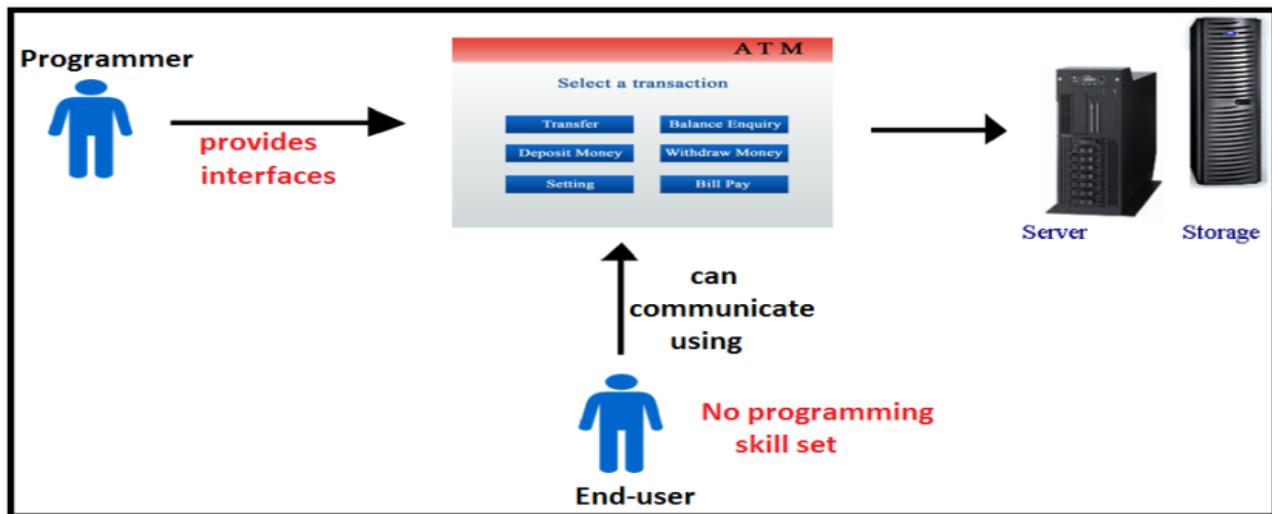
Need of computer language?

1. Communication is possible using a language.
2. We need to communicate with the machine to perform complex operations(finding 120! as shown in diagram) in an easy way
3. Directly we cannot pass machine understandable instructions to communicate.
4. Hence we need to write programs using any high level language(source code)
5. Machine can't understand source code.
6. Compiler is the pre-defined program that translates Source code into Machine code.



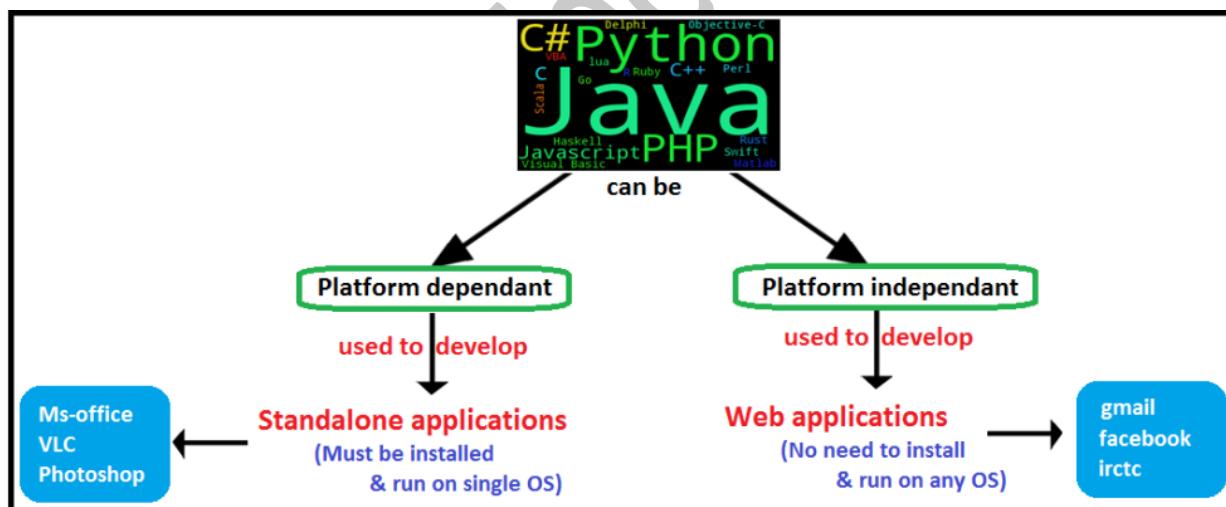
Note:

1. As a programmer, we need to develop applications/interfaces.
2. Using interfaces every End-user can easily interact with the machine without having background knowledge.
3. Interface is nothing but application/software.
4. A screen which provides instructions to end-user to perform the transaction.



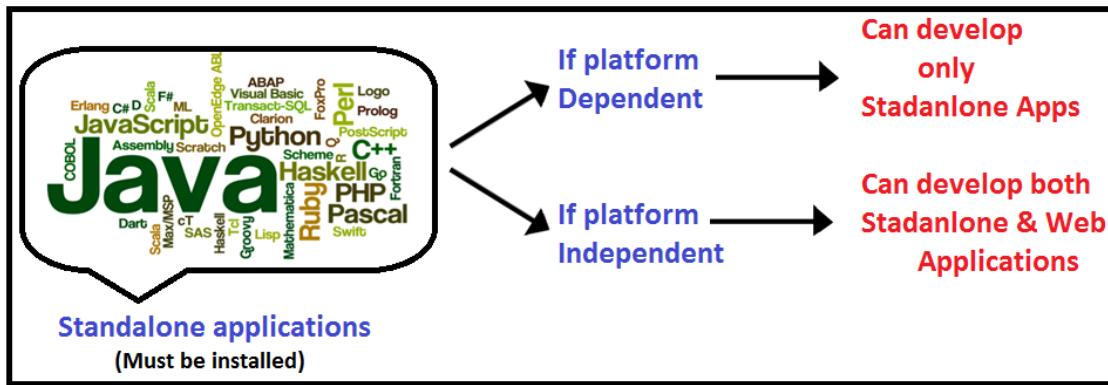
Why Java language instead of C/C++?

- Using C and C++ languages, we can develop only standalone apps.
- standalone app :
 - Runs on a single Operating system.
 - Installation is required on the machine to use the application.
 - For example ms-office, browsers, players.....
- To develop web apps, we must use Java/.net



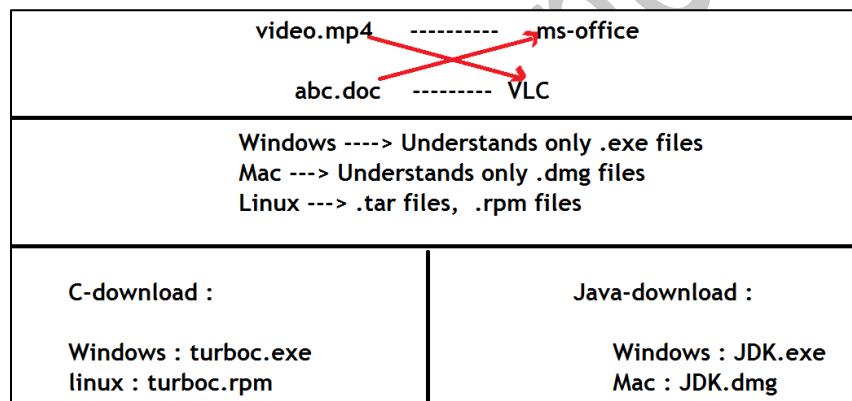
Note:

1. All the programming languages are pre-defined applications.
2. And these applications are standalone.
3. We must install programming language software into machine, dependents on OS.



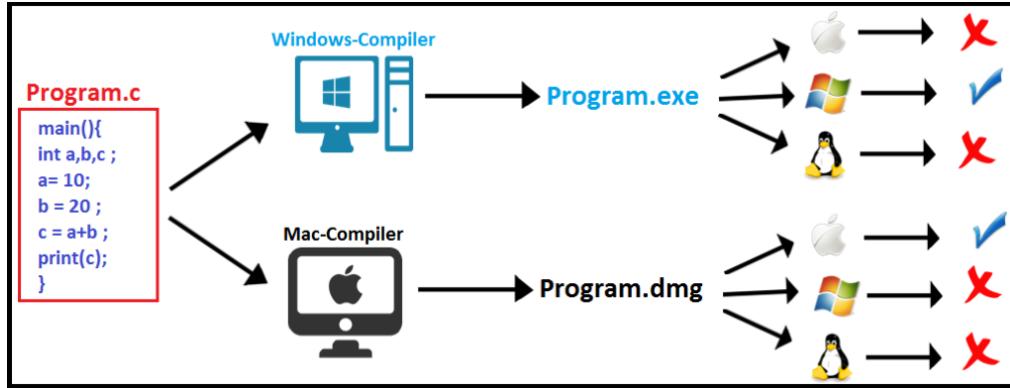
Note the followings:

- Extensions need to be considered to understand the file format
 - As shown below, a video-file cannot be opened by ms-office software.
 - In the same way Different OS's understands different types of file formats.
 - Every programming language is a standalone application.
 - Hence we need to download & install OS-compatible file format to write programs.



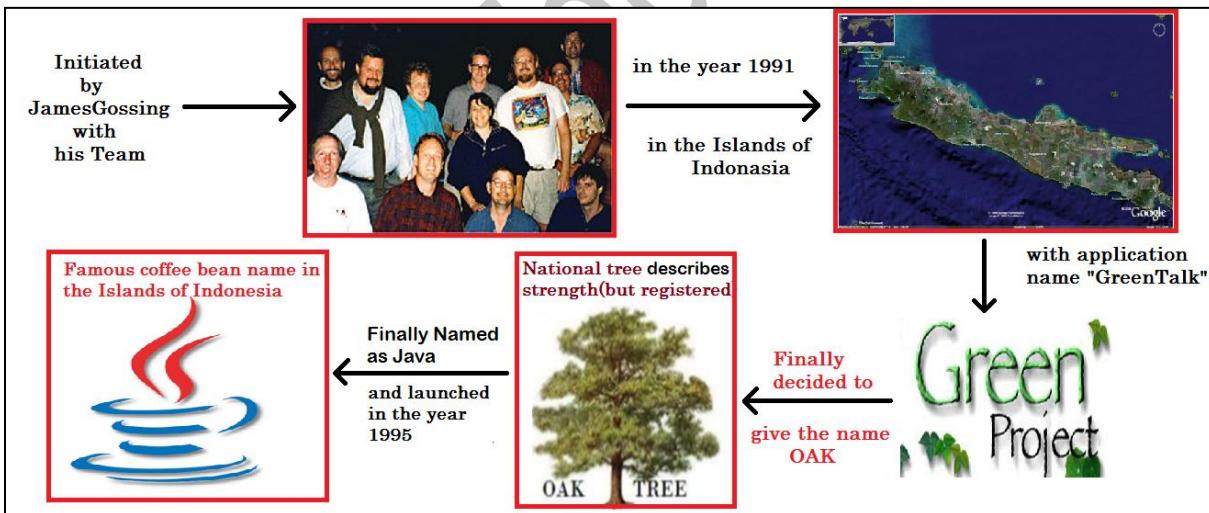
Platform dependency:

- C & C++ languages are platform dependent. Hence we can develop only standalone apps.
 - C-compiler is OS-dependent; hence it translates source code into specific OS understandable instructions.
 - Hence the instructions generated by C-compiler become platform-dependent.



Java History:

- Initiated by James Gosling in the islands of Indonesia, in the year 1991.
- Under the project name “Green Project”
- Decided to launch with the name “OAK”
- “OAK” is the national tree for most of the countries.
- “OAK” is the “symbol of strength”
- As it was registered, finally James, launched the application with the name “Java”
- “Java” is the famous coffee bean in the Indonesia.
- First kit launched in the year 1995.
- JDK 1.0 from Sun Micro Systems. It is the organization of James Gosling.

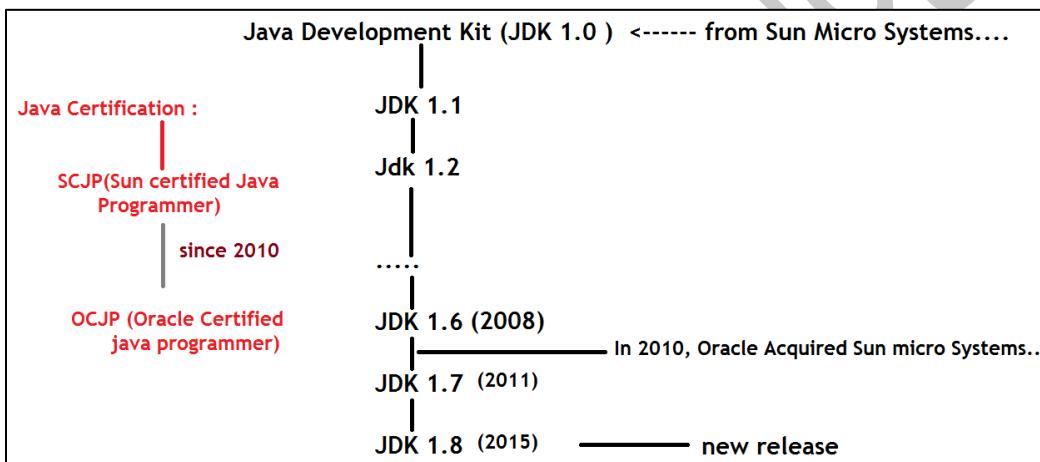


Java versions:

- Following diagram describes release of different versions of Java software.
- Learning these things is not that much important but it is recommended to be good at basic things of Java programming language also.
- First Versions released by “Sun Micro Systems”
- In the year 2010, “Oracle Corporation” acquired “Sun Micro Systems” and the next 2 versions released by “Oracle”

- JDK 1.0 (January 21, 1996)
- JDK 1.1 (February 19, 1997)
- J2SE 1.2 (December 8, 1998)
- J2SE 1.3 (May 8, 2000)
- J2SE 1.4 (February 6, 2002)
- J2SE 5.0 (September 30, 2004)
- Java SE 6 (December 11, 2006)
- Java SE 7 (July 28, 2011)
- Java SE 8 (March 18, 2014)

- In the year 2010, Oracle company acquired Sun micro systems.
- Every organization providing certifications on programming languages.
- Java(J2SE) certification was initially providing by Sun Micro Systems.
- Since 2010, Oracle corporation providing certifications in the same way

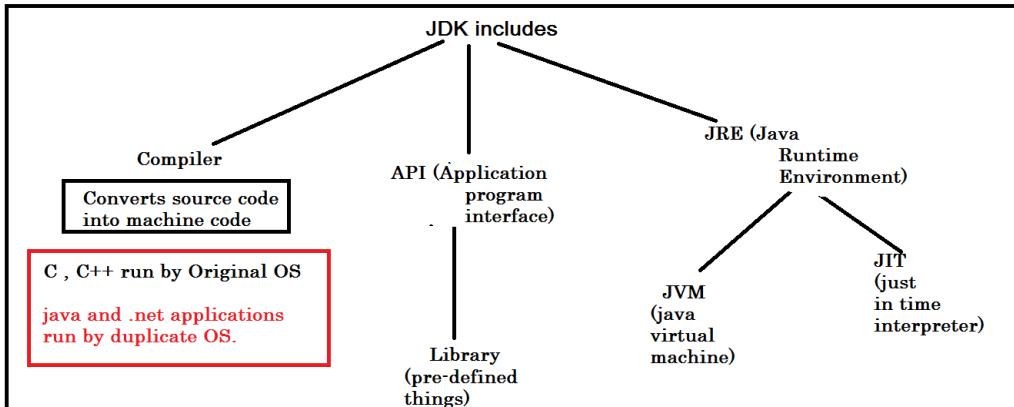


Java v/s .net :

- .net is not the open source technology. It is dependent to Windows OS only. Hence we need to work with .net is more complex and expensive.
- Java is open source technology.
- Most of the organizations (start ups) using Java to develop applications.
- Now days, all the apps ruling the market such as Selenium, Android, Hadoop depends on Java.
- JDK is compatible for all types of Operating systems. Hence while using Java, no need to worry about OS.

Installation of JDK:

The following diagram describes, what are the things will be installed into machine when we install JDK-kit.



API:

- Stands for Application Program Interface.
- Generally referred as Library.
- Interface is nothing but, working with a method or class without knowing the background details.
- Here, we are working with library things without look into code. Hence called as **Program interface**.

JRE:

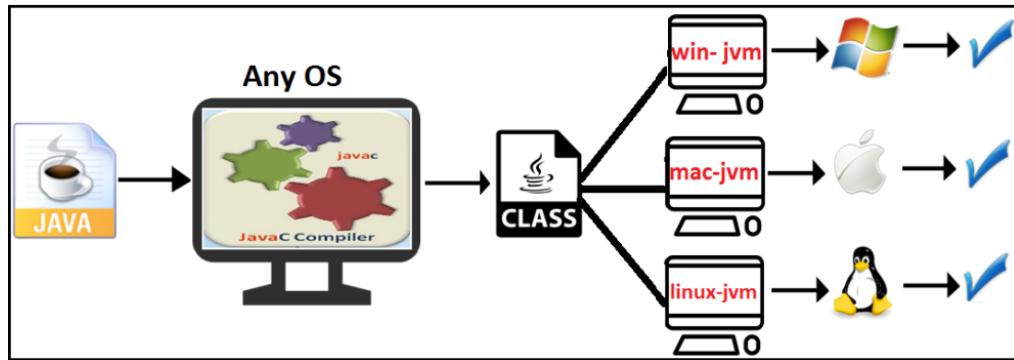
- Stands for Java Runtime Environment.
- To execute any program “Environment setup” is required, for example to take the java class – we need class room environment, to watch a movie – theatre environment
- C and C++ applications run in Original OS environment. But java and .net applications having their own environments called JVM(Java Virtual Machine)....

JIT:

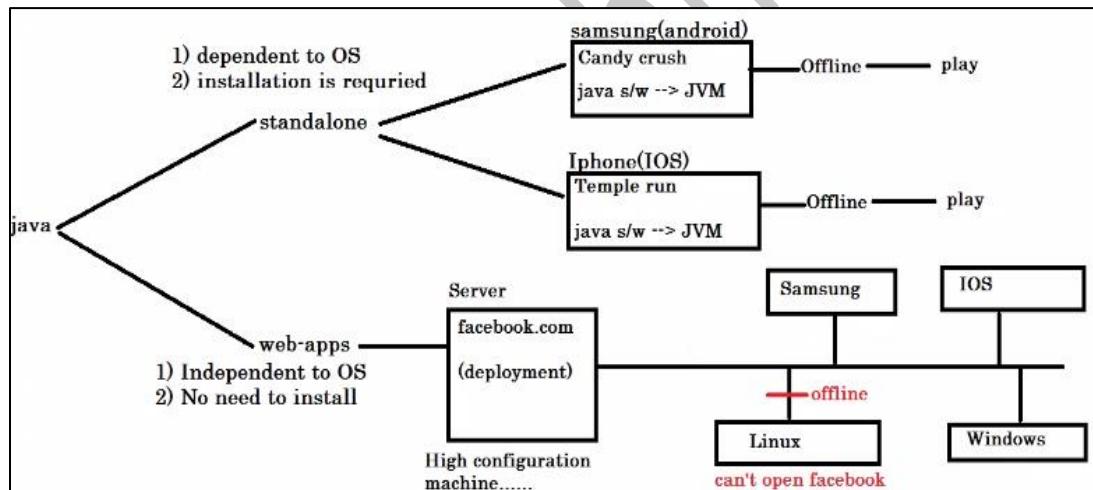
- Stands for Just In-time interpreter.
- Translates JVM understandable code into a specific machine understandable code.
- Works under JVM.
- Works in Virtual Environment.

Platform Independence:

- Java is the Platform independent language.
- Java compiler plays key role in independency.
- C-compiler is platform dependent; hence it generates only a specific OS understandable instruction set.
- Java compiler is dependent to JVM (virtual OS), hence it generates, JVM understandable instructions.
- JVM instructions once again interpreted by JIT into a specific OS understandable instructions.



- Following diagram specifies how standalone app running on a specific OS and Web-app runs on different platforms.
- In case of Standalone app, we need to install application and required software to run that application. For example if we want to play a video in offline mode, we need video file and player.
- But in case of web application, no need to maintain any information in client machine.
- For example if we have data connection, we can play videos (youtube.com) without having file and player in our machine.



Introduction to Object Oriented Programming

- Java is Object Oriented Programming Language.
- Features of OOP are
 - 1) **Encapsulation**
 - 2) **Inheritance**
 - 3) **Abstraction**
 - 4) **Polymorphism**
- These features neither belong to java nor to any other language.
- These are global features. Any language can adopt these features to become OOPL
- Examples.....
C++, java, .net, PHP.....
- In any language, concept of OOP feature is same, but implementation is according to syntax of language,
- For example, I want to print a message in different languages....
C ---> printf("Hello");
C++ ---> cout<<"Hello";
Java ---> System.out.println("Hello");

Note: In any language, we can implement these features using 2 things...

- 1) **Object**
- 2) **class**

Object: A real world entity/substance is called Object & it is having 3 things....

- 1) **Identity / name**
- 2) **Properties / variables**
- 3) **Functionality / methods**

Examples:

Object type: Human

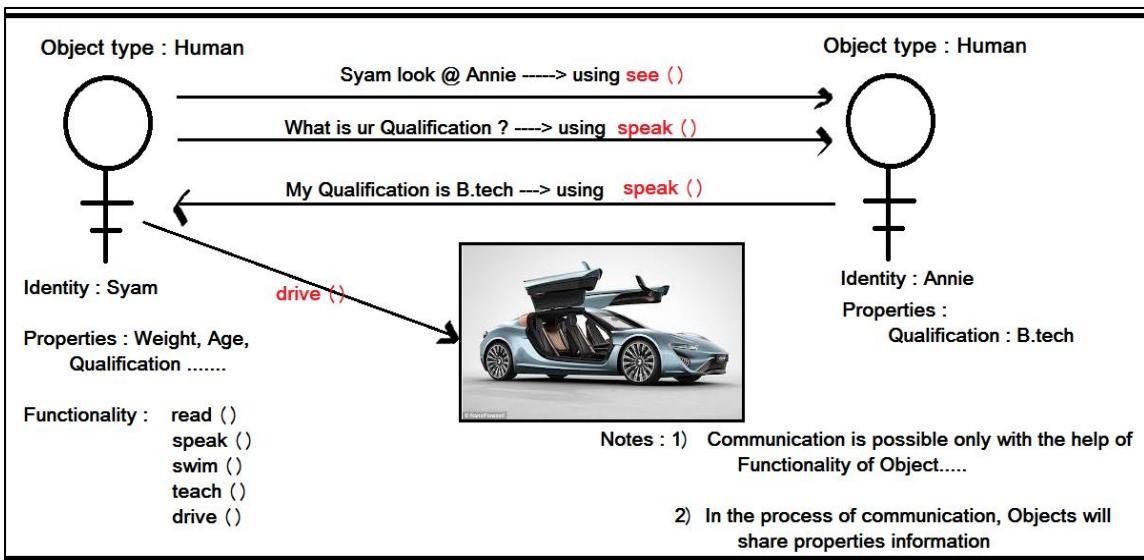
Identity : Shyam
Properties : Color, Height, Weight, Age, Qualification.....
Functionality : read(), see(), walk(), swim(), drive(), teach().....

Object type: Mobile

Identity : Samsung
Properties : Model, price, configuration, color.....
Functionality : call(), message(), browse(), play().....

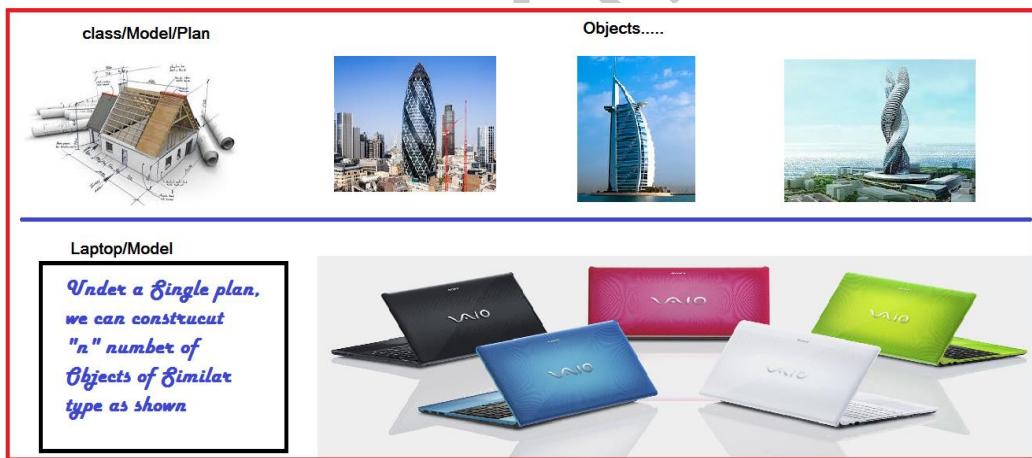
Note:

1. **Objects will participate in the process of communication using functionality (methods).**
2. **In the process of communication, Objects will share properties information.....**



Class:

- Complete representation of real world entity (object).
- Class is a plan/model/blue print of an object.
- Using single model, we can construct "n" number of objects of same type.

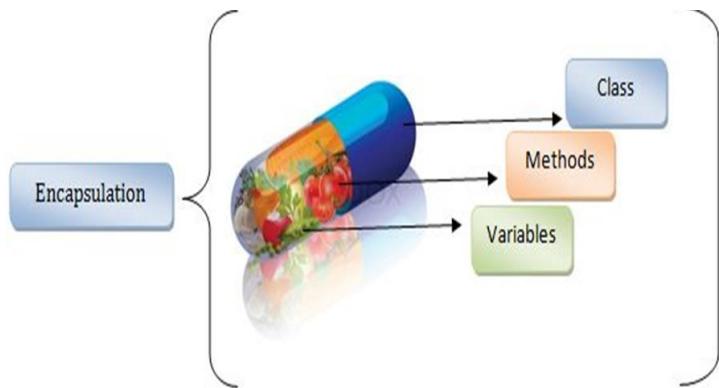


Encapsulation:

- Concept coming from "capsule".
- Whatever the content which is inside the capsule is sensitive, hence protected with cap.
- We need to protect complete information of object.....
- Encapsulation is the concept of protecting object information by placing properties and functionality as a block...

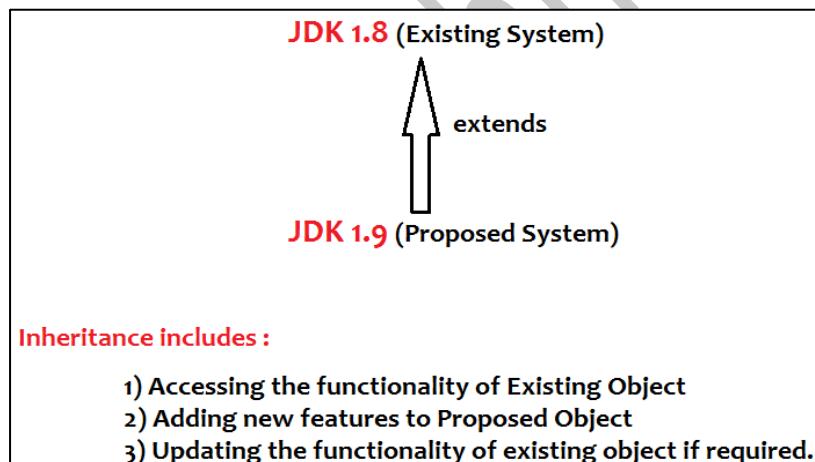
```
class / Encapsulation{
    properties
    +
    methods
}
```

Encapsulation --> theory...
class --> technical implementation in object oriented programming.....



Inheritance:

- Defining new Object with the help of existing object.
- Releasing next version of existing object.....
- **Existing system --> proposed system....**



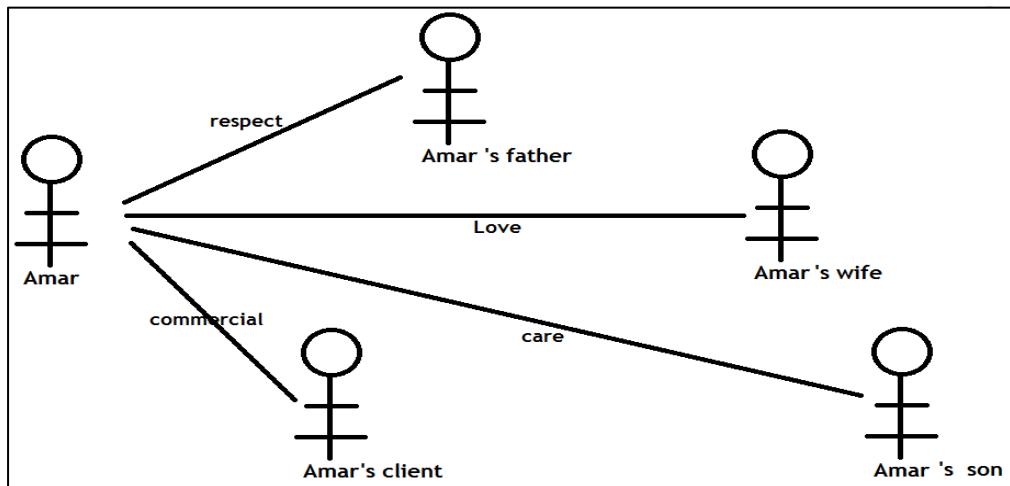
Abstraction:

- Hiding unnecessary details of Object and shows only essential features to communicate.
- Abstraction lets you know how to communicate with object
- Abstraction cannot provide information about what is happening in the background.



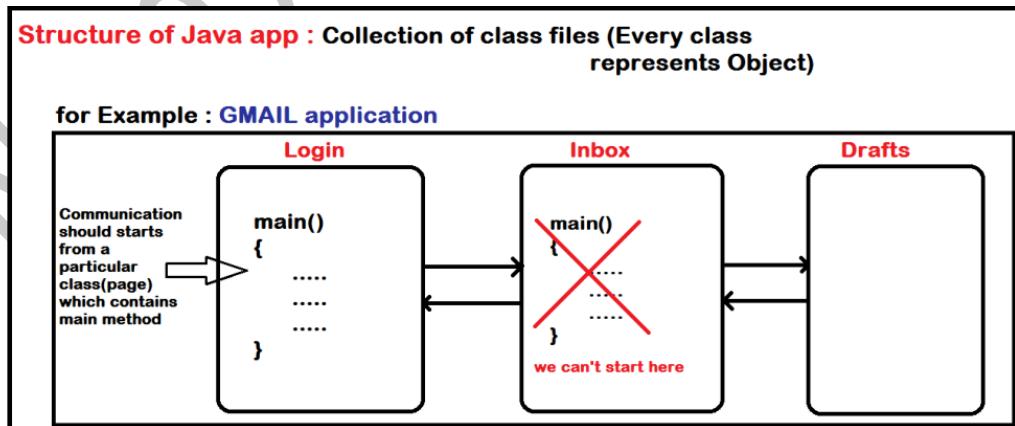
Polymorphism:

- One Object is showing different behavior while communicating with different objects.
- Java supports 2 types of Polymorphism
 - Compile time polymorphism
 - Runtime polymorphism
- Simple example showing in diagram, brief discussion later....



Structure of Java application:

- 1) Every application is a set of files/programs.
- 2) Every java application is a set of class files.
- 3) Every class file is a Model that represents Object.
- 4) JVM will run java application.
- 5) Every application should have a single starting point(main method)
- 6) We need to place main method only inside a single java file in application.



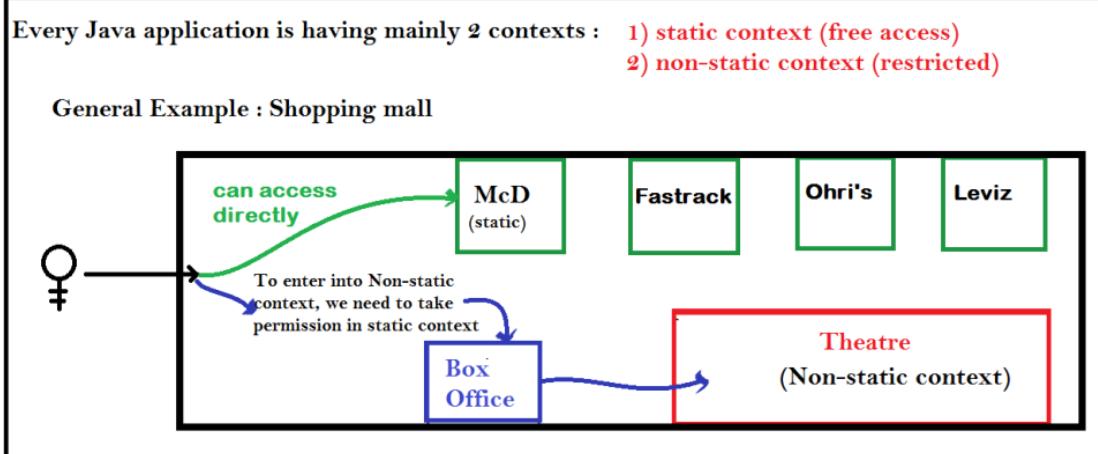
Application contexts:

Every java application is mainly having 2 contexts:

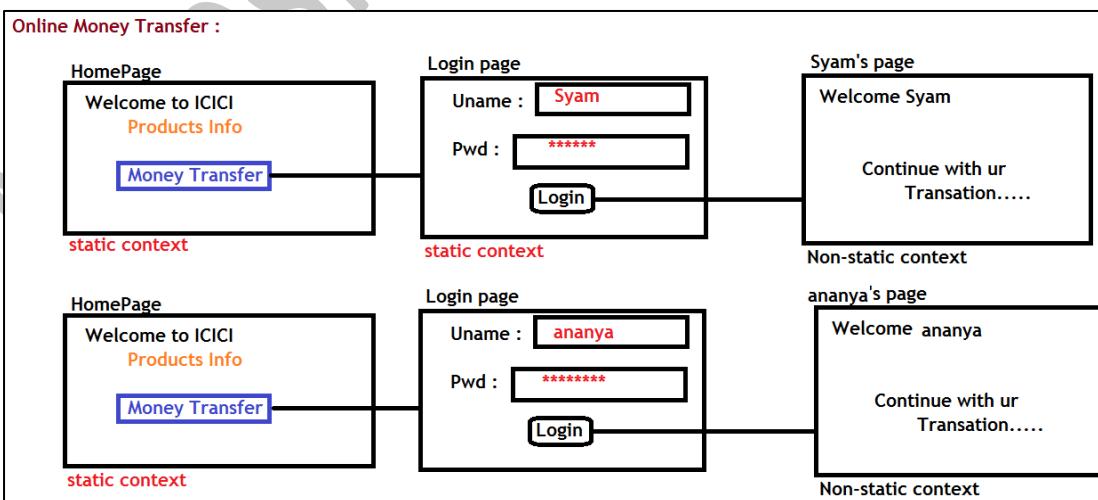
- 1) Static context (free accessible area)
- 2) Non-static context (restricted access/permissions required)

General example:

- Generally static context of Object we can access directly.
- Freely accessible area must be defined as static in java
- Specific functionality of object must be declared as non-static.
- Permissions are required to access non-static area.
- Mostly permissions are available to access non-static area in static only.
- Following example describes clearly.

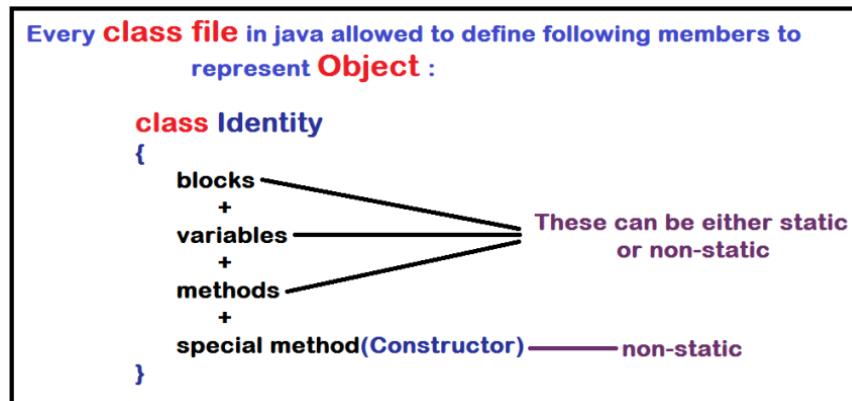


Technical example: The following example represents the technical way, how we are defining and access members of static and non-static.



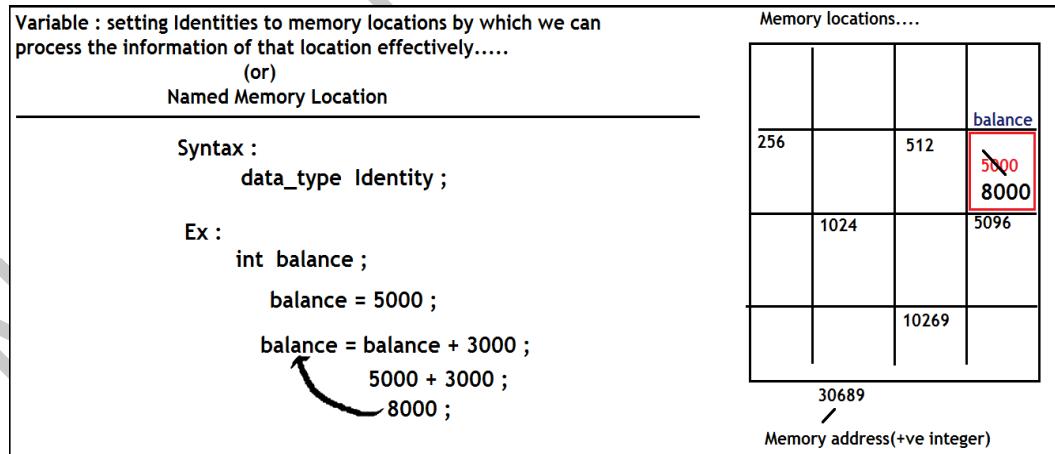
Class members

- 1) In Object oriented programming language, every object is represented by class.
- 2) Encapsulation can be implemented by a class.
- 3) We need to write a class to define object.
- 4) Every instruction in java application must be placed inside the class.
- 5) A class is allowed to defined following members inside a class.



Variable:

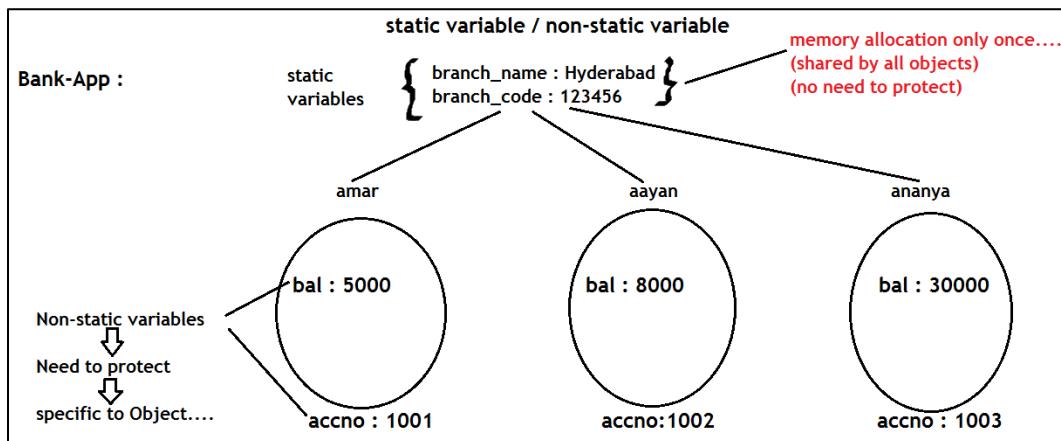
- 1) Providing identities to standard memory locations.
- 2) Without identities we cannot access the data once it has been stored.
- 3) Named memory location.
- 4) Using variables we can process the information effectively.



Static variables v/s Non-static variables:

- 1) As a programmer first we need to know whether the variable need to be declared as static or non-static.
- 2) Variables which are common for all the objects must be declared as static.
- 3) Variables which are specific to Object must be defined as non-static
- 4) Static variable get memory allocation only once.

- 5) Non-static variables get memory allocation inside every object.
- 6) Look at the example to get clarity. We discuss briefly about static variables & non-static variables in coming sessions.



Block:

- 1) Set of Instructions having no identity.....
- 2) We cannot call explicitly (because of no identity)....
- 3) JVM invokes implicitly every block in the process of application execution.....
- 4) Block providing basic information to start communicates with object.
- 5) General blocks such as while, for, if... used to define logical programming. These blocks also cannot be called explicitly in the program.

General blocks in Every programming language used to implement Logical programming.....	Java language is having 2 blocks along with general blocks.... 1) static block 2) non-static block...	
<pre>if() { }</pre> <pre>for() { }</pre> <pre>while() { }</pre>	static block syntax : static Identity { statement-1; statement-n; }	non-static block syntax: { statement-1; statement-n; }

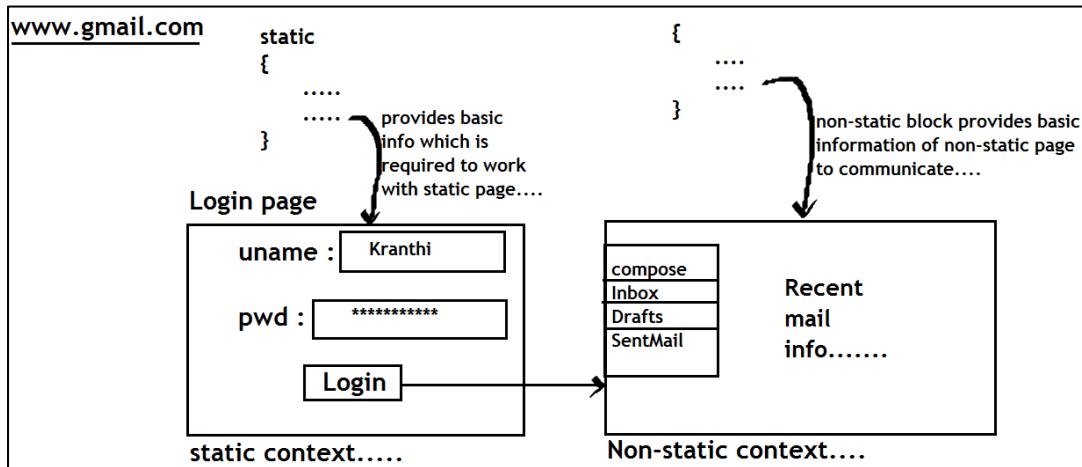
Questions:

1. What is the use of block?
2. What type of information need to place inside the block?
3. When block get executes?

Answer:

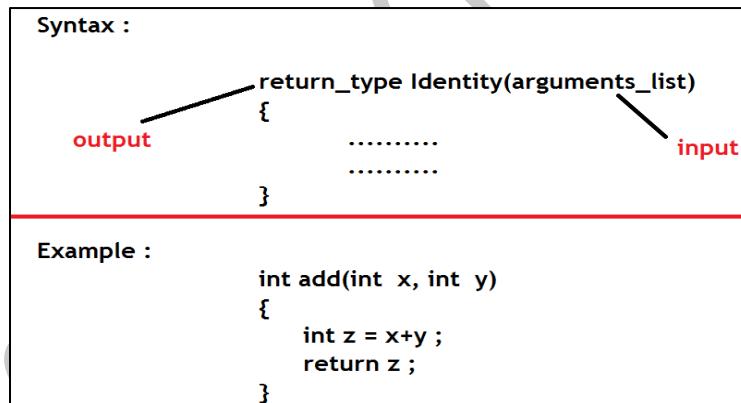
- To start communicating with any object, basic information is required.

- Static block & non-static blocks are responsible for providing basic info of object
- We can communicate with any object with the available information only.



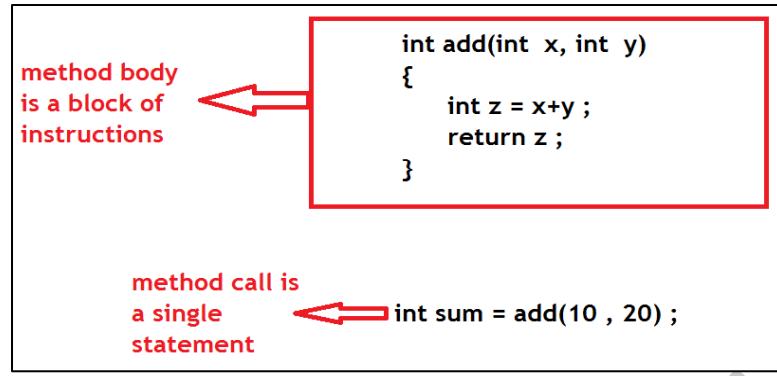
Method:

1. A block of instructions having Identity.
2. Every method is taking input, processing input and returns output.
3. Every method must be called explicitly (using its identity).



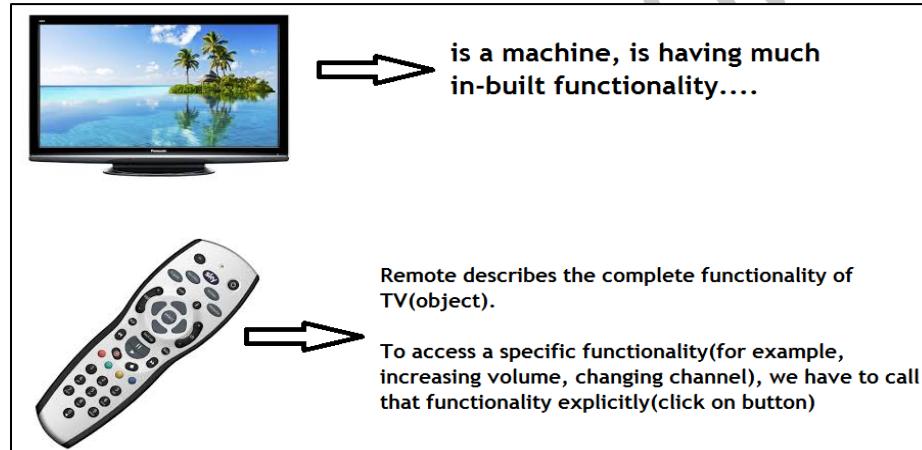
Note : Every method is having 2 main things.

- 1) Method definition → block
- 2) Method call → single statement



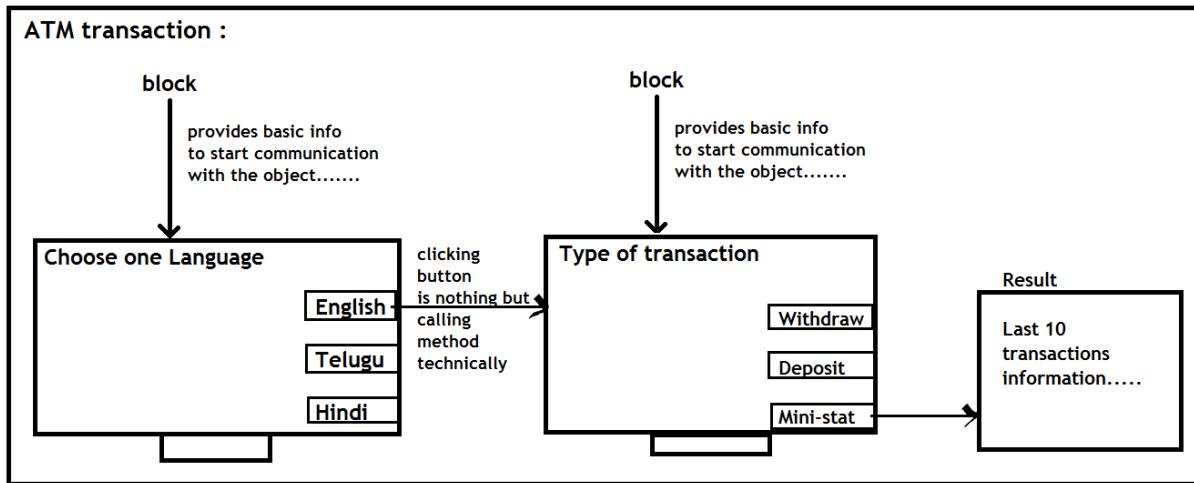
Why we need to call a method explicitly?

- Look at the example shown in the diagram.
- Every electronic device (object) having much inbuilt functionality.
- But the functionality of any object will not execute automatically(tv will not be controlled itself)
- Another object (controller) is required to access functionality of any object.



Block v/s method:

Block	Method
Block of instructions	Block of Instructions
No identity	Having identity
Can't take input	Can take input
Can't produce output	Can produce output
Can't call explicitly	Must be called explicitly



Classification of Methods:

- 1) No arguments and No return values
- 2) With arguments and No return values
- 3) With arguments and With return values
- 4) No arguments and With return values

No args & No return values <code>void fun(void) { } fun() ; ---> calling</code>	With args & No return values <code>void fun(int x) { } fun(10) ;</code>
With args & With return values <code>int fun(char x) { return 13 ; } int y = fun('g') ;</code>	No args & with return values <code>float fun(void) { return 34.56 ; } float x = fun() ;</code>

First Java application:

- Java programming language is case sensitive.
- Look at the capital letters in the following program.
- Some of the Naming conventions (discussion later) we need to follow to write programs.

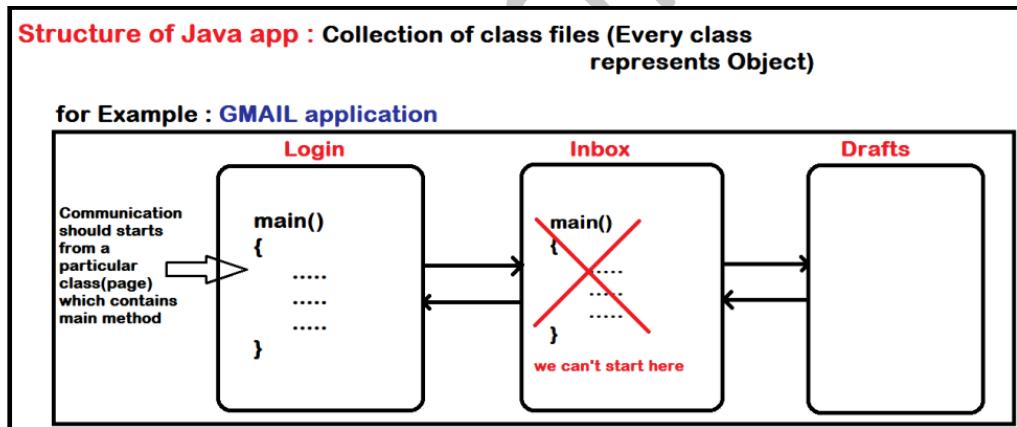
```
class FirstApplication
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

class:

- Java application is nothing but creating objects to communicate.
 - To define object, model (class) is required.
 - Hence in java application every instruction must be placed inside the class.
 - Using class, we are implementing Encapsulation (protect).

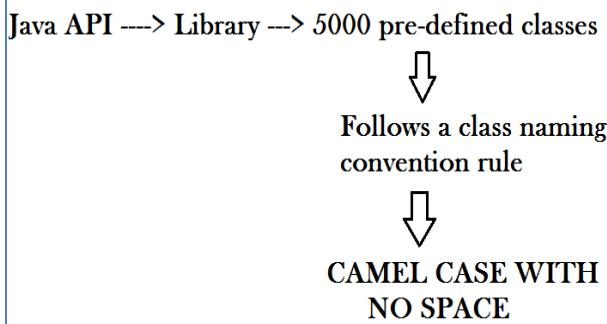
class Identity: (FirstApplication)

- Every application is a set of class files.
 - In java application, every class should be uniquely identified.
 - JVM executes a specific class using its identity only.



Question: Why java class name starts with capital letter ?

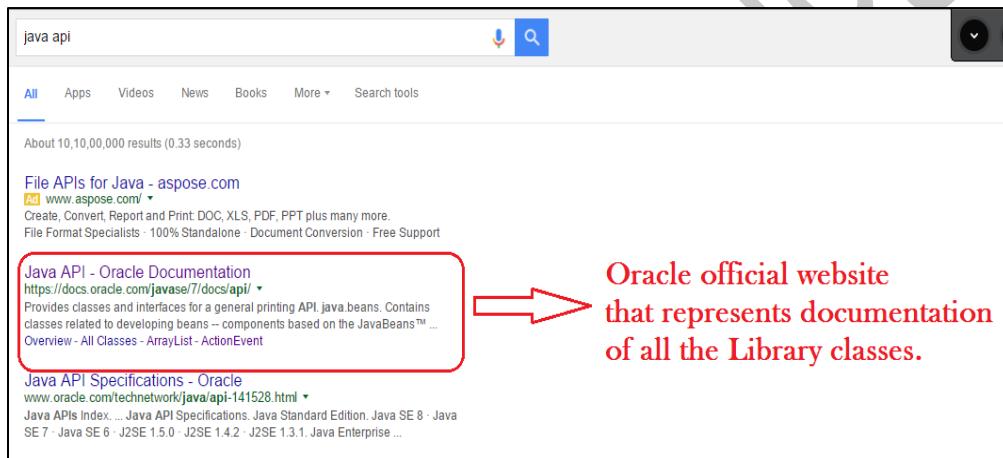
- Java API is having thousands of pre-defined classes
 - All these classes follow a naming convention rule “CAMEL CASE WITH NO SPACE”.
 - In case of pre-defined class, we must follow this rule.
 - In case of user defined class, naming convention rule is optional.
 - We have to face problems if we violate the convention rules while writing application.



Examples :

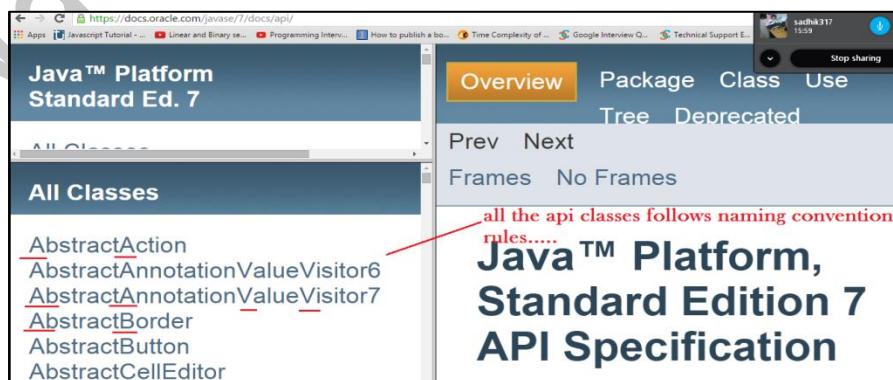
System
PrintStream
NullPointerException
FileNotFoundException

- Java Library includes definition of around 5000 pre-defined classes.
- To use the functionality of these classes, first we have to study API.
- Description of these classes will not be available along with JDK installation.
- **Oracle Company** provided documentation about these classes as an OFFICIAL WEBSITE



Note:

- Every java developer should maintain this documentation to develop java applications.
- Instead of searching the documents online, free downloads available for basic java developers.
- Instead of maintain data connection (internet) we can go for website downloads.

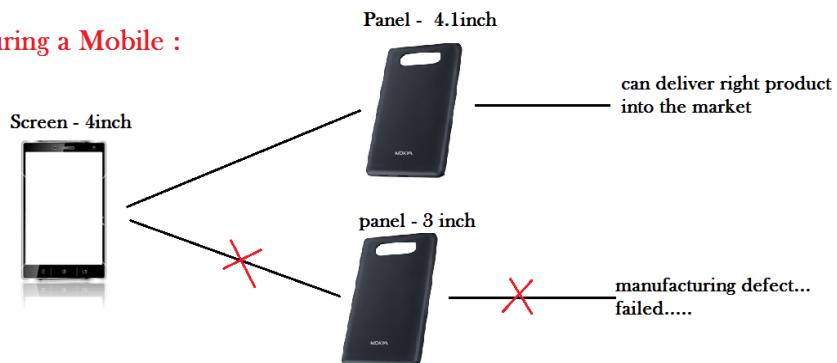


Question: What are the problems when we don't follow the rules of naming conventions?

- Application development is not possible by a single programmer.
- We need to convert an application into modules and each module developed by different programmer.
- While sharing, every developer must follow set of rules.

1) One application is not possible to develop by a single programmer.
2) When we convert an application into modules, all the programmers must follow a set of rules, then only they generate exact product into the market....

Manufacturing a Mobile :



Question: Why main method is public & static?

- Non-static functionality is specific to particular object.
- Static functionality is common for all the objects in application.
- Main method is the common(static) starting point of all(public) the applications. Hence it is public static.

Non-Static : Specific to only 1 object (accno , balance)

Static : Common to more than 1 object (IFSC , bank_name)

private package protected public
<10 <100 <1000 All

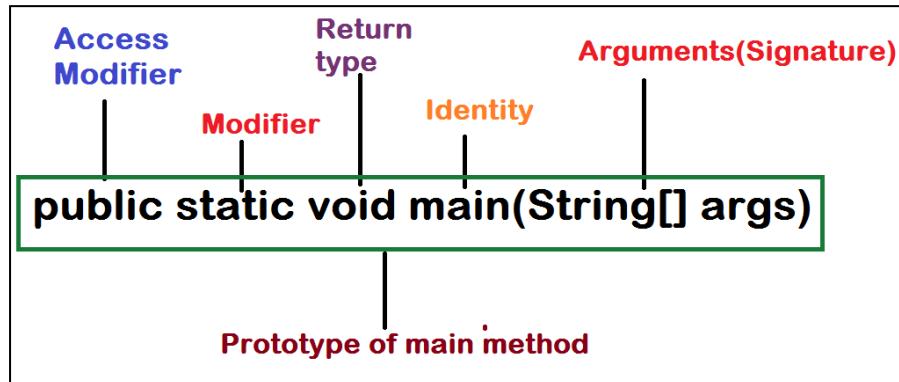
Question: Why main method return type is void?

- JVM invokes main() method to start application execution.
- main() method is not returning anything to JVM, Hence it is void.
- Method naming convention “MIXED CASE & NO SPACE”
- **Examples :**

```
next()  
println()  
nextFloat();  
getAccountNumber();
```

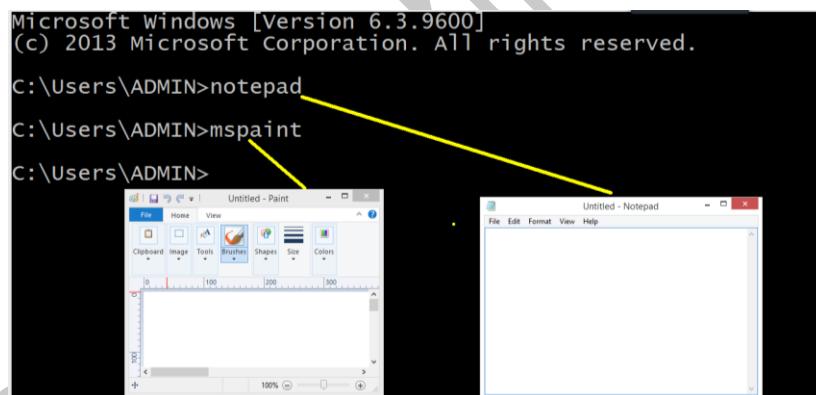
```
getAccountHolderName();  
getAccountHolderHouseAddress();
```

- In case of pre-defined methods, we must follow convention rules.
- In case of user-defined methods, these rules are optional (but recommended to follow)

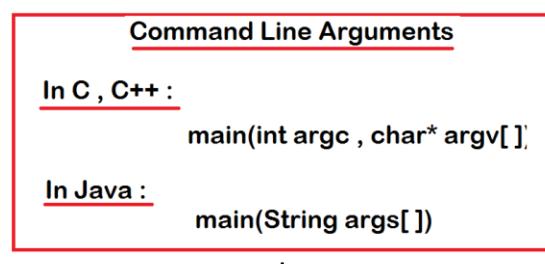


Signature of main method:

- List of arguments is called signature.
- Generally all the programs in a computer execute in OS environment (DOS).



- C, C++ applications execute in the OS environment.
- Java application also compile and run from OS environment **with the help of JVM** (command prompt)
- While executing java application, we can pass input to program from command line is called command line arguments.



- Main() method is responsible for collecting these arguments and store into an array of type String.
- Hence main() method signature is
 - main(String args[])

How to install JDK kit:

Open any browser and type “jdk8 download” in search box.



Click on first result (URL) to open website.

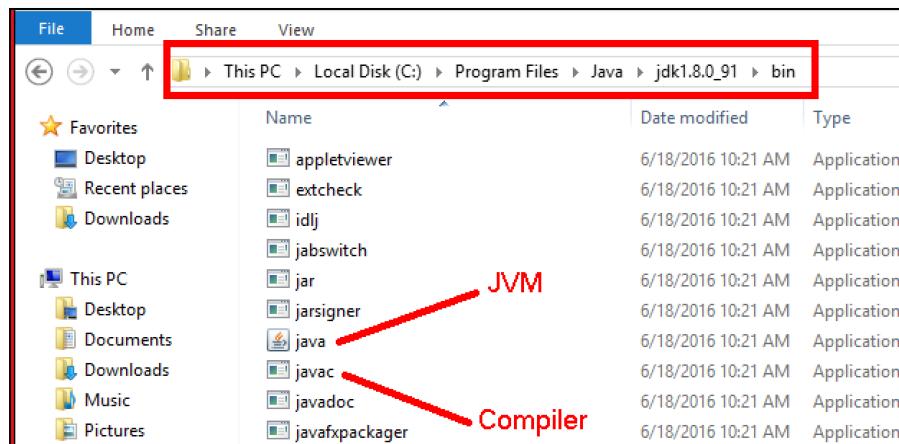
The screenshot shows the "Java SE Development Kit 8 - Downloads - Oracle" page. The title is "Java SE Development Kit 8 - Downloads - Oracle" and the URL is "www.oracle.com > Java > Java SE". Below the title, there is a "Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™)." message. The page content includes a table of available Java distributions for different platforms and architectures.

Accept the license agreement and download the jdk kit depends on OS you are using...

Java SE Development Kit 8u111			
You must accept the Oracle Binary Code License Agreement for Java SE to download this software.			
Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.			
linux	Product / File Description	File Size	Download
	Linux ARM 32 Hard Float ABI	77.78 MB	jdk-8u111-linux-arm32-vfp-hf.tar.gz
	Linux ARM 64 Hard Float ABI	74.73 MB	jdk-8u111-linux-arm64-vfp-hf.tar.gz
	Linux x86	160.35 MB	jdk-8u111-linux-i586.rpm
	Linux x86	175.04 MB	jdk-8u111-linux-i586.tar.gz
	Linux x64	158.35 MB	jdk-8u111-linux-x64.rpm
	Linux x64	173.04 MB	jdk-8u111-linux-x64.tar.gz
mac	Mac OS X	227.39 MB	jdk-8u111-macosx-x64.dmg
	Solaris SPARC 64-bit	151.92 MB	jdk-8u111-solaris-sparcv9.tar.gz
	Solaris SPARC 64-bit	93.02 MB	jdk-8u111-solaris-sparcv9.tar.gz
	Solaris x64	140.38 MB	jdk-8u111-solaris-x64.tar.gz
	Solaris x64	96.82 MB	jdk-8u111-solaris-x64.tar.gz
32 bit	Windows x86	189.22 MB	jdk-8u111-windows-i586.exe
64bit	Windows x64	194.64 MB	jdk-8u111-windows-x64.exe

- Install the software by following the steps as shown in installation wizard.
- After installation, java folder will be extracted in the specified path.

- We need to set path to the “bin” folder which contains the entire related .exe file such as compiler and JVM to compile and run java applications.

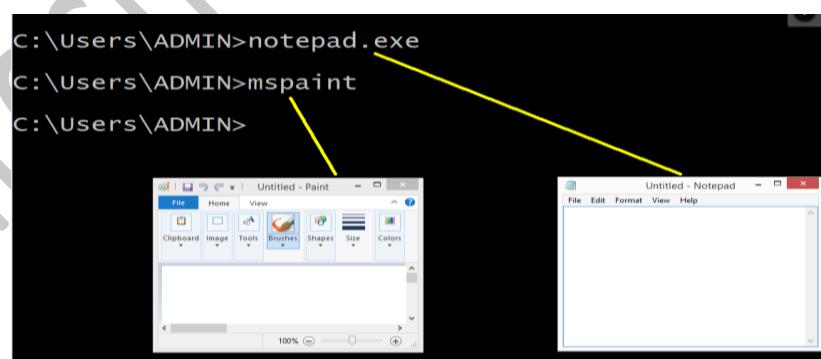


Steps to set path:

- Copy exe files folder path as shown in diagram.
- Right click on my computer.
- Select properties option
- Click on “Advanced System settings” in the opened window.
- Click on “Environment variables”.

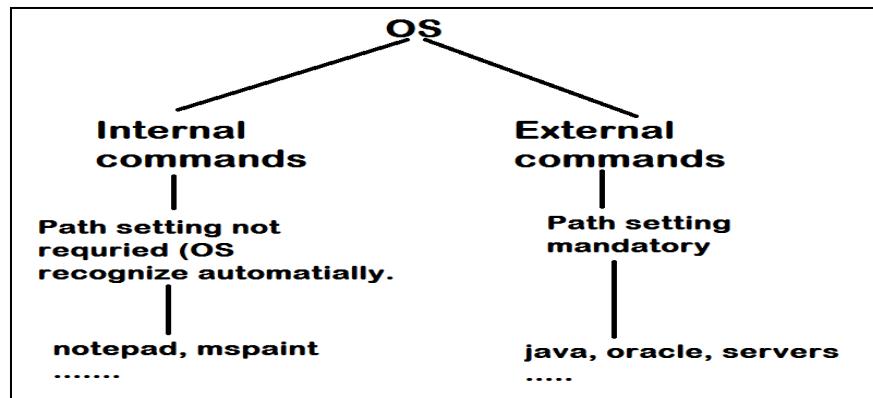
Why we need to set path to use these executable files?

- Generally OS, runs every application in Machine.
- When we install OS into machine we can use both GUI and CUI interfaces to use the functionality of OS.
- CUI interface is nothing but DOS as shown in diagram.
- It can run any program/application which has installed.



OS understands 2 types of commands.

- 1) Internal commands (Applications installed along with OS)
- 2) External commands (Applications not installed along with OS)

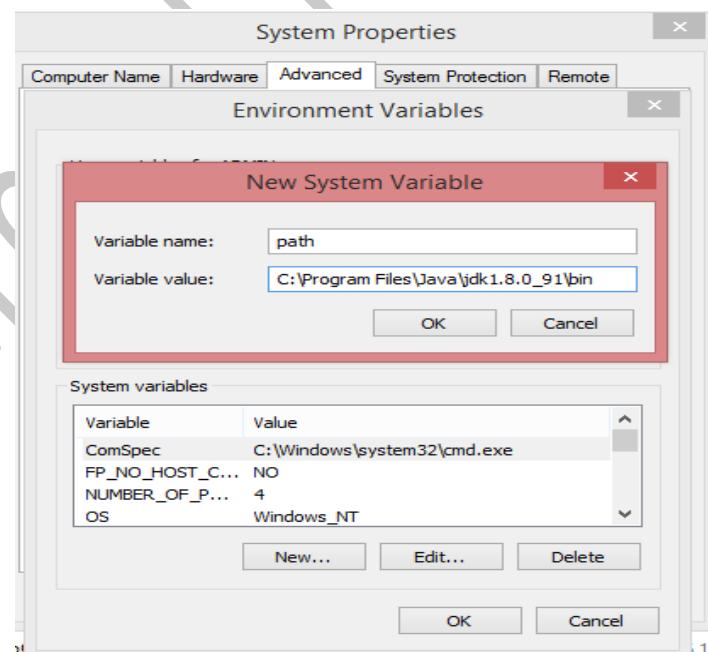


- OS is called Environment in which applications/programs execute.
- Path setting is nothing but creating Environment variable by which we can provide information to OS about installed applications.

Path = "C:\Program Files\Java\jdk1.8.0_91\bin"

How to set path:

- Right click on “MyComputer” icon . Select “Properties” option
- Select “Advanced System Settings” option. Click on “Environment Variables” button
- Use the “System variables area”. Look for the “path” variable
- If present, edit the variable value and the new path value at the end. Values separated with semi colon symbol.
- If path not present, create new variable as shown below.



Static members flow using JVM architecture

- Common functionality of Java application must be defined as static.
- Java supports 4 static members.
 - Static main method
 - Static block
 - Static user method
 - Static variable

In java we can save java source file with any name.

Program.java :

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

- Open command prompt
- Move to file location area.

```
D:\>javac Program.java
D:\>java Test
Hello World!
```

```
class Test
{
    // empty ....
}
```

- Compiler takes care of syntactical errors only.
- Compiler never checks main method is present or not.
- JVM invokes main method to start application execution.

```
D:\>javac Program.java compilation
D:\>java Test Execute
Error: Main method not found in class Test, please main method as:
    public static void main(String[] args)
or a JavaFX application class must extend javafx.
.Application
D:\> Error : No main method to start
application.
```

```
class Test
{
    public static void main(String arg[ ])
    {
        // empty.....
    }
}
```

```
D:\>javac Program.java
D:\>java Test
D:\>_
```

- `System.out.println()` is working like debugging statement.
- It is used to analyze the program using user messages.

```
class Test
{
    public static void main(String arg[ ])
    {
        System.out.println("Main method....");
    }
}
```

```
C:\Windows\system32\cmd.exe
D:\>javac Program.java
D:\>java Test
Main method.....
D:\>
```

static block :

- Block of statements having no identity.
- We cannot invoke static block(no identity).
- JVM invokes the static block implicitly at the time of class loading.
- Static block is optional.
- Static block executes before main method.

```
class Test
{
    static
    {
        System.out.println("Static block....");
    }
    public static void main(String arg[])
    {
        System.out.println("Main method....");
    }
}
```

```
C:\Windows\system32\cmd.exe
D:\>javac Program.java
D:\>java Test
Static block....
Main method.....
D:\>
```

```
class Test
{
    public static void main(String arg[])
    {
```

```

        System.out.println("Main method....");
    }
static
{
    System.out.println("Static block....");
}
}

```

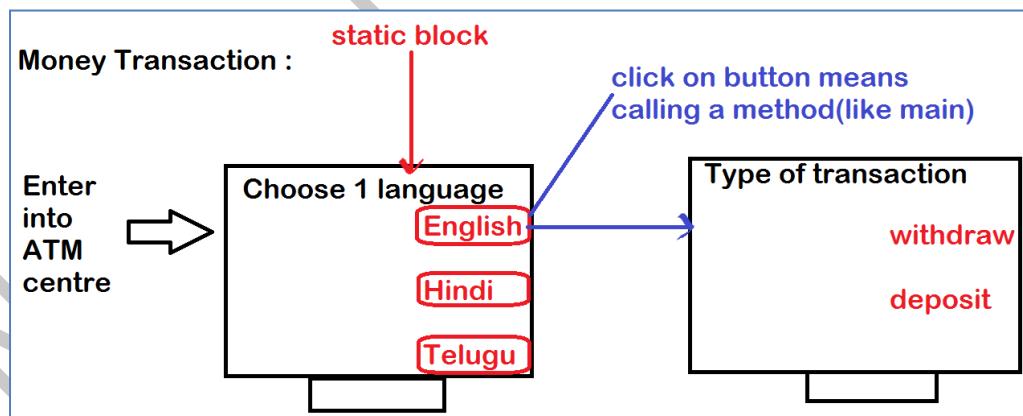
```

C:\Windows\system32\cmd.exe
D:\>javac Program.java
D:\>java Test
Static block....
Main method....
D:\>

```

Question: Why static block execute before the main method?

- Static block provides basic information to start communication.
- Communication is possible using methods (main method).
- Without basic information, we cannot communicate.
- Hence static block execute before main method.



Question : Can we define more than one static block?

Answer : Yes allowed. And all these blocks execute in the defined order. Two static blocks having equal priority.

class Test

```

{
    static
    {
        System.out.println("Static block1");
    }
    public static void main(String arg[ ])
    {
        System.out.println("Main method....");
    }
    static
    {
        System.out.println("Static block2");
    }
}

```

```

C:\Windows\system32\cmd.exe
D:\>javac Program.java
D:\>java Test
Static block1
Static block2
Main method....

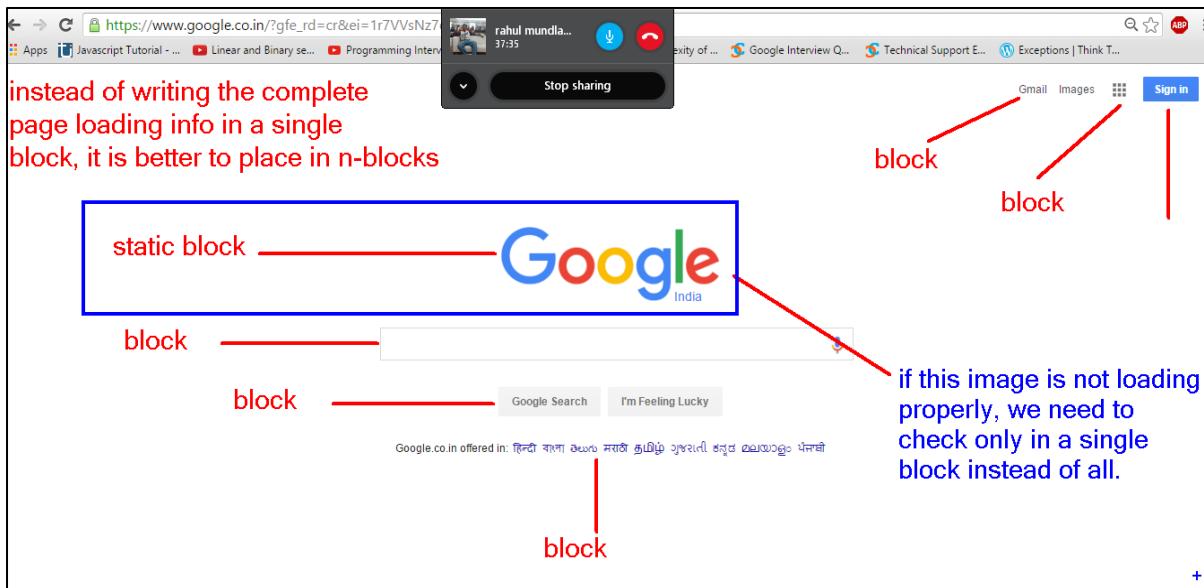
```

Question : When we use more than one static block in the application?

Answer : If class loading info(basic info) is lengthy, we divide into modules(static blocks).

Modularity:

- Dividing logic into blocks
- For example in Bank application divides into modules like “savings account”, car loan, home loan, current account.....
- Advantages are
 - We can share the work
 - Easy to integrate
 - Easy to debug (error).



Static user method:

- A block of instructions having identity
- Method is taking input, processing input & producing output.
- We must call every method(except main) explicitly using its Identity.

```

<static> return_type Identity(arguments / signature)
{
    .....
    processing logic
}

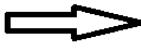
<static> int add(int x , int y)
{
    int z = x+y ;
    return z ;
}

```

- Every method should have return type (at least void)
- If we don't call, method body never executes in the application.



is a machine, is having much in-built functionality....



Remote describes the complete functionality of TV(object).

To access a specific functionality(for example, increasing volume, changing channel), we have to call that functionality explicitly(click on button)

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Main method");
    }

    static void fun()
    {
        System.out.println("User method....");
    }
}
```

```
D:\>javac Test.java
D:\>java Test
Main method
```

Question : How can we access static members in Java application?

Ans : Using class name.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Main method starts....");
        Test.fun(); // method call --> single statement
        System.out.println("Main method ends....");
    }

    static void fun() // method definition --> block
    {
        System.out.println("User method....");
    }
}
```

```
D:\>javac Test.java

D:\>java Test
Main method starts....
User method....
Main method ends....
```

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("main starts");
        Test.fun();
        System.out.println("ctrl back to main method....");
    }

    static void fun()
    {
        System.out.println("ctrl in fun");
    }
}
```

```

static
{
    System.out.println("block starts");
    Test.fun();
    System.out.println("ctrl back to block from fun...");
}
}

class Test
{
    static void f1()
    {
        System.out.println("ctrl in f1....");
    }

    public static void main(String[] args)
    {
        System.out.println("main method starts...");
        System.out.println("Calling f2 method...");
        Test.f2();
        System.out.println("Control back to main from f2 method..");
        System.out.println("Main ends.....");
    }

    static
    {
        System.out.println("static block starts....");
        System.out.println("Calling f1 method...");
        Test.f1();
        System.out.println("control back to static block from f1");
        System.out.println("static block ends....");
    }

    static void f2()
    {
        System.out.println("ctrl in f2 ....");
    }
}

```

```

        System.out.println("Calling f1 method....");
        Test.f1();
        System.out.println("Control back to f2 from f1");
        System.out.println("f2 method ends....");
    }
}

```

```

D:\>java Test
static block starts....
Calling f1 method...
ctrl in f1....
control back to static block from f1
static block ends....
main method starts...
Calling f2 method...
ctrl in f2 .....
Calling f1 method....
ctrl in f1....
Control back to f2 from f1
f2 method ends....
Control back to main from f2 method..
Main ends.....

```

Local variable:

- Declaration of variable inside the block or method.
- Local variable must be initialized before its use.
- Local variables we can access directly.

```

class Test
{
    public static void main(String[] args)
    {
        int a ; // local variable
        // System.out.println(a); // Error : Not yet initialized...

        a = 10 ;
        System.out.println(a);
    }
}

```

- Local variables can be accessed only within the block or method in which it has defined...

```
class Test
{
    static
    {
        int a = 10 ; // local
        System.out.println(a); // allowed to access
    }
    public static void main(String[] args)
    {
        // System.out.println(a); // Error : undefined symbol
    }
}
```

Note : Local variables cannot be static.

Scope of local variable only within the block or method in which it has defined.

Static variables available throughout the application.

JVM architecture:

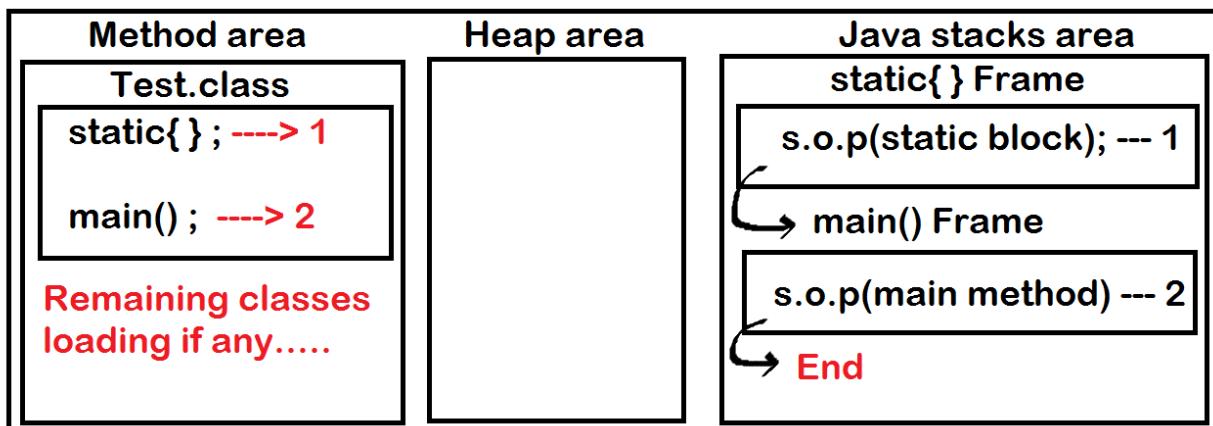
- Every java application run by JVM
- JVM is having many areas (mainly 3) to run java application.
- Generally every application will be processed in main memory (RAM).
- As soon as application has invoked, processor allocates memory to JVM inside RAM memory.
- Once application has been completed, the memory will be destroyed.

JVM architecture

Method area	Heap area	Java stacks area
<p>1) Java application is a set of class files.</p> <p>2) Required class files to run application will be loaded into method area.</p> <p>3) All the static variables get memory allocation.</p>	<p>1) Objects creation area.</p> <p>2) Restricted access.</p> <p>3) Non-static variables get memory allocation inside heap.</p>	<p>1) Blocks & Methods execution area.</p> <p>2) The amount of memory allocated to execute a block is called FRAME.</p> <p>3) Local variables get memory allocation inside the block or method.</p>

```
class Test
{
    static
    {
        System.out.println("static block....");
    }
    public static void main(String[] args)
    {
        System.out.println("main method....");
    }
}
```

JVM flow



Static variable:

- Declaration of variable inside a class and outside to all the methods and blocks.
- We can initialize static variable directly.

int a ; ----> Declaration
a = 10 ; ---> Assignment
a = a + 20 ; ----> Modify / Update
int b = 50 ; ----> Initialization

Question: Why it is allowed to initialize a static variable directly?

Answer:

Example of static variable in bank_app → bank_name;
Non-static variable in bank_app → accno;

- Consider, in bank application development as a programmer we can decide(provide) values to static variable.
 - static String bank_name = "ICICI";
- But we can't decide the value of non-static variable at the time of application development.
 - int acc_no;
- **static variables we can access using class name.**

```
class Test
{
    static int a = 100 ;
    static
    {
        System.out.println("Static block");
        System.out.println(Test.a);
    }
    public static void main(String[] args)
    {
        System.out.println("Main method");
        System.out.println(Test.a);
    }
}
```

}

Program execution steps:

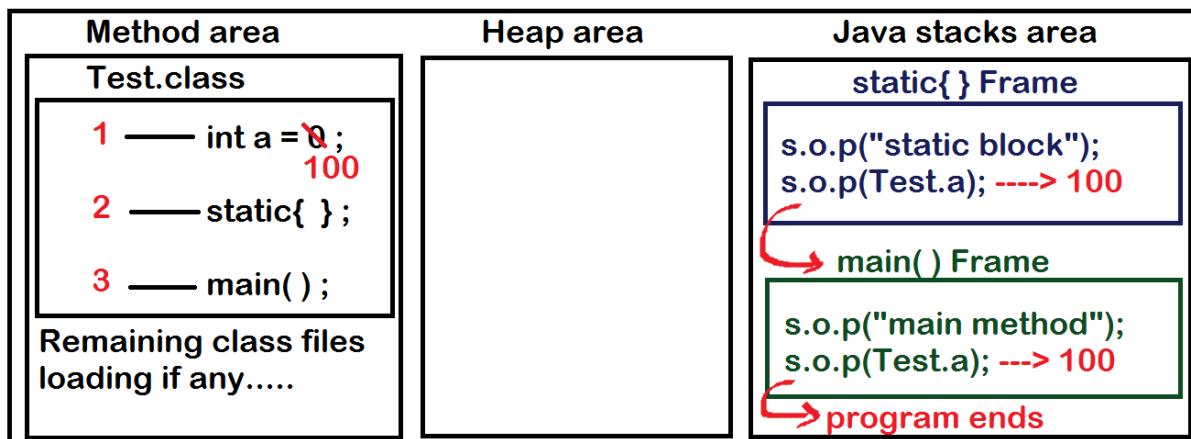
1. Memory allocation to load the class inside method area.
2. All the members of class will be registered in the defined order.
3. All the static variables initializes with default values at the time class loading.

Data_type	Default_value
int	0
char	Blank('o')
float	0.0
boolean	false
pointer	null

4. JVM set priorities to all the registered members.
5. JVM execute the members according to their priorities.

Note: Static block & static variable is having equal priority. Hence these members execute in the defined order.

JVM architecture



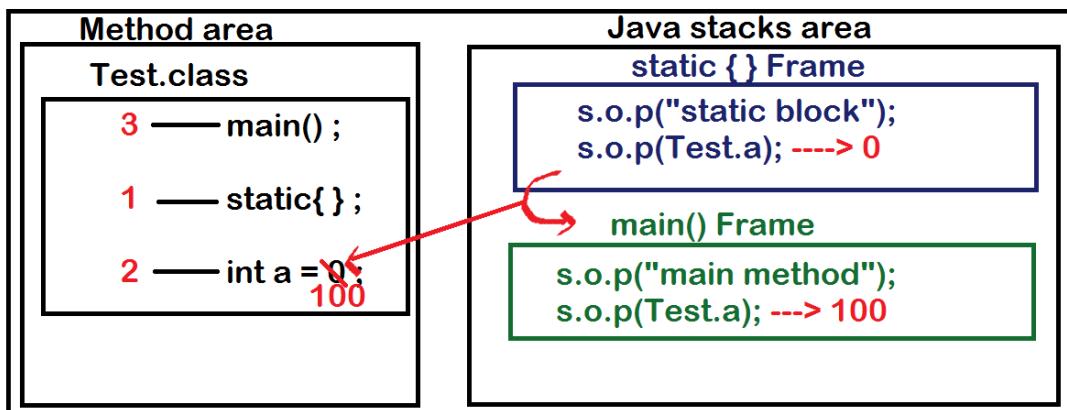
```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Main method");
    }
}
```

```

        System.out.println(Test.a);
    }
static
{
    System.out.println("Static block");
    System.out.println(Test.a);
}
static int a = 100 ;
}

```

JVM flow



```

/*
     static variable : access using class name
     local variable : direct access
*/
class Test
{
    static int a = 100 ;
    public static void main(String[] args)
    {
        System.out.println(Test.a); // access static variable directly
        System.out.println(a); // first it is looking for local variable. if it is
not present, it will access global variable
    }
}

```

```
class Test
{
    static int a = 100 ;
    public static void main(String[] args)
    {
        int a = 200 ;
        System.out.println(Test.a);
        System.out.println(a); // local priority..
    }
}
```

WAP to print all the default value of data types:

```
class Test
{
    static int a ;
    static char b ;
    static float c ;
    static boolean d ;
    static String e ; // char* = String
    public static void main(String[] args)
    {
        System.out.println("int default value : "+Test.a);
        System.out.println("char default value : "+Test.b);
        System.out.println("float default value : "+Test.c);
        System.out.println("boolean default value : "+Test.d);
        System.out.println("pointer default value : "+Test.e);
    }
}
```

```
class Test
{
    static int a = 10 ;
    public static void main(String[] args)
    {
        int a = 20;
```

```

        Test.a = Test.a + a ;
        a = a + Test.a ;
        Test.a = Test.a + a + Test.a ;
        System.out.println(Test.a);
        System.out.println(a);
    }

}

class Test
{
    static
    {
        int a = 20 ;
        System.out.println("Global a : "+Test.a);
        Test.a = Test.a + a ;
    }
    static int a = 10 ;
    public static void main(String[] args)
    {
        System.out.println("Global a : "+Test.a);
    }
}

class Test
{
    static
    {
        int a = 20 ;
        Test.a = Test.a + a ;
        a = a + Test.a ;
        Test.a = a + a ;
    }
    static int a = 10 ;
    public static void main(String[] args)
    {
        System.out.println("Global a : "+Test.a);
    }
}

```

```

}

class Test
{
    static
    {
        int a = 20 ;
        Test.a = Test.a + a ;
        a = a + Test.a ;
        Test.a = a + a ;
    }
    static int a ;
    public static void main(String[] args)
    {
        System.out.println("Global a : "+Test.a);
    }
}

class Test
{
    static
    {
        int a = 20 ;
        Test.a = a + a ;
    }
    static int a ;
    public static void main(String[] args)
    {
        System.out.println("Global a : "+Test.a);
    }
    static
    {
        a = a + Test.a ;
    }
}

```

Initialization of static variable in different ways using static user method

```

class Test
{
    static int a ;

```

```

public static void main(String[] args)
{
    System.out.println(Test.a);
    Test.initialize();
    System.out.println(Test.a);
}
static void initialize()
{
    Test.a = 100 ;
}
}

class Test
{
    static int a;
    public static void main(String[] args)
    {
        System.out.println(Test.a);
        Test.a = Test.initialize();
        System.out.println(Test.a);
    }
    static int initialize()
    {
        return 100 ;
    }
}

class Test
{
    static int a;
    public static void main(String[] args)
    {
        System.out.println(Test.a);
        Test.initialize(100);
        System.out.println(Test.a);
    }
    static void initialize(int x)
}

```

```
    {  
        Test.a = x;  
    }  
}
```

Naresh Technologies

```
class Test
{
    static int a = Test.initialize();
    public static void main(String[] args)
    {
        System.out.println(Test.a);
    }
    static int initialize()
    {
        return 100 ;
    }
}
```

Example problems to analyze:

```
class Pro1
{
    static int a = Pro1.fun();
    public static void main(String[] args)
    {
        System.out.println(Pro1.a);
    }
    static int fun()
    {
        Pro1.a = 50;
        return Pro1.fun1();
    }
    static int fun1()
    {
        System.out.println(Pro1.a);
        return 100;
    }
}
```

```
class Pro1
{
```

```

static int a = 50 ;
public static void main(String[] args)
{
    System.out.println(Pro1.fun()+Pro1.a);
}
static int fun()
{
    Pro1.a = Pro1.a +100;
    return Pro1.a;
}
}

class Pro1
{
    static int a = Pro1.fun();
    public static void main(String[] args)
    {
        System.out.println(Pro1.a);
    }
    static
    {
        System.out.println(Pro1.a);
        Pro1.a = Pro1.a+20 ;
    }
    static int fun()
    {
        Pro1.a = 50;
        return Pro1.fun1();
    }
    static int fun1()
    {
        System.out.println(Pro1.a);
        return 100;
    }
}

```

```
class Pro1
{
    static
    {
        Pro1.a = Pro1.fun();
    }
    static int a = 70 ;
    public static void main(String[] args)
    {
        System.out.println(Pro1.a);
    }
    static
    {
        Pro1.a = Pro1.a+Pro1.fun();
    }
    static int fun()
    {
        Pro1.a = 50;
        return Pro1.fun1();
    }
    static int fun1()
    {
        System.out.println(Pro1.a);
        return Pro1.a+30;
    }
}
```

Non-static members flow using JVM architecture

- Specific functionality of Object is defined as non-static.
- The members are
 - Non-static block
 - Non-static variable
 - Non-static method
 - Constructor

Note: Static members we can access using class name.

Question: How can we access non-static members in the application?

Answer: Using Object reference (address).

Question: Where I can create object for a class?

Answer: Using any one of 4 static members.

- Application having 2 contexts.
- In a Shopping Mall(application), To enter into non-static context(theatre), we need to take permissions in static context(box-office).

Creating object inside main method:

Non-static variable:

- Declaration of variable inside the class and outside to methods & blocks.
- We can't initialize directly (accno , balance of bank application we cannot decide as a programmer).
- We can initialize these variables using constructor.
- In the process of Object creation, these variables initializes with default values.

Constructor:

- Special java method.
- Constructor name & Class name should be same.
- Return type is not allowed.
- Used to initialize non-static(instance) variables.
- We need to call constructor explicitly in the process of object creation.

Non-static block:

- Defining a block with no identity.
- We can't access explicitly.
- JVM invokes implicitly as soon as object has created.

```
class Test
{
    int a ; // non-static variable
```

By Srinivas (C/DS/Java trainer)

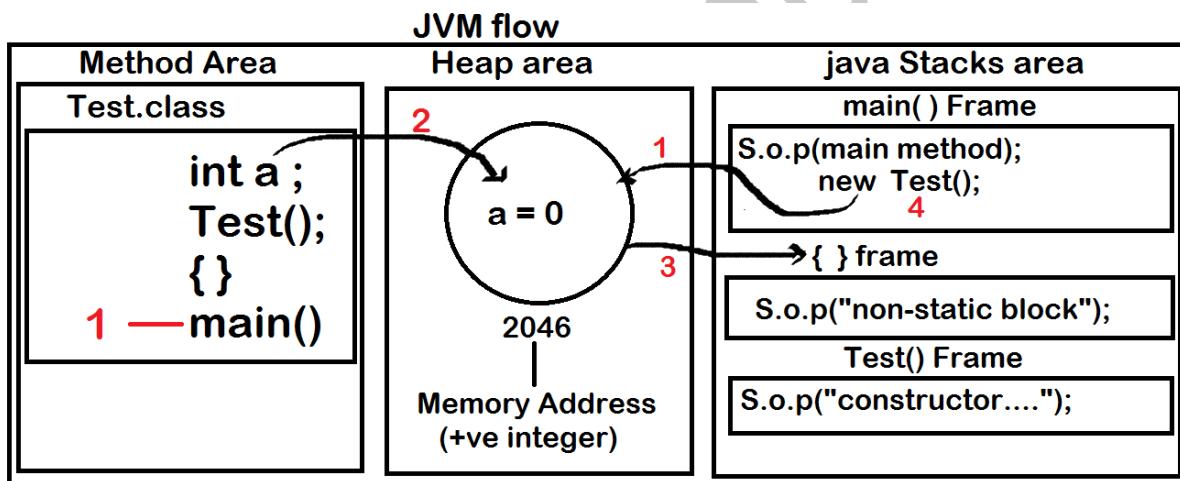
```

Test()
{
    System.out.println("Constructor");
}

{
    System.out.println("Non-static block");
}

public static void main(String[] args)
{
    System.out.println("main method");
    new Test();
}
}

```



Static members	Non-static members
Belongs to class	Belongs to Object
Can access using class name	Can access using object reference
Static block executes implicitly at the time of class loading	Non-Static block executes implicitly at the time of object creation
Static variable initializes with default values at the time of class loading.	Non-static variable initializes with default value at the time of object creation.

Note: static block executes only once in the application at the time of class loading.

But non-static block and constructor executes every time in the process of object creation.

```

class Test
{

```

```

static
{
    System.out.println("Class loading...");
}

Test()
{
    System.out.println("Object initialization process....");
}

{
    System.out.println("Object creation process....");
}

public static void main(String[] args)
{
    new Test(); // object creation
    new Test();
    new Test();
}
}

```

Note: Compiler **raises an error** if accessed method definition is not present.

```

class Test
{
    public static void main(String[] args)
    {
        Test.fun(); // Error : no such method
    }

    /*static void fun()
    {
        System.out.println("User method...");
    }*/
}

```

Note: In the process of Compilation, compiler is looking for constructor definition in the java source file. If not present, Compiler will supply a default constructor.

Default constructor is not taking any arguments and with empty definition.

```

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

```

By Srinivas (C/DS/Java trainer)

Page 52

```

{
    new Test(); // No error : invokes default constructor
}

/*Test()
{
    System.out.println("Constructor");
}*/
}

```

this:

- It is a keyword.
- It is a pre-defined non static variable.
- It holds object address.
- **It must be used only in non-static context.**

```

class Test
{
    static
    {
        System.out.println(this); // Error :
    }

    {
        System.out.println(this); // allowed
    }
}

```

Printing object address:

```

class Test
{
    Test( ) //non-static context
    {
        System.out.println(this);
        System.out.println(this.hashCode());
    }

    public static void main(String args[ ])
    {
        new Test( );
    }
}

```

Question: How can we access non-static members from non-static context?

Answer: using “this” keyword.

```

class Test
{
    int a ;
    public static void main(String args[ ])
    {
        new Test();
    }
    {
        System.out.println(this.a);
        this.a = 100 ;
    }
    Test()
    {
        System.out.println(this.a);
    }
}

```

Note : “this” keyword points to only one object at a time among available objects in Heap area

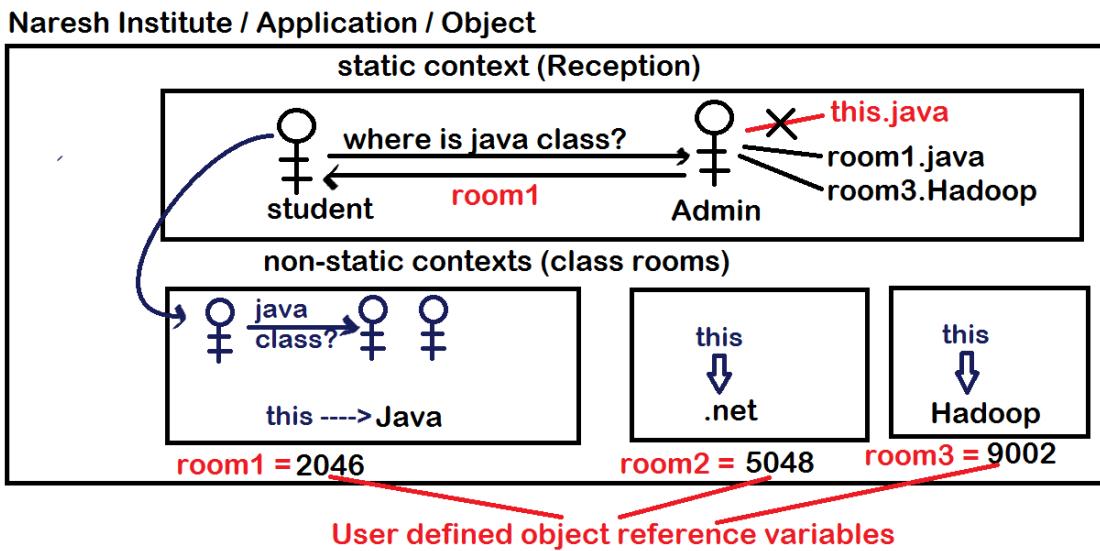
```

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

```

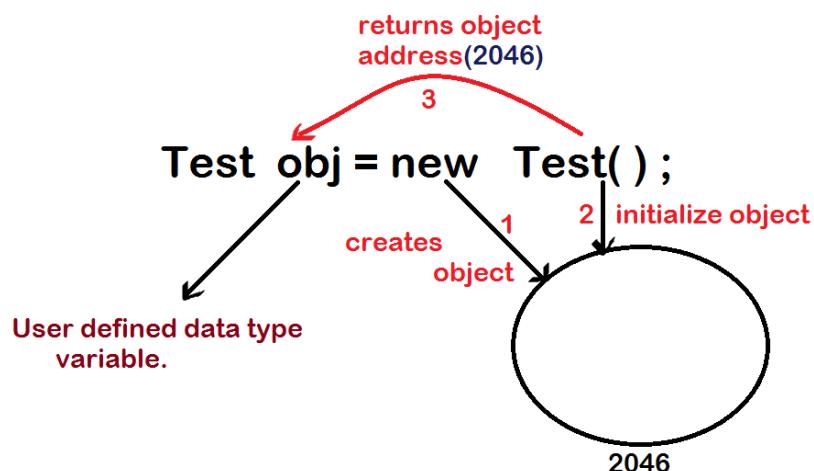
Question: How can we access non-static members from static context?

Answer: using user defined “object-reference variable”.



User-defined Object reference variable:

- Used to store object address.
- Data type should be of “class-type”.
- Constructor doesn’t allow any return type in its definition.
- But constructor returns “object address” implicitly as soon as object has created.
- We should collect address into class-type variable only.



WAP to print “object-address” both in static & non-static contexts:

```
class Test
{
```

```

Test( )
{
    System.out.println("Object address inside constructor : "+this);
}
public static void main(String args[ ])
{
    Test obj = new Test();
    System.out.println("Object address inside main : "+obj);
}
}

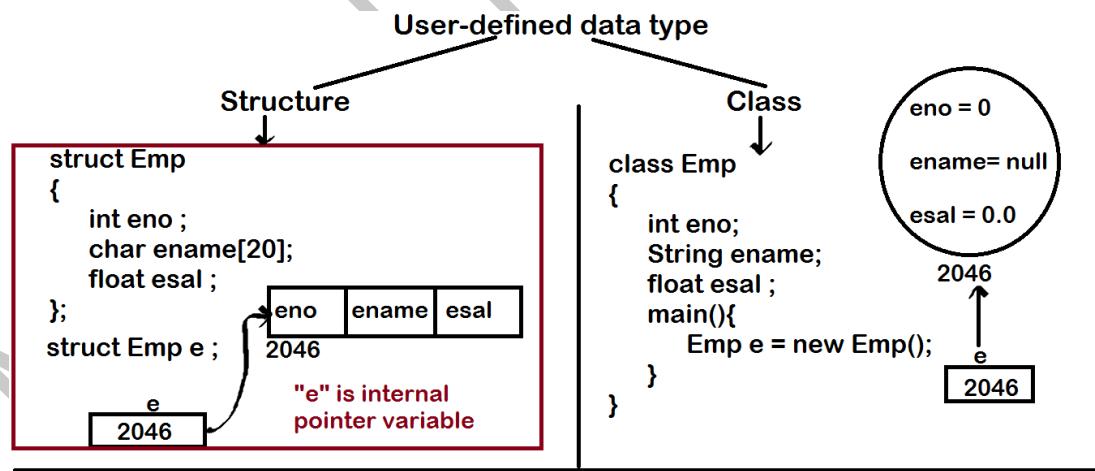
```

Question: Why can't we collect address into integer variable?

Answer:

- Class is extension to structure in C / C++
- Structure is used to allocate memory to different kinds of data elements.
- After allocating memory to a structure, we can collect the address into a variable of structure type.
- Structure variable is “internal pointer variable”, because it holds address.
- We can access structure elements using dot(.) operator.

Int variable → integer (data)
Pointer variable → integer (address)



Note : Both in structures & classes, we can access members using dot(.) operator.

```

class Employee
{
    int eno ;

```

```

String ename;
float esal;
public static void main(String[] args)
{
    Employee e = new Employee();
    System.out.println("Eno :" +e.eno);
    System.out.println("Ename :" +e.ename);
    System.out.println("Esal :" +e.esal);
}
}

```

Creating object in different static contexts

int a = 100 ; (primitive)
Test obj = new Test(); (User-defined)

Creating object inside static block and assign object reference to Local variable:

```

class Test
{
    static
    {
        int a = 100 ; // 'a' is local variable
        System.out.println(a);
    }
    public static void main(String[] args)
    {
        // System.out.println(a); // Error :
    }
}

```

```

class Test
{
    static
    {
        Test obj = new Test(); // 'obj' is local
        variable
        System.out.println(obj);
    }
    public static void main(String[] args)
    {
        // System.out.println(obj); // Error :
    }
}

```

Create object inside static block and assign object reference to static variable:

```

class Test
{
    static int a ;
    static
    {
        System.out.println(Test.a);
        Test.a = 100 ;
    }
}

```

```

public static void main(String[] args)
{
    System.out.println(Test.a);
}

```

Create object directly in the declaration of static variable:

```

class Test
{
    static int a = 100 ;
    static
    {
        System.out.println(Test.a);
    }
    public static void main(String[] args)
    {
        System.out.println(Test.a);
    }
}

```

```

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

```

Create object inside static user method and assign it to static variable:

```

class Test
{
    static int a ;
    public static void main(String[] args)
    {
        System.out.println(Test.a);
        Test.initialize();
        System.out.println(Test.a);
    }
    static void initialize()
    {
        Test.a = 100 ;
    }
}

```

```

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

```

A static method returns Object reference:

```

class Test
{
    static int a ;
    public static void main(String[] args)
    {
        System.out.println("a value :
"+Test.a);
        Test.a = Test.initialize( );
    }
}

```

```

class Test
{
    static Test obj;
    public static void main(String[] args)
    {
        System.out.println("obj value :
"+Test.obj);
        Test.obj = Test.initialize( );
    }
}

```

<pre> System.out.println("a value : "+Test.a); } static int initialize() { return 100 ; } } </pre>	<pre> System.out.println("obj value : "+Test.obj); } static Test initialize() { return new Test(); } } </pre>
--	---

Initialization of Object:

- All the non-static variables initializes with default value in object creation process.
- We can also initialize these variables explicitly using constructor.

Default initialization:

```

class Test
{
    int a ;
    public static void main(String[] args)
    {
        Test obj = new Test();
        System.out.println("a value : "+obj.a);
    }
}

```

- Arguments of any method working like “local variables”
- We must access non-static members from non-static context using “this”.

```

class Test
{
    int a ;
    Test(int x)
    {
        a = x ; // local variable value assigned to non-static variable (with the name ‘a’ local
variable is not present)
    }
    public static void main(String[] args)
    {
        Test obj = new Test(10);
        System.out.println("a value : "+obj.a);
    }
}

```

```

class Test
{

```

```

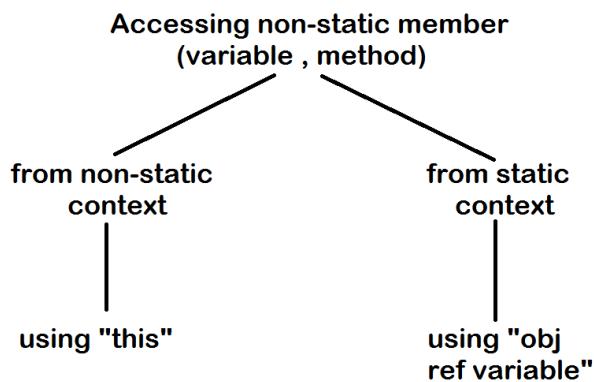
int a ;
Test(int a)
{
    a = a ; //local variable value re-assigned to local variable only.
}
public static void main(String[] args)
{
    Test obj = new Test(10);
    System.out.println("a value : "+obj.a);
}
}

class Test
{
    int a ;
    Test(int a)
    {
        this.a = a ; // local variable value is assigned to non-static variable
    }
    public static void main(String[] args)
    {
        Test obj = new Test(10);
        System.out.println("a value : "+obj.a);
    }
}

```

Non-static user method:

- A block of instructions having Identity.
- Method is taking input, processing input and producing output.
- Every method must be called explicitly.
- We can access non-static method using object reference variable



```

class Test
{
    public static void main(String[] args)
    {
        // Test.fun(); // Error :
        // this.fun(); // Error :

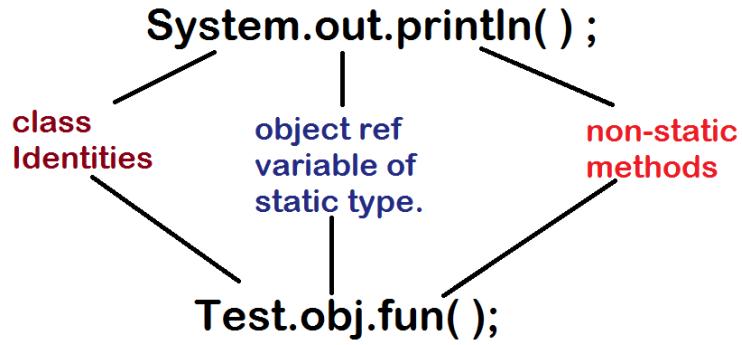
        Test obj = new Test(); // 'obj' is local variable
        obj.fun( );
    }

    void fun()
    {
        System.out.println("non-static user method....");
    }
}

class Test
{
    static Test obj = new Test(); // 'obj' is static variable
    public static void main(String[] args)
    {
        Test.obj.fun();
    }

    void fun()
    {
        System.out.println("non-static user method....");
    }
}

```



Instructions to write program

1. Create object inside static block and assign address to static variable.
2. Access non-static user method using object reference variable from main method.

```
class Test
{
    static Test obj;
    static
    {
        Test.obj = new Test();
    }
    public static void main(String[] args)
    {
        System.out.println("Main method");
        Test.obj.fun();
    }
    void fun()
    {
        System.out.println("non-static user method");
    }
}
```

Instructions to write program

- a) Define 2 static blocks before and after to main method.
- b) Declare a static variable followed by main method and proceeded by second static block, initialize static variable with 100.
- c) Display static variable value in first static block.
- d) Assign 200 to static variable in the second static block
- e) Display static variable value in the main method.

```
class Test
{
    static
    {
        System.out.println("a value : "+Test.a);
    }
    public static void main(String[] args)
    {
        System.out.println("a value : "+Test.a);
    }
    static int a = 100 ;
    static
    {
```

```
        Test.a = 200 ;  
    }  
}
```

Instructions to write program

- a) Define static block
- b) Define non-static block
- c) Define main method
- d) Define constructor
- e) Create object in the main method

Output should be :

class loaded with static block
execution starts at main method
non-static block
constructor
execution ends at main method

Instructions to write program

- a) declare a non-static integer variable
- b) define a constructor
- c) create object inside the main method
- d) assign "100" to non-static variable from the constructor.

```
class Test  
{  
    int a;  
    Test()  
    {  
        this.a = 100;  
    }  
    public static void main(String[] args)  
    {  
        new Test();  
    }  
}
```

Instructions to write program

- a) declare a non-static integer variable
- b) define a constructor
- c) create object inside the main method
- d) print variable value inside the non-static block in the first statement and assign new value "100" to variable in the second statement.
- e) print variable value inside the constructor in the first statement and assign another new value "200" to variable in the second statement.
- f) print variable value in the main method.

Instructions to write program

- a) declare a static variable of class type
- b) define static block and print variable value inside.
- c) define main method.
- d) create object inside the main method
- e) print variable value inside the constructor and check the flow

Instructions to write program

- a) Declare a static int variable
- b) Declare a non-static int variable
- c) Define a static block and assign a value to static variable
- d) define a constructor
- e) define main method
- f) create object inside the main method and assign the reference to local variable inside the main method
- g) assign static variable value to non-static variable inside the constructor.
- h) print static variable and non-static variables inside the main method

Instructions to write program

- a. declare a static variable of class type and initialize directly using object reference.
- b. define static block.
- c. define main method.
- d. define non-static user method
- e. invoke non-static user method from static block and main method.

```
class Test
{
    static Test obj = new Test();
    static
    {
        Test.obj.fun();
    }
    public static void main(String[] args)
    {
        Test.obj.fun();
    }
    void fun()
    {
        System.out.println("non-static user method.....");
    }
}
```

Instructions to write program:

By Srinivas (C/DS/Java trainer)

Page 64

1. create a class
2. create static variable of class type
3. define a constructor
4. define main method
5. create object inside main method as follows
new Test();
6. Assign object reference to static variable from constructor.
7. print object reference variable inside main method

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

Instructions to write program:

- 1) Define a class with Identity "Demo"
- 2) Declare a static variable "s1" of class type
- 3) Declare a non-static variable "s2" of class type
- 4) Define main method.
- 5) Declare a local variable "s3" inside main of class type
- 6) Create object inside the main method and assign it to "s3"
- 7) Define constructor.
- 8) assign object reference to s1 & s2 from constructor.
- 9) define non-static method
- 10) access non-static method from main method using s1, s2 & s3.

Note : only one object creation inside the main method

Instructions to write program:

- 1) Define a class with identity “Demo”
- 2) Declare a static variable of class type.
- 3) Define a static block
- 4) Define main method
- 5) Define static user method of return type is void
- 6) Define non-static method.
- 7) Create object inside the static block and pass the reference to static user method
- 8) Collect reference inside the static user method and assign to static variable
- 9) Call non-static method from main method

Instructions to write program:

1. Define a class
2. Declare static & non-static variables of class type
3. Define constructor
4. Define main method
5. Create object inside main method and assign object address to static variable.
6. Assign object address to non-static variable from constructor
7. Define user method
8. Access user method from main method using non-static variable.

```
class Demo
{
    static Demo obj1;
    Demo obj2;
    Demo()
    {
        this.obj2 = this;
    }
    public static void main(String[] args)
    {
        Demo.obj1 = new Demo();
        Demo.obj1.obj2.fun();
    }
    void fun()
    {
        System.out.println("User method....");
    }
}
```

```
class Demo
{
    Demo obj2;
    Demo()
    {
        this.obj2 = this;
    }
    public static void main(String[] args)
    {
        Demo obj1 = new Demo();
        obj1.obj2.fun();
    }
    void fun()
    {
}
```

```
        System.out.println("User method....");
    }
}
```

Note: We can't create object for any class inside constructor

```
class Demo
{
    Demo()
    {
        new Demo(); // infinite object creations.....
    }
    public static void main(String[] args)
    {
        new Demo();
    }
}
```

Connecting classes in java application:

- Every application (software) is a set of programs.
- When we define application logic in various programs (classes), we can connect them by calling one class members from another class.

Question: Can we define more than one class in a single source file?

Answer: Yes allowed.

Program.java

```
class First
{
    main()
    {
        ...belongs to
        only one class
    }
}
class Second
{
}
```

----> Compiler

First.class ---> Run ---> output

Second.class---> Run --> Error

(main is not present)

- JVM will not load un-used classes in the application.
- As a programmer, we can analyze whether a class has loaded or not only by defining static block.
- Static block executes implicitly at the time of class loading.

```
class Login
{
    static
    {
        System.out.println("Login page is loading....");
    }
    public static void main(String[] args)
    {
        System.out.println("Communication starts....");
        Inbox.maillInfo(); // It loads the Inbox class first.
    }
}

class Inbox
{
    static
    {
        System.out.println("Inbox page is loading....");
    }
    static void maillInfo()
    {
        System.out.println("Recent mail info.....");
    }
}
```

```

class Account
{
    int balance ;
    Account(int balance)
    {
        this.balance = balance ;
    }
    int getBalance()
    {
        return this.balance ;
    }
    void withdraw(int amt)
    {
        System.out.println("Trying to withdraw : "+amt);
        System.out.println("Avail Balance : "+this.balance);
        if(amt <= this.balance)
        {
            this.balance = this.balance - amt;
            System.out.println("Pls collect : "+amt);
        }
        else
        {
            System.out.println("Insufficient funds.....");
        }
    }
    void deposit(int amt)
    {
        System.out.println("depositing.... : "+amt);
        this.balance = this.balance + amt ;
    }
}

class Bank
{
    public static void main(String[] args)
    {
        System.out.println("Processing Account.....");

        int amount = 5000 ;
        Account acc = new Account(amount);
        System.out.println("Initial balance : "+acc.getBalance());

        amount = 4000 ;
        acc.withdraw(amount);
    }
}

```

```

        System.out.println("Balance after withdrawal : "+acc.getBalance());

        amount = 2000 ;
        acc.deposit(amount);
        System.out.println("Final balance : "+acc.getBalance());
    }
}

class First
{
    public static void main(String[] args)
    {
        Second addr = new Second();
        addr.check();
    }
}
class Second
{
    void check()
    {
        System.out.println("Second class non-static method....");
    }
}

class First
{
    static Second addr = new Second();
    public static void main(String[] args)
    {
        First.addr.check();
    }
}
class Second
{
    void check()
    {
        System.out.println("Second class non-static method....");
    }
}

```

System.out.println() :

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

class PrintStream
{
    void println()
    {
        // prints a msg on console.....
    }
}
```

Naresh Technologies

Data types

In the declaration of every variable, it is mandatory to represent its data type.

Syntax:

```
data_type <variable_name>;
```

Example:

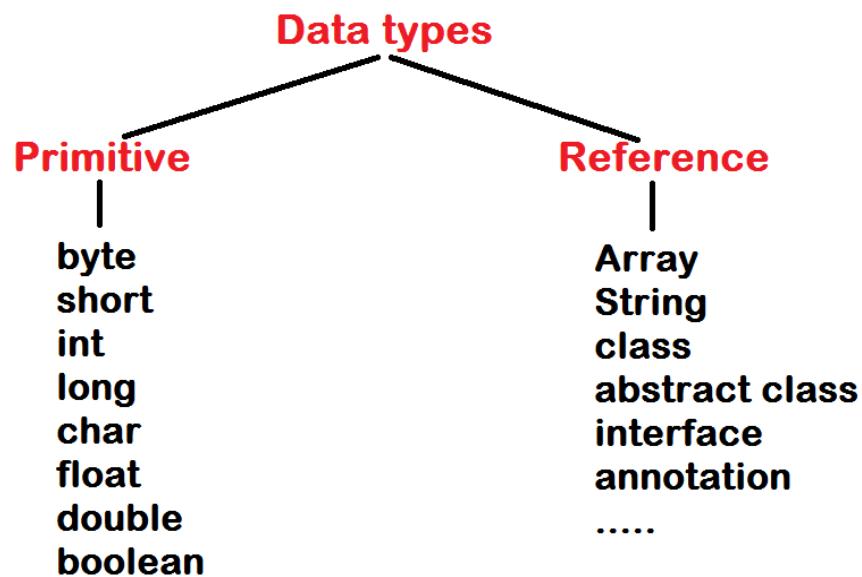
```
int sum = 0;
```

Why data type?

1. Represents how much memory is required to hold the data.
2. Represents what type of data is allowed.

Classification:

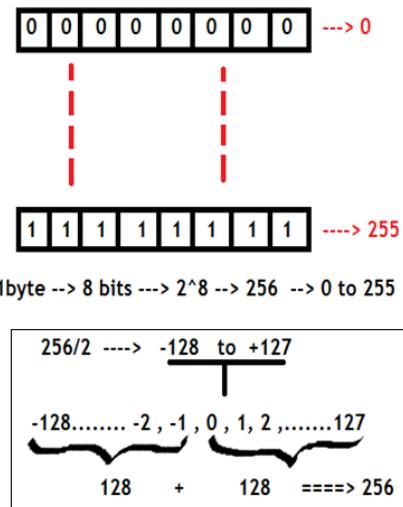
- Java data types classified into 2 types
- Primitive data types and user defined.
- Class is the user-defined data type
- User defined data types are reference(object) type in java application



Limits of each data type :

- Every data type is having size and it is represented in bytes
- One byte equals to 8 bits.
- Following diagram represents what are the minimum and maximum values can be stored into each data type depends on their size.

Type	size	Limits
byte	1b	-128 to +127 (-2^7 to +2^7-1)
short	2b	16bit -> 65536 -> -32768 to +32767 (-2^15 to +2^15-1)
int	4b	32 bit -> -2^31 to +2^31-1
long	8b	64 bit -> -2^63 to +2^63-1
char	2b	16bit -> 65536 -> 0 to 65535
float	4b	Exponential values
double	8b	
boolean		true , false



Questions on character data type:

1. Why char data type limits discussion in integers?
 - To represent character set in integers.
2. Why character occupy 2 bytes memory in java and .net where as it occupies only 1 byte in C and C++?
 - Java & .net apps are web apps.
 - Web app need to represent more than one language character set a time. Hence it occupies 2 bytes.
 - C & C++ are platform dependent. These can represent only one language at a time.
3. How can we store a symbol into 1 byte memory in C or C++?
 - A language at most having 256 symbols(1 byte range)
4. What is character system?
 - A character system represents all the symbols of one language using constant integer values.
5. What is ASCII?
6. What is UNICODE?

Explanation:

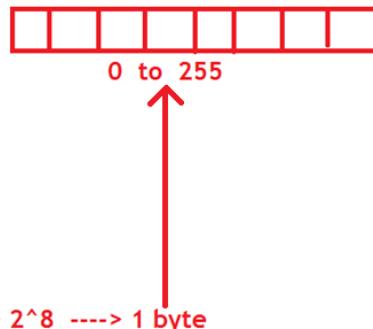
- Generally, compiler needs to convert all the symbols of java source file into class file.
- To convert all the characters of 1 language into machine code, compiler uses character system.
- Character system introduced along with computer languages.
- We have so many character systems but Popular one is "ASCII"
- Character system represents all the symbols of one language using constant integer values.

ASCII : (American Standard Code for Information Interchange)

fixed code....

English - ASCII :

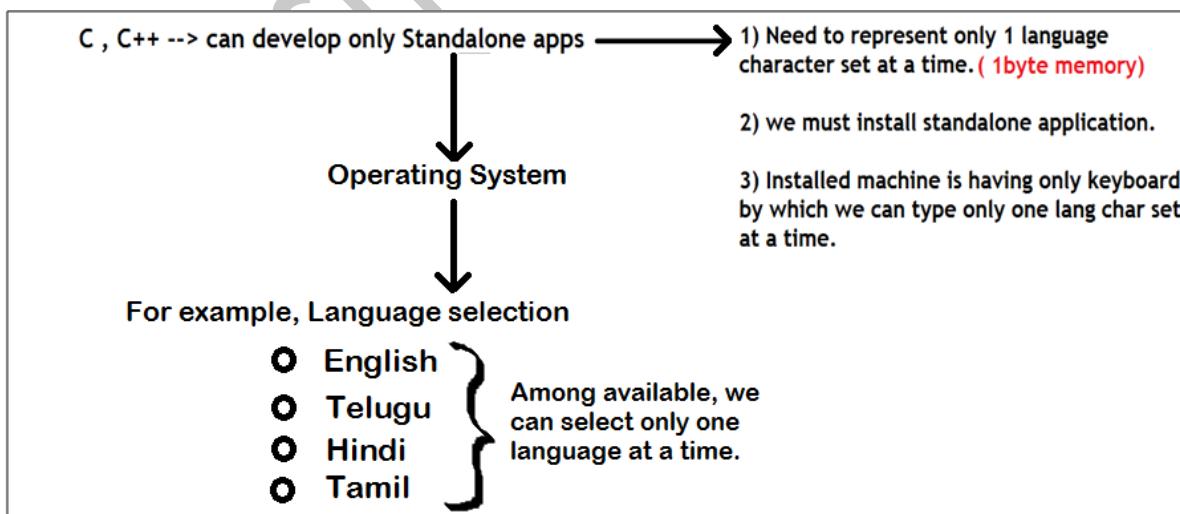
A - 65	a - 97	0 - 48	# - 35
B - 66	b - 98	1 - 49	'' - 32
...	+
...	/
...	\
Z - 90	z - 122	9 - 57
26	+ 26	+ 10	+ 150 <= 256 -----> 2^8 -----> 1 byte



The best example that describes any language in this world is having not more than 256 symbols.
is keyboard.

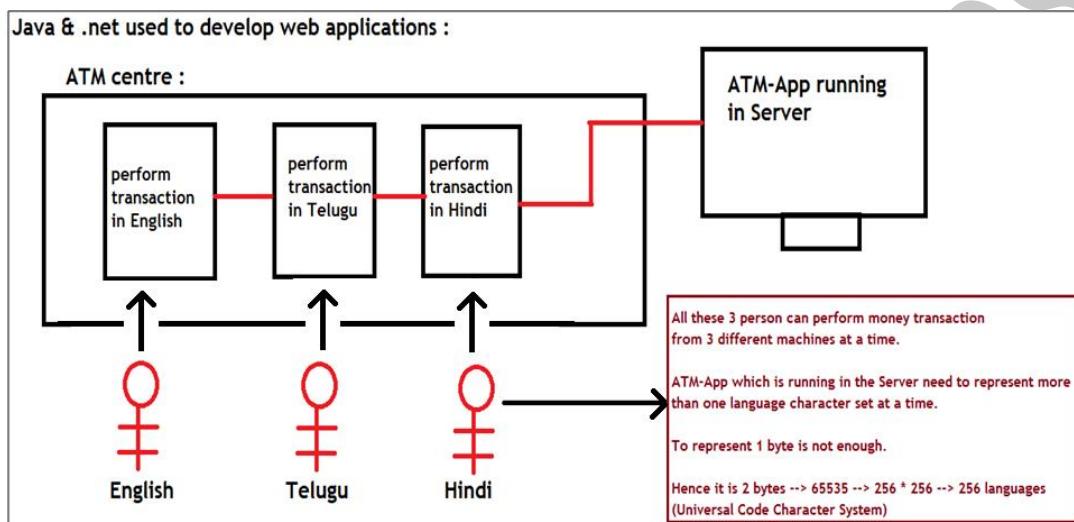
ASCII v/s UNICODE:

- C & C++ languages need to represent only one language character system at a time. Because these languages used to develop only standalone applications.
 - ASCII can represent all the symbols of one language using 1 byte
 - Hence C & C++ character data type occupies 1 byte memory.
- In the following diagram standalone application (photo shop) supports “n” number of languages but at a time it has to represent only one language.
- Suppose in our mobile, at a time we can select only language (radio button) hence it is also standalone.
- To represent one language character set completely, 1 byte (256) number enough.



- Java & .net applications need to represent more than one language character system at a time.
- It can be accessed from number of systems at a time, hence they may use different languages from different machines.
- Hence to represent more than one language character set, we need to use UNICODE character system

In this diagram, a web-application need to represent more than one language character set as we can run the web application from different machines at a time in different languages.



Programs on data types:

```

class DataTypes {
    public static void main(String[] args) {
        //byte b = 150 ; //CE : out of range

        short s = 150 ; //Allowed (-32768 to +32767)
        short s1 = 50000 ; //CE : out of range
    }
}

class DataTypes {
    public static void main(String[] args) {
        byte a=10 , b=20 ;

        //byte c = a+b ; //CE : arithmetic operations return result in integer format.
        //short s = a+b ; //CE : not allowed
        int c = a+b ; //Allowed
        long c = a+b ; //Allowed
    }
}

```

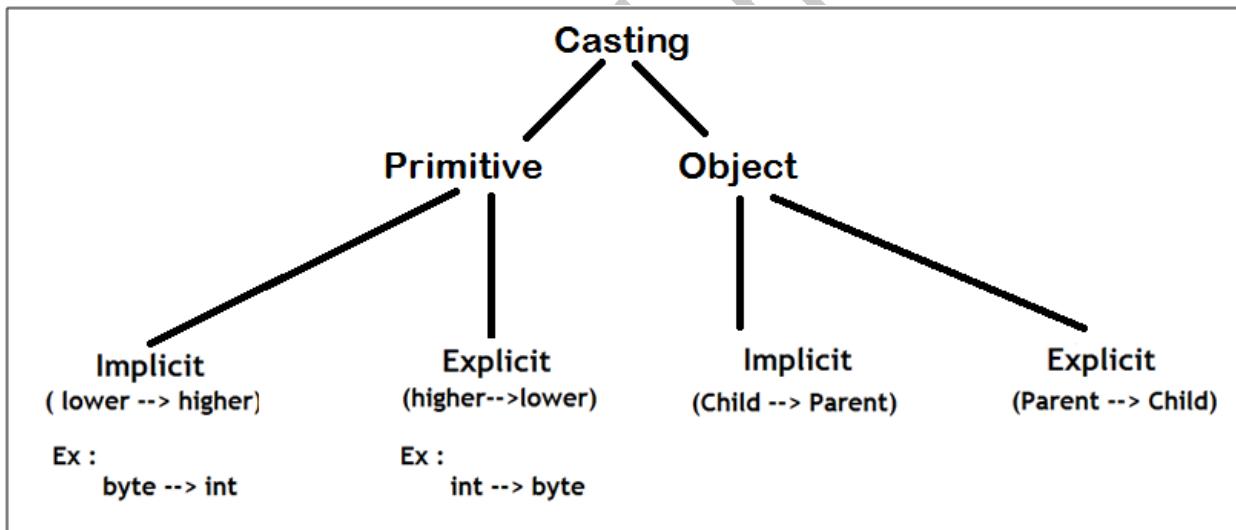
```

class Test {
    public static void main(String[] args) {
        long a=10 , b=20 ;
        // int c = a+b ; //Error :
        long c = a+b ;
        System.out.println("sum : "+c);
    }
}

```

Type casting:

- Conversion of data from one data type to another data type.
- Casting happens at runtime.
- Java allows
 - Primitive casting
 - Object casting
- Casting can be either
 - implicit (internal) by JVM
 - Explicit (external) by Programmer.



Programs :

```

class ImplicitCast {
    public static void main(String[] args) {
        byte b = 100 ;
        int i ;
        i = b ; //implicit cast.....
        System.out.println("i value : "+i);

        char ch = 'A' ;
        int j ;
        j = ch ; //implicit cast.....
        System.out.println("j value : "+j);
    }
}

```

```

    }

}

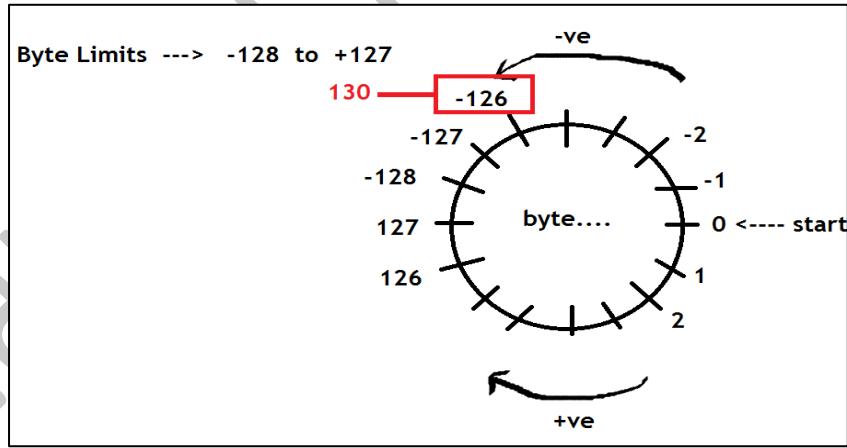
class ExplicitCast {
    public static void main(String[] args) {
        int i = 100 ;
        byte b ;

        //b = i ; //direct assignment is not allowed
        b = (byte)i ;
        System.out.println("b value : "+b);

        int j = 97;
        char ch = (char)j;
        System.out.println("ch value : "+ch);
    }
}

class ExplicitCast {
    public static void main(String[] args) {
        int i = 130 ;
        byte b ;
        b = (byte)i ;
        System.out.println("b value : "+b);
    }
}

```



```

class ExplicitCast {
    public static void main(String[] args) {
        int i1 = 256 , i2 = 512 , i3 =1024 ;
        byte b1, b2, b3 ;

        b1 = (byte)i1 ;
        b2 = (byte)i2 ;

```

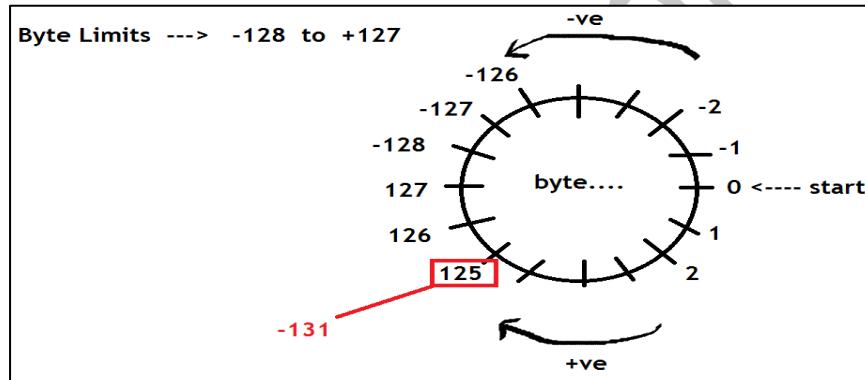
```

        b3 = (byte)i3;

        System.out.println("b1 value : "+b1);
        System.out.println("b2 value : "+b2);
        System.out.println("b3 value : "+b3);
    }
}

class ExplicitCast {
    public static void main(String[] args) {
        int i1 = -131;
        byte b1;
        b1 = (byte)i1;
        System.out.println("b1 value : "+b1);
    }
}

```



Note : Using type casting, we can conserve/save the memory in the application....

```

class DataTypes {
    public static void main(String[] args) {
        byte a=10 , b=20;

        //int c = a+b ; //allowed but occupies 4 byte memory.....
        byte c = (byte)(a+b); //using typecast, if we store into byte variable, we can
        save 3 bytes memory.....  
  

        System.out.println("c value : "+c);
    }
}

```

```

class DataTypes {
    public static void main(String[] args)
    {
        int x=5 , y=2 ;
        int z = x/y ;
        System.out.println("z value : "+z);
    }
}

```

```

        }
    }

/*
    int / int --> int
    int / float --> float
    float / int--> float
    float / float --> float
*/
}

class DataTypes {
    public static void main(String[] args) {
        int x=5 , y=2 ;
        float z = x/y ;
        System.out.println("z value : "+z);
    }
}

class DataTypes {
    public static void main(String[] args) {
        int x=5 , y=2 ;
        float z = (float)x/y ;
        System.out.println("z value : "+z);
    }
}

class DataTypes {
    public static void main(String[] args) {
        int x=5 , y=2 ;
        float z = (float)x/(float)y ;
        System.out.println("z value : "+z);
    }
}

```

Wrapper classes

Question: Is java fully Object Oriented Programming Language(OOPL)?

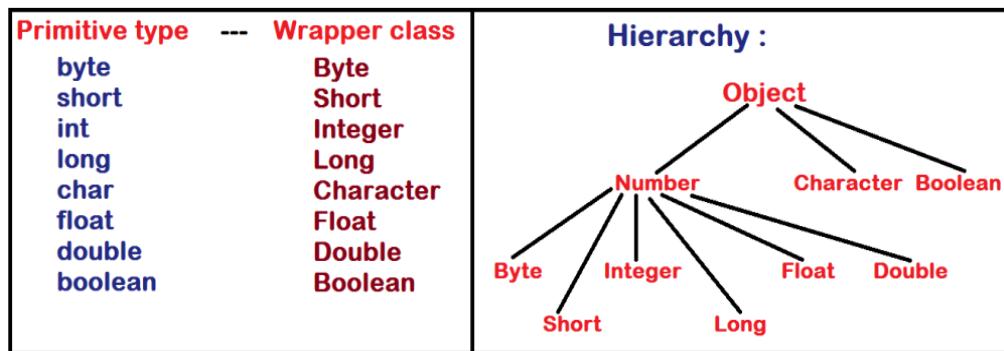
Answer: No, java supports primitive data types.

Note: If any language supports primitive types as well as OOP features, referred as "Partial OOPL".

Examples of fully OOPL... 1) Small talk
 2) Ruby

- Using primitive data types we cannot develop fully object oriented java application.
- We use wrapping to convert primitive data into objects and object data into primitives

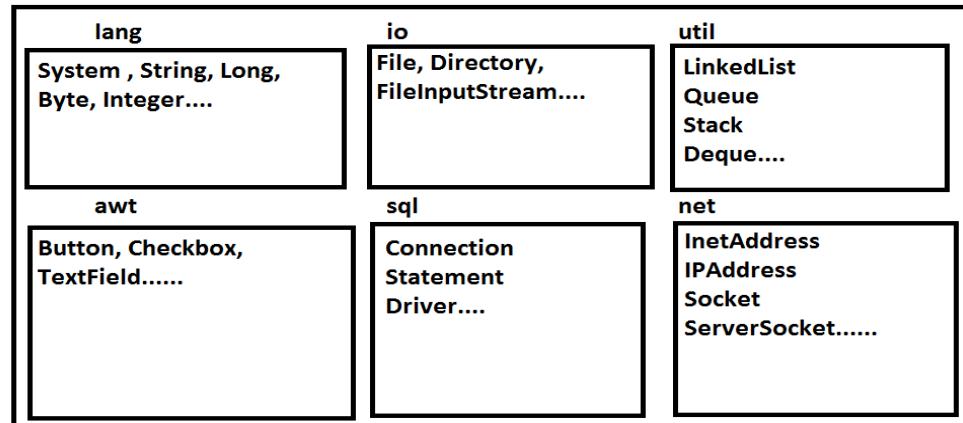
Every **Primitive** data type having corresponding **Wrapper class** :



- All the wrapper classes belong to "lang" package.
- All the Java API classes belong to a specific pre-defined packages.
- "java" is the super package of all.
- Naming conventions of package is "SINGLE WORD IN LOWER CASE"
- Examples... java, lang, io, awt, net, sql.....

Pre-defined packages structure...

java



Note: java.lang package classes will be imported into any java application implicitly.

- “lang” package is called default package in java.
- “lang” package classes mostly used by basic java programmer(learners).
- Working with “import” statement is bit complex.
- As basic programmer cannot understand “import” in the beginning, basic classes(belongs to lang) available directly.

Conversions using Wrapper classes:

- In any java application, we are using mainly 6 conversions...
 - 1) Primitive to Object (Boxing)
 - 2) Object to Primitive (UN boxing)
 - 3) Primitive to String
 - 4) String to Primitive
 - 5) Object to String
 - 6) String to Object

Primitive to Object Conversion : (Boxing)

Pre-defined method in Byte class

```
class Byte
{
    static Byte valueOf(byte b)
    {
        converts byte -> Byte
    }
}
```

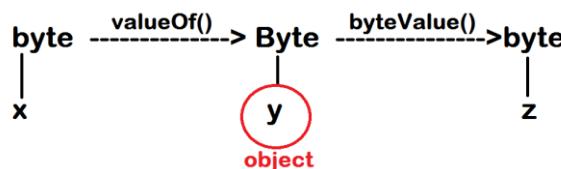
Conversion Logic :

```
class PrimitiveToString
{
    p.s.v.main(String args[])
    {
        byte x = 100 ;
        Byte y = Byte.valueOf(x);
    }
}
```

In the above example, we use static method of Byte class to convert primitive byte to Object Byte type.

UN boxing:

- Conversion of Object Byte to primitive byte.
- We use non-static method of Byte class for this conversion.



Object to Primitive conversion : (Unboxing)

Pre-defined method in Byte class :

```
class Byte
{
    public byte byteValue()
    {
        converts Byte --> byte
    }
}
```

Conversion Logic :

```
class ObjectToPrimitive
{
    p.s.v.main(String args[])
    {
        byte x=100;
        Byte y = Byte.valueOf(x); //Boxing
        byte z = y.byteValue(); //Unboxing
    }
}
```

Primitive to String and String to Primitive Conversions:

- In some of the situations, we need to take the input in the form of string.
- To process the collected data, we need to convert the data into primitive types.
- After processing the data, again we should re-convert the data into String to display.
- Observe the following concept to understand, why conversions required....

Applet / Swing / HTML

Calculator

Enter no1 :

Enter no2 :

Result :

ADD

1) Text fields accept input in String format.

2) We must convert these input values into primitive data using `parseXxx()` methods.

3) Perform "Add" operation
4) Re convert into String
5) Set to "Result" field

Primitive to String :

Primitive to String conversion :

Pre-defined method in Byte class :

```
class Byte
{
    public static String toString(byte b)
    {
        converts byte --> String
    }
}
```

Conversion logic :

```
class PrimitiveToString
{
    p.s.v.main(String args[])
    {
        byte x = 100 ;
        String y = Byte.toString(x);
    }
}
```

String to Primitive conversion :

Pre-defined method in Byte class :

```
class Byte
{
    public static byte parseByte(String s)
        throws NumberFormatException
    {
        1) converts String -> byte
        2) throws runtime error in case
           of invalid data conversion
    }
}
```

Conversion logic :

```
class StringToPrimitive
{
    p.s.v.main(String args[])
    {
        String s = "100";
        byte b = Byte.parseByte(s);
    }
}
```

Note: parseByte() method throwing exception in the above conversion. We will discuss briefly how to handle exceptions. But here I can explain why exception is throwing....

Why parseXxx() method throws Exceptions ?

Ans : Data conversion will be done at runtime.
Invalid data conversions results "Runtime errors"

```
class StringToPrimitive
{
    p.s.v.main(String args[ ])
    {
        String s1 = "150";
        byte b1 = Byte.parseByte(s1); // Exception : value out of range

        String s2 = "abcd";
        byte b2 = Byte.parseByte(s2); // Exception : invalid input string
    }
}
```

Object to String conversion :

Pre-defined method in Byte class :

```
class Byte
{
    public String toString()
    {
        converts Byte --> String
    }
}
```

Conversion logic :

```
class ObjectToString
{
    p.s.v.main(String args[])
    {
        byte x = 100 ;
        Byte y = Byte.valueOf(x);
        String z = y.toString();
    }
}
```

String to Object conversion :

Pre-defined method of Byte class :

```
class Byte
{
    public static Byte valueOf(String s)
        throws NumberFormatException
    {
        1) converts String --> Byte
        2) Runtime Error in case of
            invalid data conversion
    }
}
```

Conversion logic :

```
class StringToObject
{
    p.s.v.main(String args[])
    {
        String s = "100";
        Byte b = Byte.valueOf(s);
    }
}
```

Practice byte → String → Byte → byte → Byte → String → byte:

class Conversions

```
{
    public static void main(String args[ ])
    {
        byte a = 100;
        String b = Byte.toString(a);           // byte → String
        Byte c = Byte.valueOf(b);             // String → Byte
        byte d = c.byteValue();              // Byte → byte
        Byte e = Byte.valueOf(d);             // byte → Byte
        String f = e.toString();              // Byte → String
        byte g = Byte.parseByte(f);          // String → byte
    }
}
```

- Byte class having 2 pre-defined constructors to construct Byte class Object from primitive byte and String.
- Among all the 6 data conversion, it is possible to perform 2 conversion using constructors also.

Why Byte class having 2 constructors :

Using Wrapper classes we can perform 6 data conversions using pre-defined methods.

Among these, 2 conversions we can also perform using constructors.

Input ---> Output

byte	--->	Byte	_____
Byte	--->	byte	
byte	--->	String	
String	--->	byte	
Byte	--->	String	
String	--->	Byte	

Byte(byte b){
 converts byte --> Byte
}

Byte(String s){
 converts String --> Byte
}

class ByteConstructors

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

```

{
    byte x1 = 100 ;
    Byte y1 = new Byte(x1); // byte -> Byte

    String x2 = "100";
    Byte y2 = new Byte(x2); // byte -> String

    String x3 = "abcd";
    Byte y3 = new Byte(x3); //Exception : Invalid data conversion
}
}

```

- Byte class is having not only methods & constructor, it has set of variables.
- All these variables represent limits & size of byte data type.

Byte class having pre-defined fields(variables) to represent

- 1) Limits of byte type
- 2) Size of byte (in bits)

```

class Byte
{
    public static final byte MIN_VALUE ;
    public static final byte MAX_VALUE;
    public static final int SIZE ;
}

```

```

class ByteFields
{
    public static void main(String[] args)
    {
        System.out.println("byte MIN value : "+Byte.MIN_VALUE);
        System.out.println("byte MAX value : "+Byte.MAX_VALUE);
        System.out.println("byte SIZE in bits : "+Byte.SIZE);
    }
}

```

Note the followings.....

- We discussed above 6 conversion using byte data type.
- Remaining conversions like int, float, boolean ... in the same way

For example...

```

byte b = Byte.parseByte("10"); //String -> byte
int I = Integer.parseInt("10"); //String -> int
boolean b = Boolean.parseBoolean("false"); //String -> boolean

```

Following example describes same kind of fields available in all the wrapper classes....

```
class Limits
{
    public static void main(String[] args) {
        System.out.println(Short.MIN_VALUE);
        System.out.println(Short.MAX_VALUE);

        System.out.println(Float.MIN_VALUE);
        System.out.println(Float.MAX_VALUE);

        System.out.println((int)Character.MIN_VALUE);
        System.out.println((int)Character.MAX_VALUE);
    }
}
```

Note:

Type casting can be applicable only on primitive data types.
We cannot convert Object to primitive through type casting.

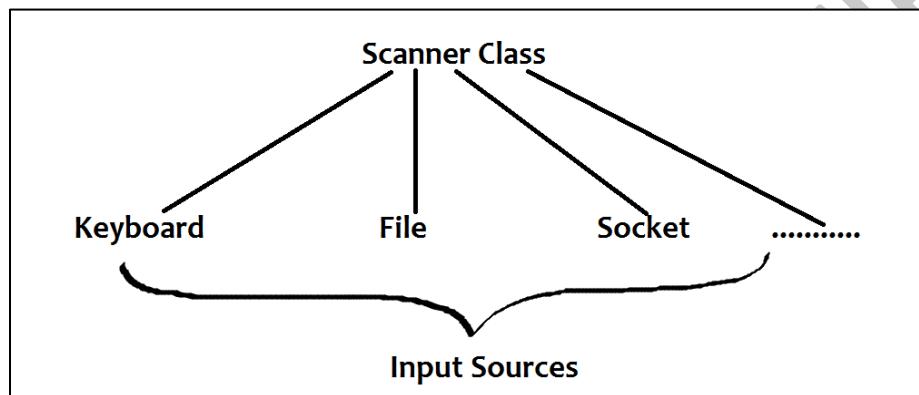
For example...

```
String s = "100";
byte b = (byte)s;
```

Scanner & Random

Scanner class:

- In C-language, we can read the input from the end-user using `scanf()` function.
- In java, using different classes we can collect input from End-user.
- Scanner class can capable of reading information from different resources.
- Scanner class available in “`java.util`” package. Hence we need to “import” Scanner class while using in the java application.
- Scanner class since JDK 1.5



While constructing Scanner class Object, we need to specify Input source.

```
import java.util.Scanner ;  
class ScannerDemo  
{  
    public static void main(String[] args)  
    {  
        Scanner obj = new Scanner(); //CE : we need to specify input source.....  
    }  
}
```

How to provide KEYBOARD input source to Scanner class Object?

```
class System  
{  
    public static final InputStream in  
    {  
        The "standard" input stream.  
        This stream is already open and ready to supply input data.  
        It represents pre-defined Keyboard Object.  
    }  
}
```

Note : Scanner class is having number of pre-defined non-static methods to read different kinds of input from End-user.

```

/*
class Scanner
{
    public Scanner(InputStream source)
    {
        Constructs Scanner class object from the specified input stream.
    }
    public int nextInt()
    {
        Scans the next token of the input as an int.
    }
}
*/
import java.util.Scanner ;
class Reading
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter one integer : ");
        int x = sc.nextInt();
        System.out.println("Input value is : "+x);
    }
}

```

Program :

```

import java.util.Scanner ;
class ScannerDemo
{
    public static void main(String[] args)
    {
        Scanner obj = new Scanner(System.in); //Constructs Scanner obj with input
        source "keyboard".

        System.out.println("Enter 2 integers : ");
        int x = obj.nextInt();
        int y = obj.nextInt();

        int z = x+y ;
        System.out.println("sum : "+z);
    }
}

```

Random class:

- 1) Available in java.util package.
- 2) Since jdk 1.0
- 3) Used to get System generated random numbers.
- 4) While creating “Scanner-class” object, we must specify “InputStream”.
- 5) But while creating “Random-class” object, no need to specify any “Input” source, because System will generate random numbers.

```
/*
class Random
{
    public Random()
    {
        Creates Random integer generator
    }
    public int nextInt()
    {
        Returns a random integer number.
        int --> 4 bytes ---> 32 bits ---> 2^32
    }
}
```

Program:

```
import java.util.Random ;
class RandomNumbers
{
    public static void main(String[] args)
    {
        Random r = new Random();

        int x = r.nextInt(); //generate random value in integer limits.
        System.out.println("x value : "+x);

        int y = r.nextInt(100); //generate random value between 0-100.
        System.out.println("y value : "+y);
    }
}
```

Program to generate a random number within the range of given integer limit:

```
/*
class Rando
{
    public int nextInt(int n)
    {
        Returns a random int value between 0(inclusive) and the specified value
        (exclusive), drawn from this random number generator's sequence.
    }
}
```

```
        }  
    }  
*/
```

Program:

```
import java.util.Random;  
class RandomDemo  
{  
    public static void main(String[] args)  
    {  
        Random obj = new Random();  
        int no = obj.nextInt(100);  
        System.out.println(no);  
    }  
}
```

Try these examples:

- 1) Generate 20 random integers.
- 2) Generate 20 random integers of 3 digits. (121 , 345 , 998 , 756)

Naresh Technologies

Command Line Arguments

- Every Java Application is allowed to take user-input from the command prompt while invoking application.
- Java application can be compiled and run only from the command prompt (OS environment).
- While invoking the application, we can input any type of data(arguments) that can be processed at later time in the application.
- main () method is responsible to collect all these arguments and stores into an array of type String. Hence prototype is

main(String arguments_list[])

Question : Why arguments array is of type String only?

Answer : main(int[]) → Accepts only integer data

 main(float[]) → Accepts only float data

 main(String[]) → Can accept any type of data.

 For example, “10”, “34.56”, “g”, “hari”

- Whatever the input type collected from the command prompt will be implicitly converted into String type.
- To process the command line arguments, it is mandatory to convert String type into corresponding primitive types using **parseXxx()** methods.
- If we don't pass any arguments, the array size will be "zero".
- The size increases depends on number of arguments passed.

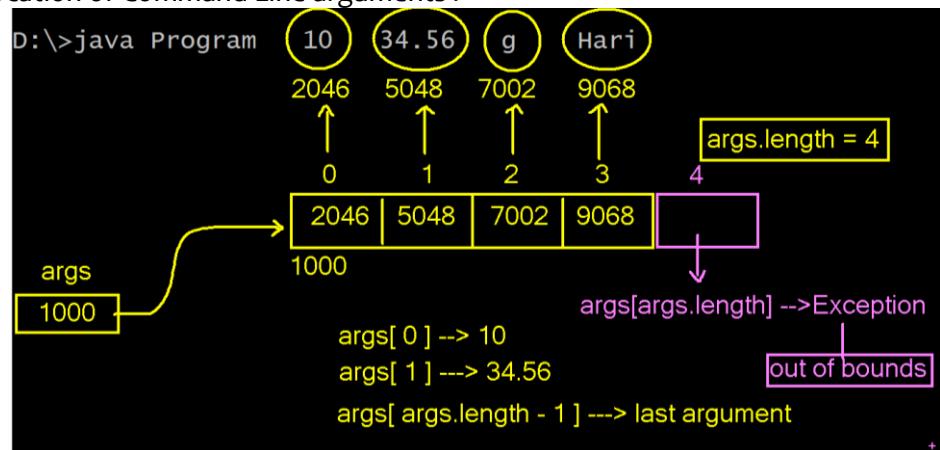
In C or C++, if we are not working with command line arguments, main method will be..

```
main(){}  
void main(){}  
  
In C or C++, if we are working with command line arguments..  
main(int argc, char* argv[ ])  
  
In java, Whether or not you are working with command line arguments, it is mandatory to specify the  
signature of main() method.  
main(String args[ ])
```

How to pass command line arguments?

- While invoking application from the command prompt, we can pass arguments.
- No limit for command line arguments
- Arguments separated using “space” while passing.
- Arguments must be placed followed by “class file” name.

Memory allocation of Command Line arguments :



```
class Arguments
{
    public static void main(String list[])
    {
        int len = list.length ;
        System.out.println("Number of arguments : "+len);
    }
}
```

```
D:\>javac Arguments.java
D:\>java Arguments
Number of arguments : 0

D:\>java Arguments 34 56.78 g Srinivas
Number of arguments : 4
```

```
class Arguments
{
    public static void main(String args[])
    {
        int len = args.length ;
        if(len == 0)
        {
            System.out.println("Please pass few arguments...");
        }
        else
        {
            System.out.println("List of arguments : ");
            for(int i=0 ; i<len ; i++)
            {
                System.out.println("args["+i+"] --> "+args[i]);
            }
        }
    }
}
```

```

        }
    }
}

D:\>javac Arguments.java

D:\>java Arguments
Please pass few arguments...

D:\>java Arguments 34 56.78 g Srinivas
List of arguments :
args[0] --> 34
args[1] --> 56.78
args[2] --> g
args[3] --> Srinivas

```

```

class Arguments
{
    public static void main(String args[ ])
    {
        System.out.println("First arg : "+args[0]);
        System.out.println("Last arg : "+args[args.length-1]);
        System.out.println("Out of array : "+args[args.length]);
    }
}

```

```

D:\>javac Arguments.java

D:\>java Arguments 34 56.78 g Srinivas
First arg : 34
Last arg : Srinivas
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
at Arguments.main(Arguments.java:7)

```

Reading Values from Command Line and performs Addition operation:

The data read from the Command line will be implicitly converted into String type. Hence to process those arguments in application, we must convert these values into Primitive type using pre-defined methods of Wrapper classes (String->Primitive conversion).

```

class CmdLineArgs
{
    public static void main(String args[ ])
    {
        int a,b,c;
        a = Integer.parseInt(args[0]);
        b = Integer.parseInt(args[1]);
        c = a+b;
    }
}

```

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

```
G:\>java CmdLineArgs 33 55
33 + 55 = 88

G:\>java CmdLineArgs 33 55 77
33 + 55 = 88

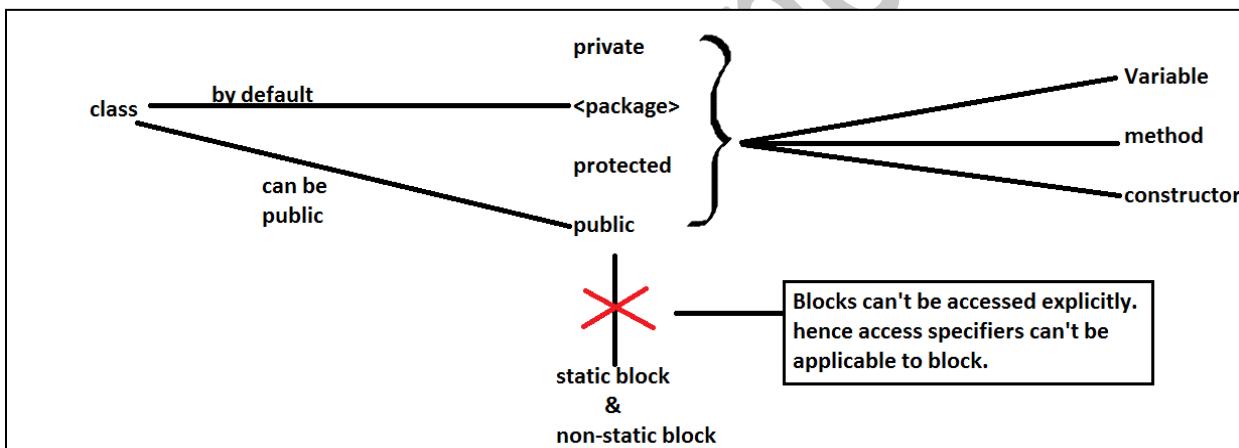
G:\>java CmdLineArgs 33
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
at CmdLineArgs.main(CmdLineArgs.java:7)
```

Access Modifiers

- An access modifier is a keyword/modifier.
- Access modifiers are differ from general modifiers.
- Access modifiers used to set access permissions on class members.
- Java supports four Access modifiers which are....
 1. public
 2. private
 3. <package> or <default>
 4. protected.

Note the followings....

- Access modifiers can be applied only to the members which we can access in the java application(class, variables, methods and constructor).
- Access modifiers cannot be applied to blocks such as static block, non-static blocks.
- Blocks execute automatically by the JVM in the application.



private:

- Private member belongs to particular object. Complete hiding from outside world.
- We can't apply to class.
- If we apply to class, complete object will be hidden; hence no one can communicate with that object. Such type of objects is useless.
- We can apply private functionality only to class members such as variable, methods & constructors.
- Private members of class can be accessed only within that class in which those are defined.

public:

- We can apply public modifier to class and its members.
- Public class can be accessed anywhere in java application.
- In real time applications, all the classes must be defined as public.

<package> :

- Can be applied to class members.
- By default every class and its member is of package level.
- It is the default access modifier in java application.
- Package level members can be accessed only within the package.

protected:

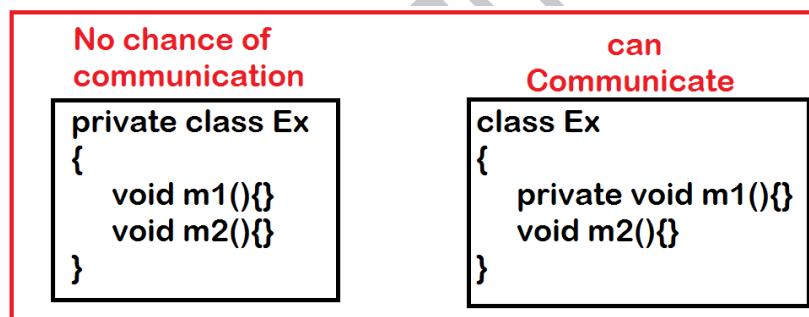
- Can be applied only to class members.
- protected members can be accessed within the package and only from the subclasses of outer package.

Question : Why we can't apply modifiers to blocks?

Answer: we can apply access modifiers to members which we can access in the application.
Blocks we cannot access explicitly.

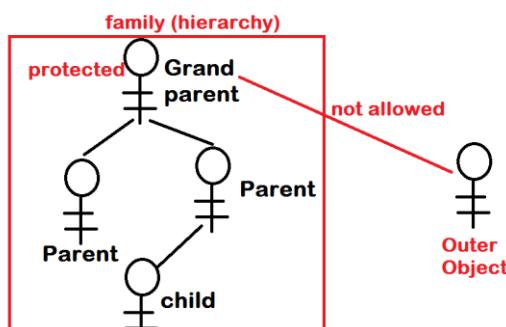
Question : Why a class cannot be private?

- A class represents object.
- Private members can be accessed within the class(by same object itself).
- If class is private, the entire object is not visible to outside world for communication.
- Such type of objects(complete private) are useless.



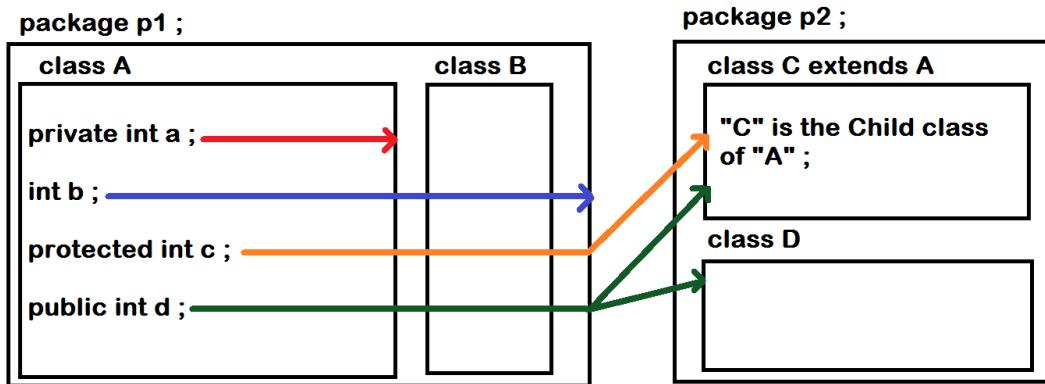
Why class cannot be protected?

- Protected members can be accessed only within the Hierarchy (family of classes).
- If a class is protected, the object functionality can be accessed only within the hierarchy in its entire life time.
- Such type of objects cannot exist in the real world.
- Hence we cannot apply protected to class.



Accessibility of Modifiers:

- The following diagram describes clearly about where we can apply access modifiers in java application.
- In this diagram we are using two packages to explain the concept easily.
- About packages, we will discuss briefly in the following concept.



Code1:

```
class PrivateMembers
{
    private static int a = 10;
    public static void main(String[] args)
    {
        System.out.println("a :" +PrivateMembers.a);
        PrivateMembers pm = new PrivateMembers();
        pm.method();
    }
    private void method()
    {
        System.out.println("Private method");
    }
}
```

Code2:

```
class Test
{
    private static int a = 10 ;
}
class PrivateMembers
{
    public static void main(String[] args)
    {
        System.out.println(Test.a); // Error : Private member
    }
}
```

Code3:

```
private class PrivateMembers
{
    public static void main(String[] args)
    {
        System.out.println("A class can't be private...");
    }
}
```

Code4:

```
protected class Test
{
    public static void main(String[] args)
    {
        System.out.println("A class can't be protected...");
    }
}
```

Code5:

```
/*These 2 classes belongs to default package*/
class Test
{
    private void m1(){}
    void m2(){} //package level
}
class PrivateMembers
{
    public static void main(String[] args)
    {
        Test obj = new Test();
        obj.m2();
        obj.m1(); // Error:
    }
}
```

Code6:

- To dis-allow outside class to instantiate, constructor must be defined as private.
- Without constructor calling, no one can create object in java application.

```
class Test
{
    private Test()
    {
        //empty
    }
}
class PrivateMembers
```

```

{
    public static void main(String[] args)
    {
        Test obj = new Test(); //can't create object -> private constructor
    }
}

```

Fill this diagram using your access modifiers knowledge:

Access modifier	with in the class	with in the package	sub class of same package	sub class of outside package	Outside package
private					
<package>					
public					
protected					

Question: Can we apply access modifiers to local variables?

Answer: No

```

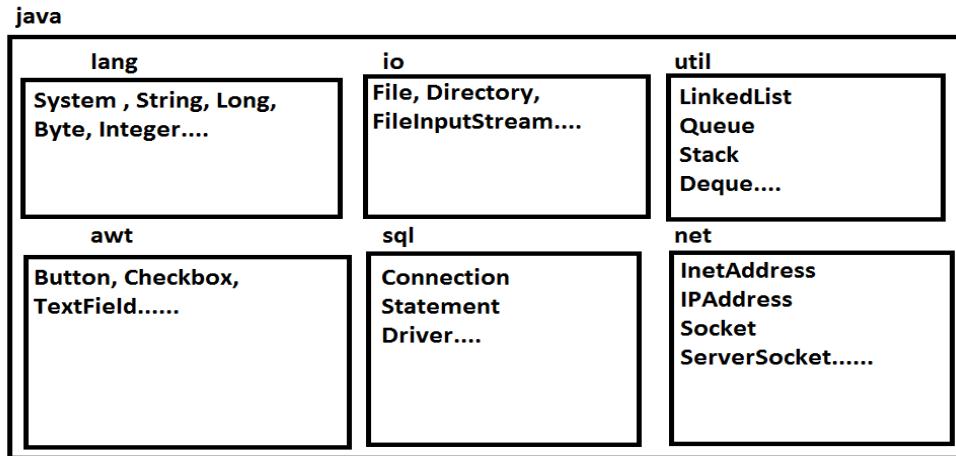
class Test
{
    public static void main(String[] args)
    {
        private int a = 100 ; // Error : not allowed here
        System.out.println("a value : "+a);
    }
}

```

Packages

- Package is a java folder or directory.
- Package is a related group of classes, abstract classes and interfaces.
- Generally in our System we are creating folders to separate the related information. So that we can access effectively.
- Java API (library) is having around 5000 pre-defined classes.
- Every class belongs to a specified pre-defined package.
- Package naming convention “SINGLE WORD, SMALL CASE“. Examples..... java, lang, nit, io.....
- “java” is the super package of all.

Pre-defined API structure:



Note : “lang” package classes available directly to every java application. These classes mainly used by basic java programmer.

System, String, Byte, Integer.... belongs to “lang” package....

How to create package?

“package” is the pre-defined keyword to create user-defined packages in java application.

Syntax :

```
package <Identity>;
```

Example :

```
package nit;
```

Note : Package statement must be the first statement in java application.

```
package nit;  
import java.lang.System ; //optional  
class First  
{
```

```

public static void main(String args[ ])
{
    System.out.println("nit.First class main method.....");
}

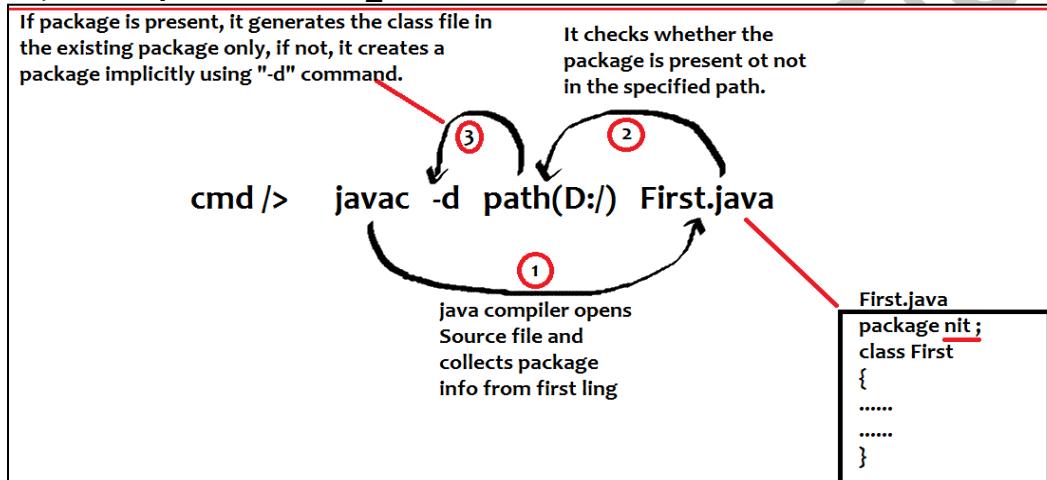
```

Compiling java program including package statement : After setting java path to JDK kit, go to command prompt and type

Cmd/> javac.exe

Provides information about javac.exe

Usage : javac <options> source_files....



```

Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\welcome>d:
D:\>javac -d d:/ First.java — compile
D:\>java nit.First — run

```

Accessing the members of one class from another class of same package:

- Default access permissions in Java application is package level.
- Hence we can access one class members from another class directly.

```

package nit;
import java.lang.System; //optional
class First
{
    void fun( )
    {
        System.out.println("nit.First class fun method.....");
    }
}

```

```

D:\>javac -d d:/ First.java — compile
D:\>java nit.First — run
Error: Main method not found in class nit.First, please define the main
  s:
    public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
D:\>

```

main method not found

```

package nit ;
import java.lang.String;
import java.lang.System;
class Second
{
    public static void main(String[] args)
    {
        System.out.println("Second class main method");
        First obj = new First();
        obj.fun();
    }
}

```

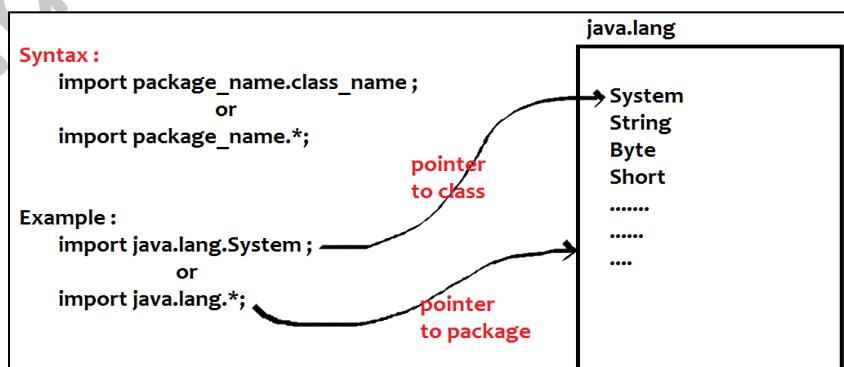
```

D:\>javac -d d:/ First.java
D:\>javac -d d:/ Second.java
D:\>java nit.Second ————— We should run Second.class file(contains main
  method)
Second class main method
nit.First class fun method.....

```

import:

1. It is a pre-defined keyword.
2. Used to connect classes in java application of different packages.
3. Import statement creates an internal pointer either “to package” or “to class”.
4. Using internal pointer, it loads the class dynamically while application is running.



- Only “public classes” can be accessed from outer package.

private class	not allowed
<package> class	can be accessed within package
protected class	not allowed
public class	can be accessed from outer package

- If class is package level, compiler supplies “package level” default constructor.
- If class is public, default constructor is also public.

```
public class TestProgram
{
    // empty...
}
```

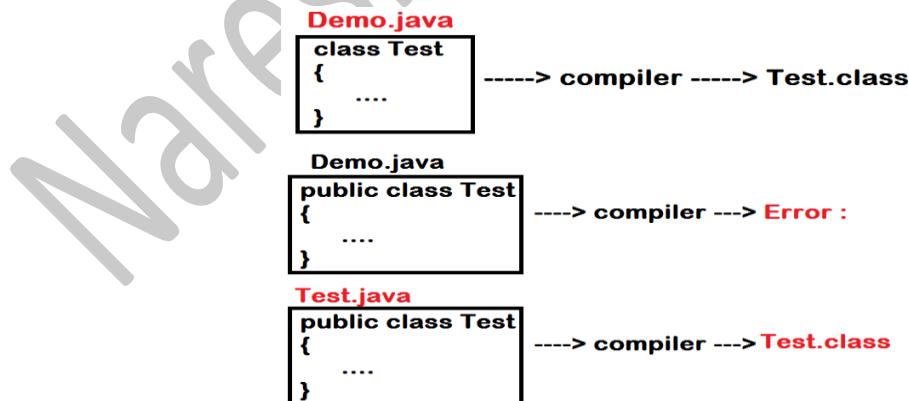
```
D:\>javac TestProgram.java

D:\>javap TestProgram
Compiled from "TestProgram.java"
public class TestProgram {
    public TestProgram();
}

D:\>
```

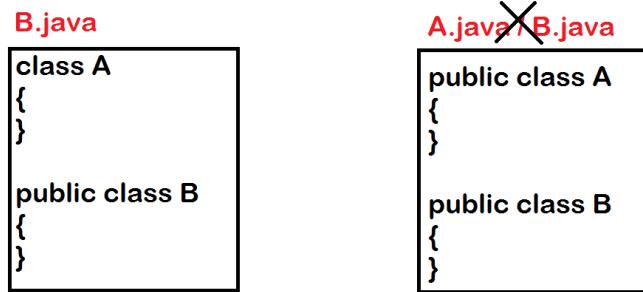
compiler written constructor

- If class is public, class name and file name should be same.
- We cannot define 2 “public” modifier classes in a single java source file.



Question : Can we place more than one public class in a single source file?

Answer : not allowed.



Note : if class is not public, we must save the class with any name.

Demo.java :

```

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

```

```

Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\welcome>d:

D:>javac Demo.java

D:>java Check
Hello World!

```

Note: If class is “public”, we must save with the same name of class name.

File Edit View Search Document Project Tools Window Help EditPlus - [D:\Demo.java]

```

1 public class Check
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Hello World!");
6     }
7 }
8

```

```

D:\>javac Demo.java
Demo.java:1: error: class Check is public, should be declared
check.java
public class check
 ^
1 error

```

One.java

package p1 ; public class One	package p2 ; import p1.One ; //to import, class must be public
----------------------------------	---

Two.java

```
{
//it is having public default constructor.....
public void fun()
{
    System.out.println("p1.One fun method.....");
}
}
```

```
class Two {
public static void main(String[] args)
{
    System.out.println("p2.Two main method....");
    One obj = new One(); //invokes public
constructor.
    obj.fun(); //allowed to access, it is public.....
}
```



D:\>javac -d . one.java
D:\>javac -d . Two.java
D:\>java p2.Two
p2.Two main method....
p1.One fun method.....

Note:

- “import” statement doesn’t load the class directly.
- It creates only internal pointer to package or class.
- “import” statement only loads the class at runtime, on use.
- un used classes will not be loaded into JVM.

Practical example that proves, “import” statement doesn’t load the class directly:

- As a java programmer, we can analyze whether a class is loading or not into JVM, we need to define a static block.
- Static block executes implicitly at the time of class loading.

```
package p1;
public class One
{
static
{
    System.out.println("p1.One class loading");
}
public One()
{
    System.out.println("p1.One object creating");
}
}
```

```
package p2 ;
import p1.One ;
class Two
{
static
{
    System.out.println("p2.Two class is loading.");
}
public static void main(String[] args)
{
    System.out.println("p2.Two main method");
    new One(); // it loads "One-class" first.
}
}
```

```

C:\Windows\system
D:\>javac -d . One.java
D:\>javac -d . Two.java
D:\>java p2.Two
p2.Two class is Loading.....
p2.Two main method...
p1.One class loading.....
p1.One object creating.....

```

Fully Qualified Name:

- Writing full package name while using any class.
- We can replace “import” statement with “Fully qualified name”.

```

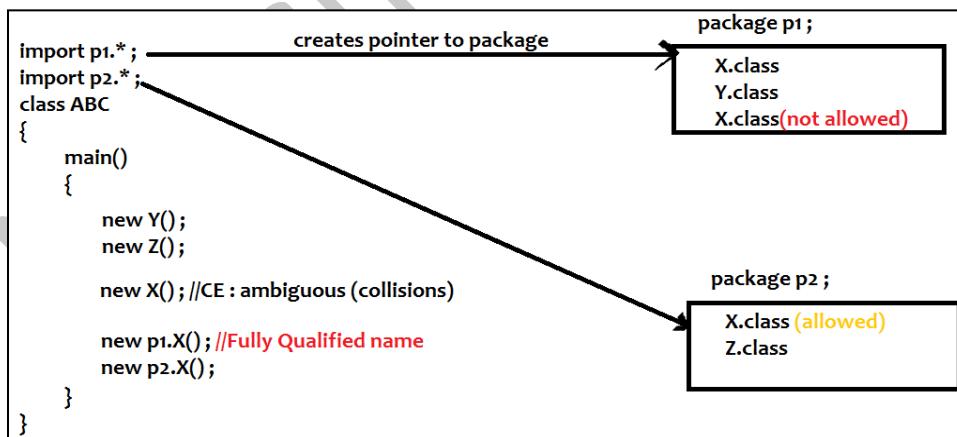
//import java.util.Random;
class FullyQualified
{
    public static void main(String[] args)
    {
        java.util.Random obj = new java.util.Random();
    }
}

```

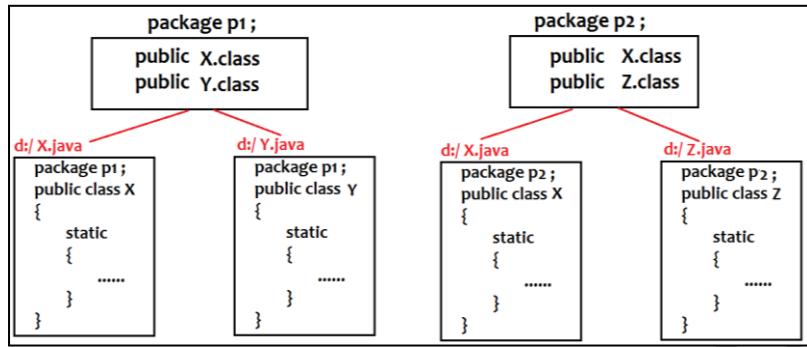
fully qualified name

Advantages:

- The main advantage of packages is, “**Avoiding collisions between class names**”.
- Consider, we need to define 2 classes with the same identity in java app, we need to place these 2 classes in 2 different packages.
- And we can access only by using “Fully Qualified Name”.



- Above application can be practiced as follows.
- We need to write two X.class files separately.
- Create p1 package with X,Y classes and delete the source files because we need to create one more X.class file in p2 package.



```

package p1;
public class X {
    static {
        System.out.println("p1.X is loading....");
    }
}

package p1;
public class Y {
    static {
        System.out.println("p1.Y is loading....");
    }
}

package p2;
public class X {
    static {
        System.out.println("p2.X is loading.... ");
    }
}

package p2;
public class Z {
    static {
        System.out.println("p2.Z is loading.... ");
    }
}

import p1.*;
import p2.*;
class ABC {
    public static void main(String[] args) {
        new Y();
        new Z();
    }
}

```

```
C:\Windows\system32\cmd.exe
D:\>javac -d . X.java
D:\>javac -d . Y.java
D:\>javac -d . X.java
D:\>javac -d . Z.java
D:\>javac -d . ABC.java
D:\>java p3.ABC
p1.Y is loading.....
p2.Z is loading.....
D:\>
```

```
package p3;
import p1.*;
import p2.*;
class ABC
{
    public static void main(String[] args)
    {
        new X(); //CE : collisions
    }
}
```

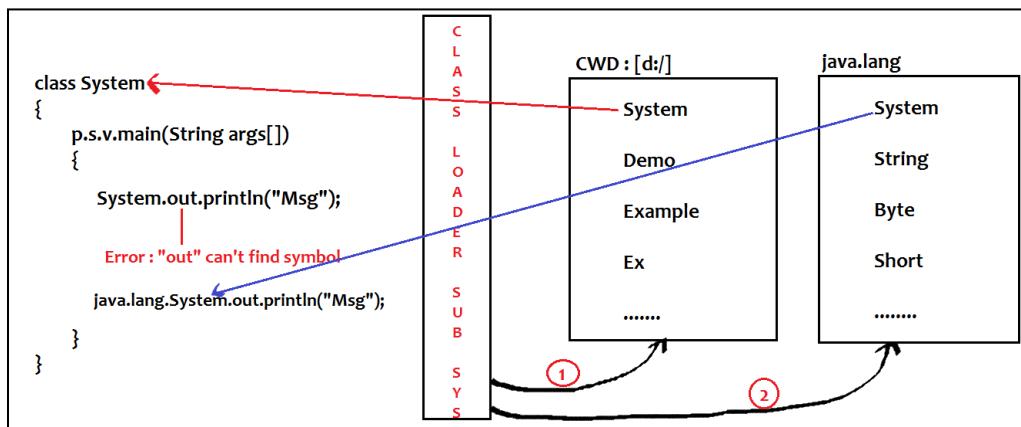
```
C:\Windows\system32\cmd.exe
D:\>javac -d . ABC.java
ABC.java:8: error: reference to X is ambiguous
          new X();
                  ^
      both class p2.X in p2 and class p1.X in p1 match
1 error
```

```
package p3;
import p1.*;
import p2.*;
class ABC
{
    public static void main(String[] args)
    {
        new p1.X();
        new p2.X();
    }
}
```

```
C:\Windows\system32\cmd.exe
D:\>javac -d . ABC.java
D:\>java p3.ABC
p1.X is loading.....
p2.X is loading.....
D:\>-
```

Question: Can we define a class with the same name of java-API class?

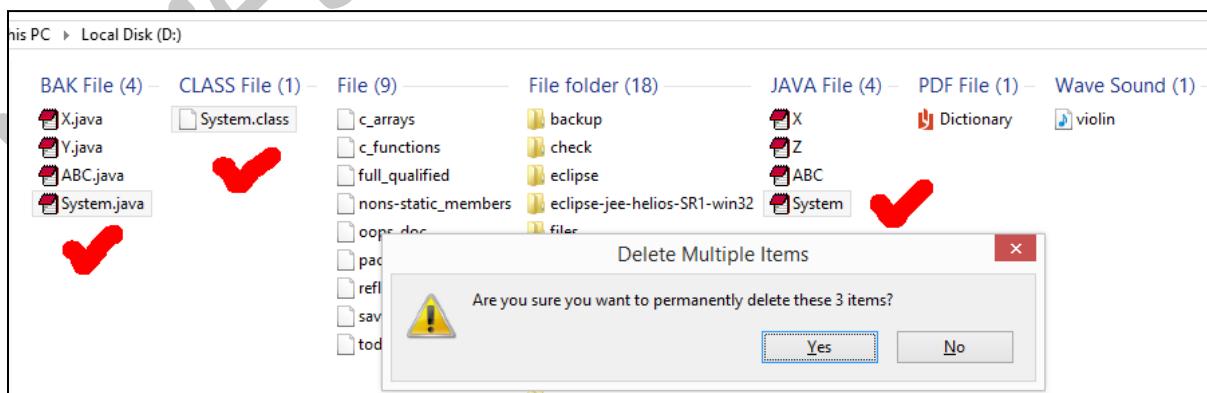
Answer: Yes allowed.....



- Class loader sub system is the pre-defined program.
- It loads the class into JVM
- It gives first priority to Current Working Directory (local drive).
- If class is not present in CWD, then it searches in API.

```
class System
{
    public static void main(String[] args)
    {
        //System.out.println("Hello World!"); //CE :
        java.lang.System.out.println("Hello World!");
    }
}
```

Note: After practiced above application, please delete System related files, if not, every time it collects the information of System class from local drive.



```
class String
{
```

```

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

```

```

C:\Windows\system32\cmd.exe
D:\>javac String.java
D:\>java String
Error: Main method not found in class String, please define the main method as:
  public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application

```

```

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

```

Creation of Sub packages : Using “package” keyword, we can create sub packages.

Syntax :

```
package parent_package.child_package;
```

Example :

```
package maths.operators ;
package maths.operators.arithmetic ;
```

Note: In the process of sub package creation, the required directories to generate the class file will be created implicitly by the Compiler.

Current Working Directory (D:/)

```
package p1;
```

```
package p2;
```

```

public class Sample
{
    void fun()
    {
        .....
    }
}
```

```
package p2;
```

```

import p1.p2.Sample ;
public class Access
{
    p.s.v.main()
    {
        Sample obj = new Sample();
        obj.fun();
    }
}
```

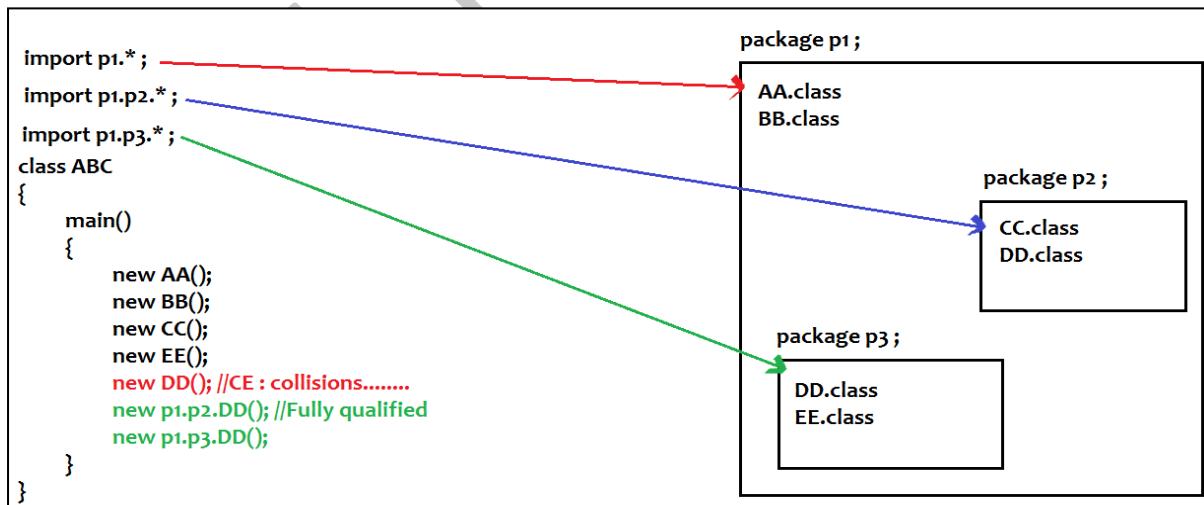
<pre>package p1.p2; public class Sample { public void fun() { System.out.println("p1.p2.Sample class method...."); } }</pre>	<pre>package p2 ; import p1.p2.Sample ; class Access { public static void main(String[] args) { Sample obj = new Sample(); obj.fun(); } }</pre>
--	---

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\welcome>d:
D:\>javac -d . Sample.java
D:\>javac -d . Access.java
D:\>java p2.Access
p1.p2.Sample class method....
```

Importing sub packages:

- If we import only Parent package, internal pointer creates to parent package only then we access only Parent package classes.
- To access the members of Child package, we must import child package explicitly.



Object Oriented Programming

Is java fully Object Oriented Programming Language?

Answer: No.

- 1) Supporting primitives
- 2) Not supporting all the features of OOPS (multiple inheritance)

Object Oriented Programming features are...

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

Notes:

- OOP features neither belong to java nor to any programming language.
- Every language can adopt OOP features to become Object Oriented Language.
- Implementation of OOP functionalities in java only through "class" and "Object".

Object: A real world entity (physical substance) having 3 things

- 1) Identity of object.
- 2) State / Properties / **Variables** / Data / Data members / Fields
- 3) Behavior / Functionality / **Methods** / Member Functions / Code

for example :

Object1: Human

Identity: Amar

Properties: Color, Height, Weight, Age, Qualification...

Functionalities: walk() , see() , run() , teach() , swim() , drive()....

Object2: Laptop

Identity: Lenovo

Properties: Color, Model , Price , Configuration..

Functionalities: calculate() , play() , store() , send()....

Identity: Every Object should be referred by unique Identity in the application.

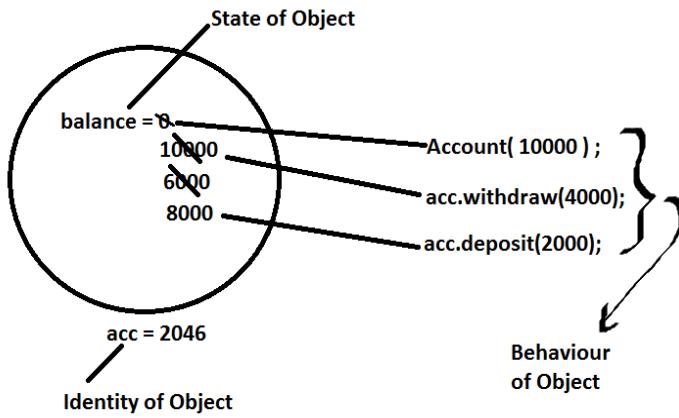
State: Every Object should have properties called state of Object. It can be modified only by Behavior of Object.

Behavior: Functionalities (methods) of Object is called Behavior. **Only Behavior can change state of Object.**

```

class Account
{
    int balance ;
    Account(int amt)
    {
        this.balance = amt ;
    }
    void withdraw(int amt)
    {
        balance = balance - amt ;
    }
    void deposit(int amt)
    {
        balance = balance + amt ;
    }
}

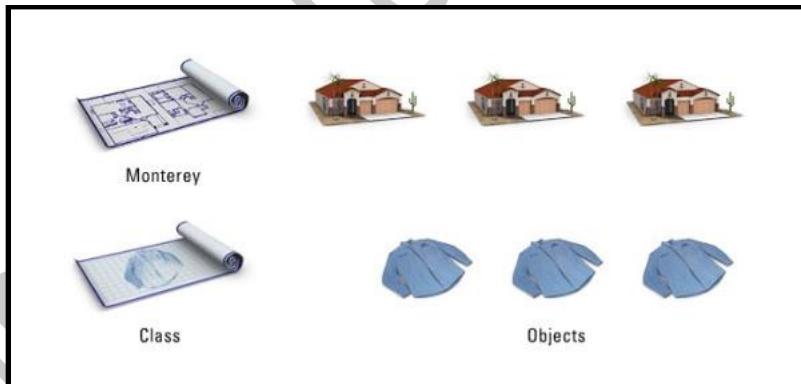
```



Class can be described as:

- Pre-defined keyword.
- User-defined data type(Extension to structure data type).
- Blue-print/Plan/Model of Object.
- Collection of Variables and Methods.
- Collection of objects(Possibility).

Definition: A class is a complete representation of real world entity (Object) which includes



Note: Different type of Objects participate in the process of communication, represented by following classes....

1. POJO class
2. Bean class
3. Mutable class
4. Immutable class
5. Final class
6. Abstract class
7. Interface
8. Factory class

9. Singleton class
10. Exception class
11. Thread class
12. Generic class
13. Annotation
14.

Note: All the above classes description will see in the following concepts...

**POJO rules: In java app, object shoud follow at least POJO rules
(Plain Old Java Object)**

- 1) Class must be public
- 2) Properties must be private
- 3) Should have the definition of public zero arguments constructor.
- 4) Can have optional arguments constructor (if required)
- 5) Every property must having public setter and getter.

Rule-1: Class must be public

- In communication world, one object should directly visible to another object.
- Once object is visible (public), then we can start communication when required.
- Hence every class must be public.

Rule-2: Properties must be private

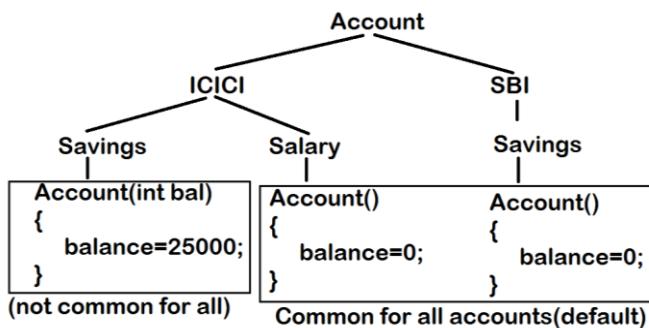
- Only Object should be visible but not properties.
- Human is visible, but not his AGE, Qualification.....
- Laptop is visible, but not its RAM-size, Price.....
- When we declare property as private, no outside object can access.

Rule-3: Should have the definition of public zero args constructor.

- Consider I am taking account from the bank.
- Account balance is implicitly initializes with “zero”.
- Public “zero args” constructor will provide default initialization to object.

Rule-4: Can have the definition of public args constructor.

- For example, some of the banks not allowed to take account with “zero” balance. Examples ICICI, AXIS, HDFC.....
- While constructing Object(taking account), if we want to initialize balance with some value, we must go for “args constructor”.



Rule-5: Every property should have getter & setter

- Properties of object will be private.
- Hence we cannot access properties information directly.
- To access, we need to start communication.
- Communication is possible with the help of “methods”.
- To implement that, in Programming every property should have its own getter & setter method.

Getter method & Setter Method :

- Getter method returns info and doesn't take input
- Setter method takes input but doesn't return output

For example....

```

private int balance = 5000 ;
public int getBalance()
{
    return this.balance;
}
public void setBalance(int balance)
{
    this.balance = balance ;
}
  
```

Example POJO class:

```

public class Emp
{
    private int eno ;
    private String ename ;
    private float esal ;

    public Emp()
    {
        // assign default values...
    }

    public Emp(int eno, String ename, float esal)
    {
        this.eno = eno ;
        this.ename = ename ;
    }
  
```

```

        this.esal = esal ;
    }

    public int getEno()
    {
        return this.eno ;
    }
    public void setEno(int eno)
    {
        this.eno = eno ;
    }

    public String getEname()
    {
        return this.ename ;
    }
    public void setEname(String ename)
    {
        this.ename = ename ;
    }

    public float getEsal()
    {
        return this.esal ;
    }
    public void setEsal(float esal)
    {
        this.esal = esal ;
    }
}

class AccessEmp
{
    public static void main(String[] args)
    {
        Emp e1 = new Emp();
        e1.setEno(101);
        e1.setEname("Hari");
        e1.setEsal(50000);

        Emp e2 = new Emp(102, "Harini" , 60000);

        System.out.println("e1-ename : "+e1.getEname());
        System.out.println("e2-ename : "+e2.getEname());
    }
}

```

Question: Why we need to define public zero args constructor explicitly in POJO class?

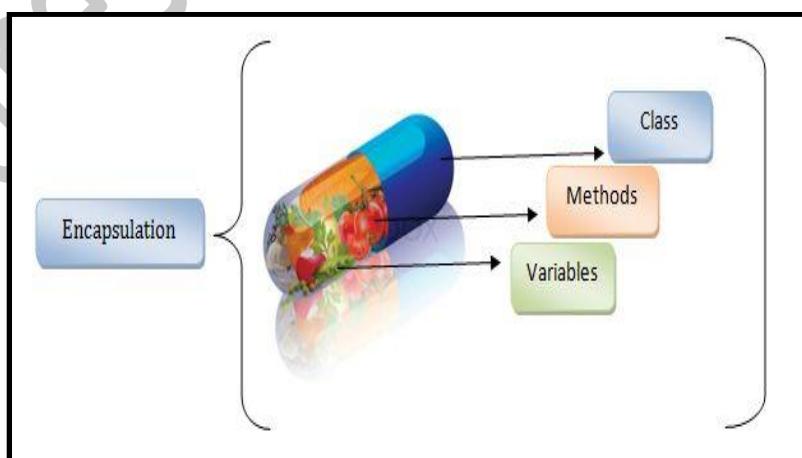
Answer: Compiler supplies default constructor only when we are not defining any constructor in java source file.

```
public class Program
{
    public static void main(String[] args)
    {
        new Program(); // No error : invokes default constructor
    }
}

public class Program
{
    public Program(int x)
    {
        // empty...
    }
    public static void main(String[] args)
    {
        new Program(); // Error : no such constructor
    }
}
```

Encapsulation:

- Writing data (variables) and code (methods) into a single unit.
- In the communication world, we should make outer visibility of Object but not properties and methods.
- Encapsulation is the concept of protecting object functionality from outside view.
- Class is a cap of object as shown in diagram.
- In java Encapsulation can be achieved through a keyword called "class".

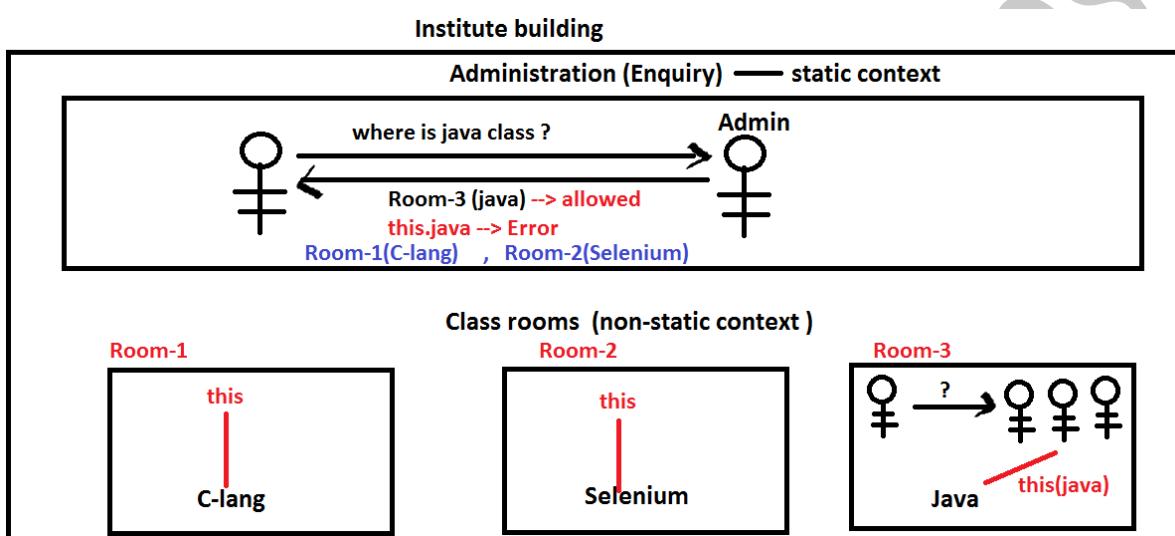


this:

- It is a keyword.
- It is a pre-defined non-static variable.
- It holds reference of Object.
- It must be used only in “non-static context”.
- It is used to access the complete functionality of Object(static & non-static)

Question: Why we need to use “this” only in non-static context?

Ans : we can use “this” only in non-static context to point out a particular object.
From the “static context”, we can point an Object using “user defined reference” variable.



Question : How can we access Non-static members of Object?

- 1) Using Object reference variable(static context)
- 2) Using “this” keyword (non-static context)

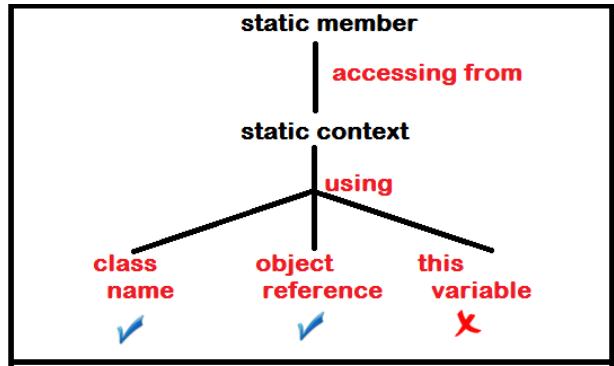
Question : How can we access Static members of Object?

- 1) Using class name (static context / non-static context)
- 2) Using Object reference variable(static context)
- 3) Using “this” keyword (non-static context)

Access class members from different contexts using class_name, object_reference ,this?

- Generally we access “static members” using “class-name”
- But we can use “obj-reference” also to access static members.

Accessing Static members from Static context:



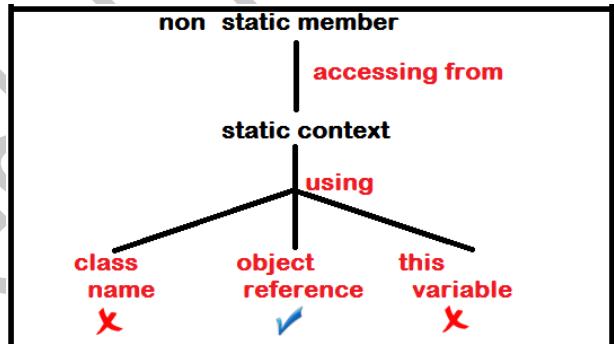
```

class Test
{
    static int a = 100 ;
    public static void main(String[] args)
    {
        System.out.println(Test.a);

        Test obj = new Test();
        System.out.println(obj.a);

        System.out.println(this.a); //Error :
    }
}
  
```

Accessing Non-Static members from Static context :



```

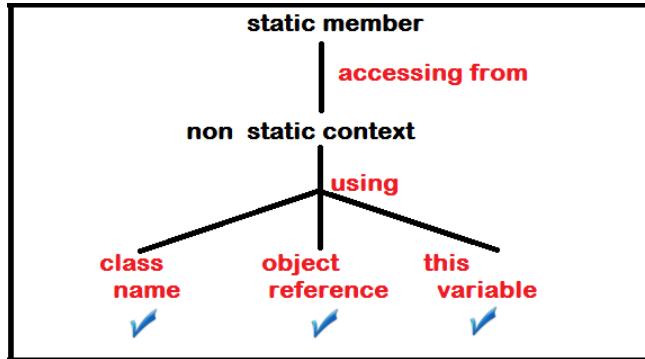
class Test
{
    int a;
    public static void main(String[] args)
    {
        System.out.println(Test.a); // Error :

        Test obj = new Test();
        System.out.println(obj.a);

        System.out.println(this.a); //Error :
    }
}
  
```

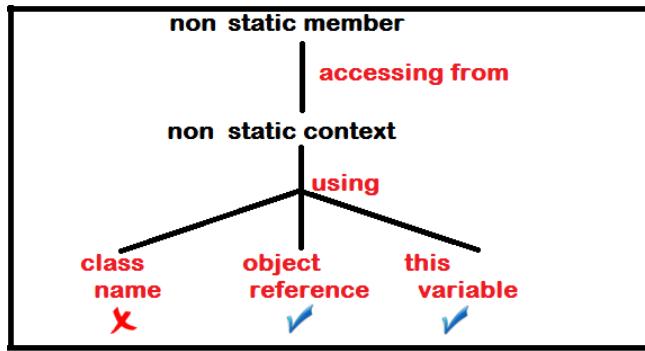
```
    }  
}
```

Accessing Static members from Non-Static context :



```
class Test  
{  
    static int a = 100 ;  
    public static void main(String[] args)  
    {  
        Test obj = new Test();  
        obj.check();  
    }  
    void check()  
    {  
        System.out.println(Test.a);  
  
        Test obj = new Test();  
        System.out.println(obj.a);  
  
        System.out.println(this.a);  
    }  
}
```

Accessing Non-Static members from Non-Static context:



```
class Test  
{
```

```

int a;
public static void main(String[] args)
{
    Test obj = new Test();
    obj.check();
}
void check()
{
    // System.out.println(Test.a); // Error :

    Test obj = new Test();
    System.out.println(obj.a);

    System.out.println(this.a);
}
}

```

Note : In the above application, we are creating 2 objects(waste of memory). It is recommended to create only one object globally(static).

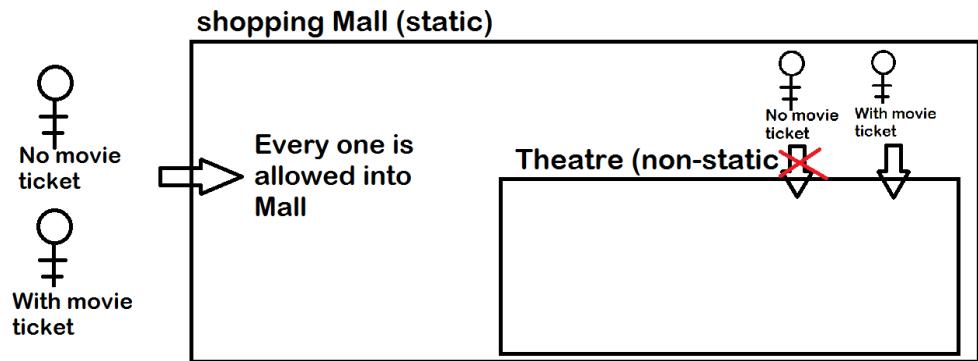
```

class Test
{
    static Test obj = new Test();
    int a;
    public static void main(String[] args)
    {
        Test.obj.check();
    }
    void check()
    {
        // System.out.println(Test.a); // Error :
        System.out.println(Test.obj.a);
        System.out.println(this.a);
    }
}

```

Question: Why it is allowed to access static members using Object address?

- We can access static members(shopping mall) either by showing permission or not.
- But to access non-static members(theatre), we must show permissions.



```
class Test
{
    static int a = 100 ;
    public static void main(String[] args)
    {
        Test obj = new Test();
        System.out.println(obj.a);
    }
}
```

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

Above 2 programs represents, Object reference variable can access both static & non-static members.

Question: How to access the members if static and non-static variables having same name?

- We can define either static or non-static members in the application with one identity.
- Hence, Object reference variable will not get any confusion while accessing either static or non-static members(because both should not be present with single identity).

```
class Test
{
    static int a = 100 ;
    int a ; // not allowed
}
```

For example....

In bank_app,

Static int branch_code = 12345.
If branch_code is static(common), there is no specific branch_code(non-static)

Int balance ;
If balance is non-static(specific), there is no common balance(static)

this():

- this() is used to invoke current(same) class constructor explicitly.
- Used to initialize an object via multiple constructors.
- Connecting constructors using this(), is called Constructor Chaining.
- We must use this() only inside another constructor of same method.
- Call to this(), must be the first statement in another constructor.

```
class Emp
{
    int eno ;
    String ename ;
    Emp()
    {
        System.out.println("Employee is present....");
    }
    Emp(int eno, String ename)
    {
        this( );
        this.eno = eno ;
        this.ename = ename ;
        System.out.println("Employee initialized....");
    }
    public static void main(String[] args)
    {
        Emp obj = new Emp(101 , "Hari");
    }
}
```

Question: Can we connect more than one constructor from one constructor?

Answer: Not allowed, call to this() must be the first statement.

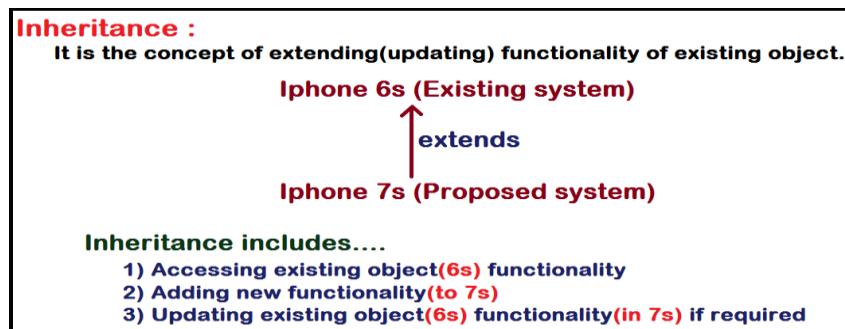
```
class Ex
{
    Ex()
    {
        this(10,20);
        System.out.println("zero args....");
    }
    Ex(int x)
    {
        this( );
        // this(10,20); // Error : we can connect only one constructor
    }
}
```

```
        System.out.println("one args....");
    }
    Ex(int x, int y)
    {
        System.out.println("two args....");
    }
public static void main(String[] args)
{
    new Ex(10);
}
}
```

Naresh Technologies

Inheritance

- Inheritance is the Concept of defining new Object from existing Object which includes...
- It is also called Is-A relation.
- The main advantage of Inheritance is code re-usability.



Terminology:

After implementing "Is-A relation" between Object, we can referred as

- a. Parent class - Child class
- b. Base class - Derived class
- c. Super class - Sub class

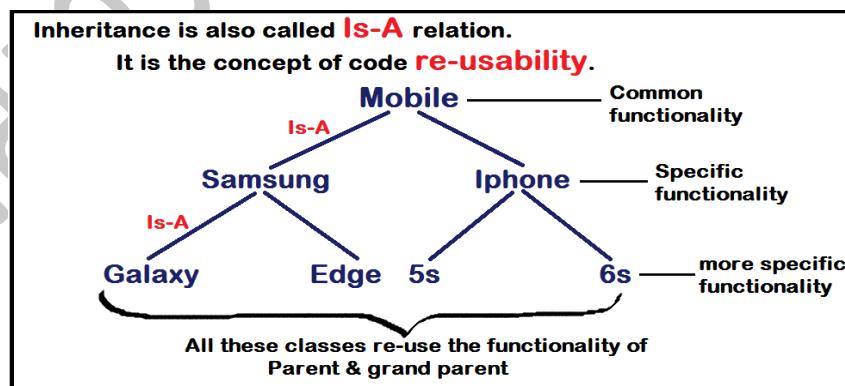
Is-A relation:

After inheriting the complete functionality of existing Object, the Sub Object working like Parent objects also.

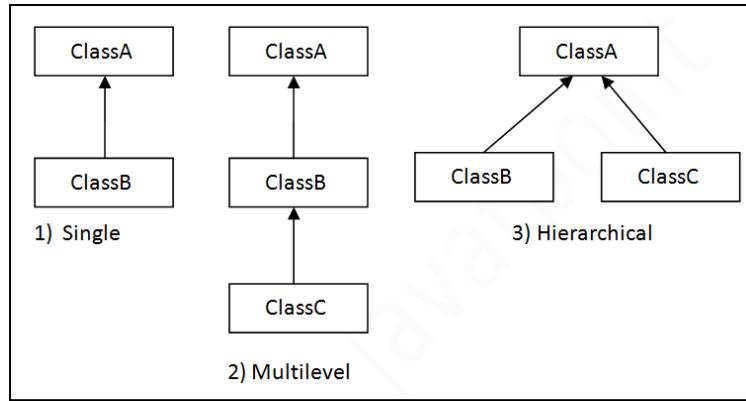
The main advantage of Inheritance concept is “Code Reusability”.

For example...

- 1) Car "is a" Vehicle
- 2) Swift "is a" Car
- 3) Nokia "is a" Mobile

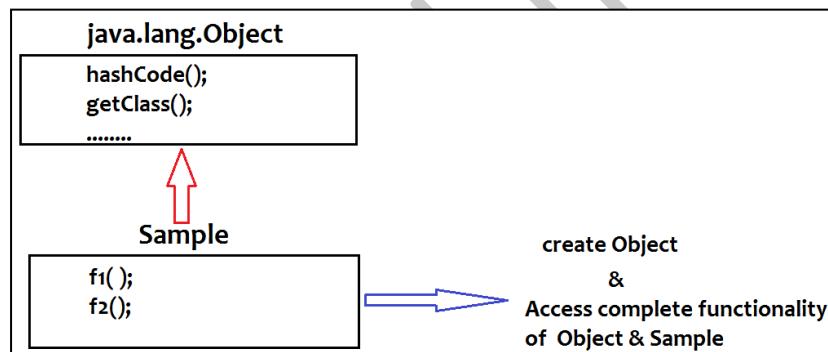


Types of inheritance: Java supports only 3 types of Inheritance out of all the types supported by Object Oriented Programming.



Single/Simple inheritance:

- Every java application silently inherits from “java.lang.Object” class.
- By this extension only, every user object gets the behavior of Real Object.
- Hence every java application implicitly exhibits “Single Inheritance”
- In the process of “Is-A” relation, by creating Object to Child class, we can access the complete functionality of Parent object.



```
/*
class Object{
    hashCode(){}
    getClass(){}
}

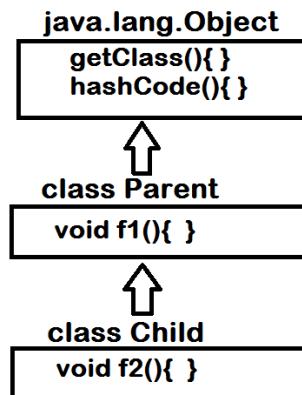
class Sample //extends Object{
    void f1( ){}
    void f2( ){}
}

class SingleInheritance {
    public static void main(String[] args) {
        Sample obj = new Sample();
        obj.f1(); //defined in Child(Sample) class....
        obj.hashCode(); //defined in Parent(Object) class
    }
}
```

```
    }  
}
```

Multi-Level inheritance:

- Accessing the functionality of Objects in more than one level.
- Child class accessing Grandparent functionality through parent.



```
class Parent //extends Object  
{  
    void f1()  
    {  
        System.out.println("Parent functionality.....");  
    }  
}  
class Child extends Parent  
{  
    void f2()  
    {  
        System.out.println("Child functionality....");  
    }  
}  
class MultiLevel  
{  
    public static void main(String[] args)  
    {  
        Child obj = new Child();  
        obj.f2();  
        obj.f1();  
        obj.hashCode();  
    }  
}
```

Note:

- In the process of Child object creation, JVM first creates Parent's Object.

- Once Parent Object has been created, then JVM constructs Child Object based on Parent object.

Question: How can we analyze “JVM instantiating Parent class first, in the process of Child instantiation”?

Ans : As a java programmer we can analyze

- 1) whether “class is loading or not” by defining “static block”
- 2) whether “Object is creating or not” by defining “Constructor”

Note: In the process of Object creation either implicitly (by JVM) or explicitly(by Programmer), constructor is calling is mandatory.

```
class JDK6
{
    JDK6()
    {
        System.out.println("Installing JDK6 features.....");
    }
}
class JDK7 extends JDK6
{
    JDK7()
    {
        System.out.println("Installing JDK7 features.....");
    }
}
class JDK8 extends JDK7
{
    JDK8()
    {
        System.out.println("Installing JDK8 features.....");
    }
}
class MultiLevel
{
    public static void main(String[] args)
    {
        new JDK8();
    }
}
```

Method overriding: (It is the concept of functionality updating)

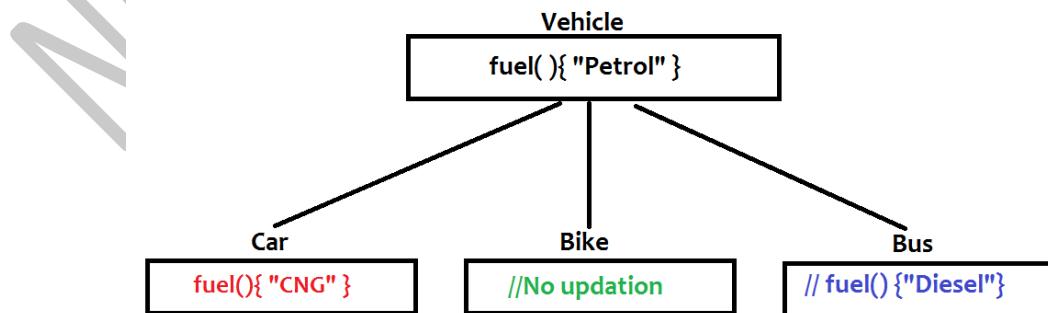
- Defining a method in the “child class” with the same name and signature of its “parent class”.
- Method overriding is the concept of updating existing object functionality if it is not sufficient to new Object.

```

class SamsungGalaxy
{
    String memory()
    {
        return "1gb";
    }
    String camera()
    {
        return "8mp"
    }
}
class SamsungNote extends SamsungGalaxy
{
    String memory() //override (updating existing feature)
    {
        return "2gb";
    }
    String camera()
    {
        return "12mp"
    }
}
class SamsungEdge extends SamsungNote
{
    String memory()
    {
        return "4gb";
    }
    String camera()
    {
        return "16mp"
    }
}

```

Hierarchal Inheritance: Sharing the properties of Object to multiple child objects.



```

class Vehicle
{
}

```

By Srinivas (C/DS/Java trainer)

```

        String fuel()
    {
        return "Petrol";
    }
}
class Car extends Vehicle
{
    String fuel() //override
    {
        return "CNG";
    }
}
class Bike extends Vehicle
{
    // No updation...
}
class Bus extends Vehicle
{
    String fuel()
    {
        return "Diesel";
    }
}
class Hierarchal
{
    public static void main(String args[ ])
    {
        Car c = new Car();
        String cf = c.fuel();
        System.out.println("Car fuel type : "+cf);

        Bike b = new Bike();
        System.out.println("Bike fuel type : "+b.fuel());
    }
}

```

- In the following program we are accessing the functionality of Parent from child by creating object for Parent class.
- Duplicate objects wasting the memory in heap area.
- Solution to this problem is using “super” keyword to access Parent functionality.

```

class Parent
{
    void fun()
    {
        System.out.println("Parent's functionality.....");
    }
}

```

```

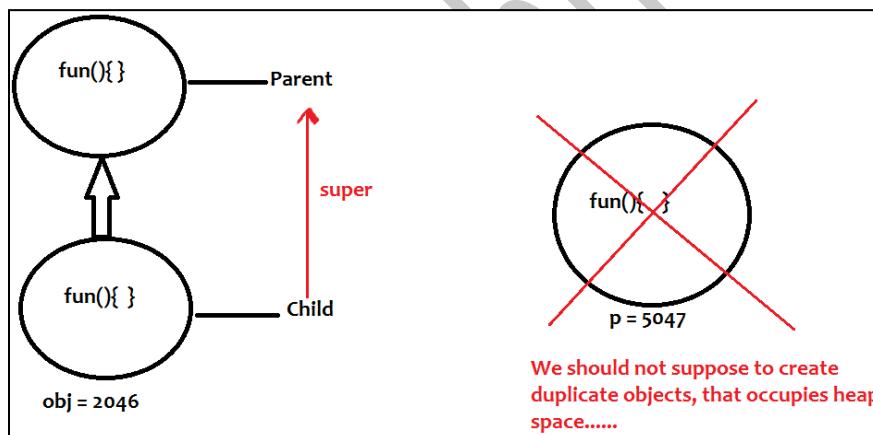
}

class Child extends Parent
{
    void fun()
    {
        System.out.println("Child's functionality.....");
    }
}

class MainClass
{
    public static void main(String[] args)
    {
        Child obj = new Child();
        obj.fun(); //Prints updated functionality......

        Parent p = new Parent();
        p.fun();
    }
}

```



super:

- It is a keyword
- It is pre-defined non-static variable.
- It is used to access the complete functionality of Parent class from Child class.
- It must be used in non-static context.

this	super
It is a keyword	It is a keyword
Pre-defined non static variable	Pre-defined non static variable
It is used to access current object(class) functionality	It is used to access Parent object(class) functionality from child
It must be used only in non-static context	It must be used only in non-static context
It holds object address	It doesn't hold any object address

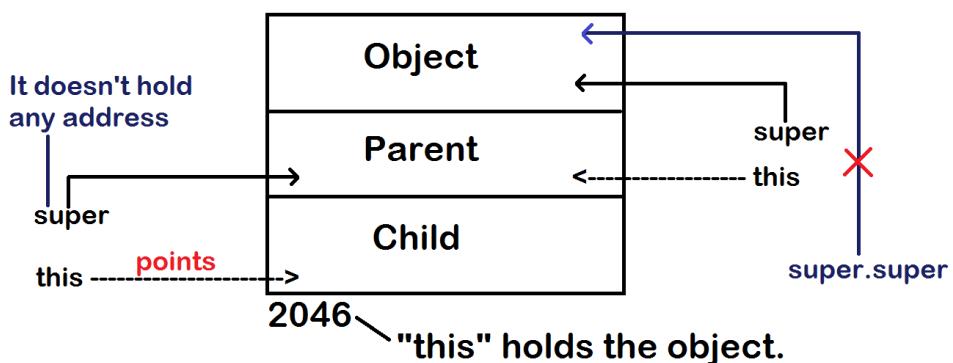
```

class Parent
{
    void fun()
    {
        System.out.println("Parent's functionality.....");
    }
}
class Child extends Parent
{
    Child() //non-static context
    {
        this.fun(); //points to Child class object
        super.fun(); //points to Parent class object
    }
    void fun()
    {
        System.out.println("Child's functionality.....");
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        Child obj = new Child();
    }
}

```

Note :

- “super” is a sub pointer to “this”.
- “this” holds address where as “super” doesn’t.
- Using “super”, we can access only one level object functionality in the hierarchy.



```

class Parent
{
    Parent()
    {

```

```

        System.out.println("Parent's Addr : "+this); // Allowed
    }
}

class Child extends Parent
{
    Child()
    {
        System.out.println("Child's Addr : "+this); // Allowed
        // System.out.println("Parent's addr : "+super); // Error :
    }
    public static void main(String[] args)
    {
        Child obj = new Child();
    }
}

```

super():

- Used to access Parent class constructor from the child class.
- We must use super() method inside the child class constructor
- Call to super() method must be the first statement in the Child's constructor
- In the process of child class object construction, it is not possible to invoke parent's constructor explicitly.
- Hence it is possible to provide initialization of Parent object only by using super() method.

this()	super()
Used to invoke current class constructor.	Used to invoke Parent class constructor from Child
Must be used only inside another constructor of same class.	Must be used only inside Child's constructor.
Call to this() must be the first statement.	Call to super() must be the first statement.
To connect constructors in a single class.	To connect constructors in Is-A relation.
Used to initialize an object via multiple constructors.	Used to initialize parent class object in the process of child object creation.

Note: In the process of child class object construction, JVM implicitly creates Parent's object first. For this construction, JVM uses super() method.

```

class Parent
{
    Parent()
    {
        System.out.println("Parent's instantiation.....");
    }
}

```

```

class Child extends Parent
{
    Child()
    {
        super(); // internal code....
        System.out.println("Child's instantiation.....");
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        Child obj = new Child();
    }
}

```

Question: When we call super() method explicitly?

Answer: In the process of Child object creation it is not possible to invoke Parent class constructor explicitly.

Hence we can't provide initialization to Parent class non-static variables directly.
We use super() method in the child class constructor to provide initialization.

```

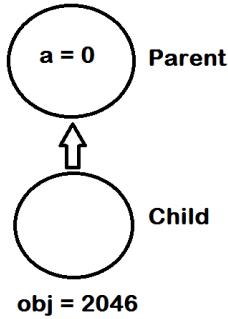
class Parent
{
    int a;
    Parent(int a)
    {
        this.a = a;
    }
}
class Child extends Parent
{
    Child(int a)
    {
        super(a);
    }
    void details()
    {
        System.out.println("Parent's a : "+super.a);
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        Child obj = new Child(100);
    }
}

```

```

        obj.details();
    }
}

```



```

class Parent
{
    int a , b ;
    Parent(int a , int b)
    {
        this.a = a ;
        this.b = b ;
    }
}
class Child extends Parent
{
    int c, d ;
    Child(int a, int b, int c , int d)
    {
        super(a , b);
        this.c = c ;
        this.d = d ;
    }
    void details()
    {
        System.out.println("Parent's a :" +super.a+"\nParent's b :" +super.b+"\nChild's c :
"+this.c+"\nChild's d :" +this.d);
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        Child obj = new Child(10,20,30,40);
        obj.details();
    }
}

```

```

/*
If we don't define any constructor inside the class, compiler supplies a default constructor.
default constructor is a zero args constructor with empty body.

*/
class Test
{
    public static void main(String[] args)
    {
        new Test(); // No error : invokes default constructor.
    }
}

/*
Compiler will not add any default constructor if we define any constructor inside the class.

*/
class Test
{
    Test(int x)
    {
        // empty...
    }
    public static void main(String[] args)
    {
        new Test(); // Error : no such constructor definition
    }
}

```

Note : In case Parent-child relation also, default constructors will be supplied by the compile only if we don't define any constructor.

Hence we need to take care of all the constructor availability in constructor chaining process among Parent-child object creation.

```

class Parent // extends Object
{
    /*Parent()
    {
        super();
    }*/
}
class Child extends Parent
{
}

```

By Srinivas (C/DS/Java trainer)

```

/*Child()
{
    super();
}*/
}

class MainClass
{
    public static void main(String[] args)
    {
        Child obj = new Child();
    }
}

```

Note : In the process of constructor chaining in Parent-Child classes, we need to check all the constructors available or not.

```

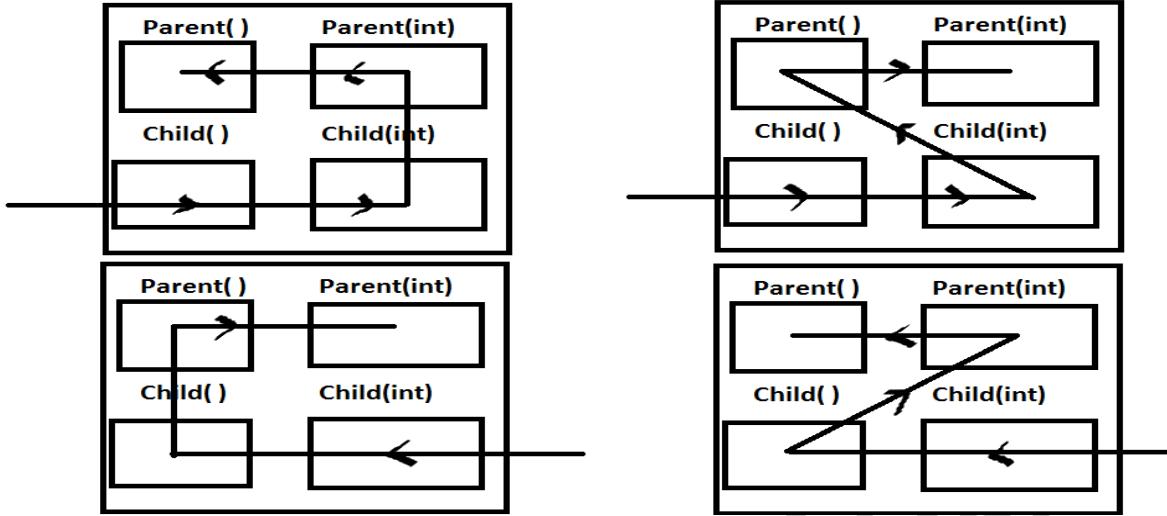
class Parent
{
    // no default constructor
    Parent(int x)
    {
        // empty....
    }
}

class Child extends Parent
{
    /*Child()
    {
        super(); // looking for default constructor in Parent class and results error as it was
not defined.....
    }*/
}

class MainClass
{
    public static void main(String[] args)
    {
        Child obj = new Child();
    }
}

```

- In the following example, we defined 4 constructors both in Parent and Child class.
- We can connect all the constructors in the instantiation of Child class in 4 ways.
- Note that, call to either super() or this() must be the first statement in the constructor.
- That means we can connect only one constructor from another constructor.



One example that describes above diagram (connecting constructors):

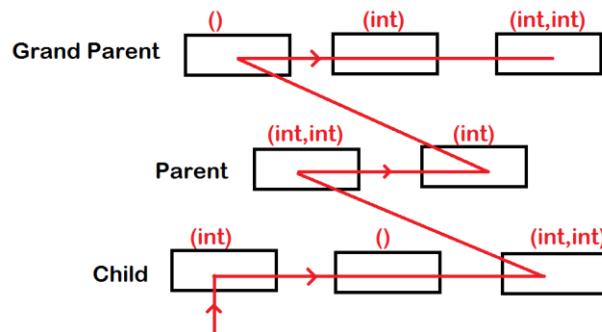
```

class Parent
{
    Parent()
    {
        this(10);
        System.out.println("Parent's zero args");
    }
    Parent(int x)
    {
        System.out.println("Parent's args");
    }
}
class Child extends Parent
{
    Child()
    {
        this(10);
        System.out.println("Child's zero args");
    }
    Child(int x)
    {
        super();
        System.out.println("Child's args");
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        new Child();
    }
}

```

```
    }  
}
```

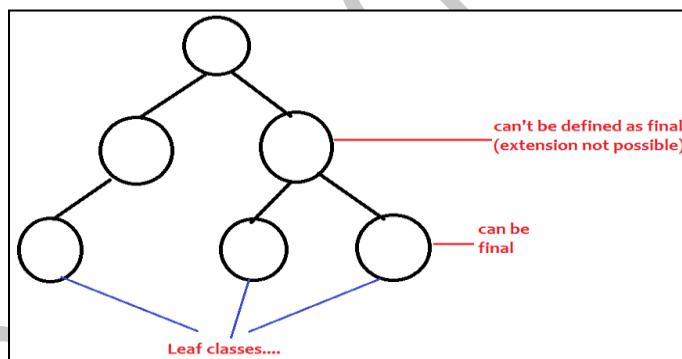
How to connect constructors in Multi level inheritance:



final: Used to set restrictions on object updation.

- It is a keyword.
- It is a modifier.
- It can be applied to Class, Method & Variable.

Note: If class is “final” that cannot be inherited (extended). Only leaf classes in the hierarchy must be defined as “final”.



```
final class First  
{  
    //logic  
}  
class Second extends First //Error:  
{  
    //logic  
}
```

Method is final:

- If method is final, that cannot be overridden.
- Overriding is the concept of updating a specific functionality of Object.
- But if we define any function/method as final, that cannot be updated.

- For example...

```
class Galaxy
{
    final String camera()
    {
        return "12mp";
    }
}
class Edge extends Galaxy
{
    String camera() // Error : can't Override
    {
        return "20mp";
    }
}
```

Variable is final:

- If variable is “final” that cannot be modified.
- Constants in java application must be defined as final.
- Most of the final variables are static in java application.
- For example.....
 - **static final double PI = 3.142**

```
class Test
{
    static final double PI = 3.142 ;
    public static void main(String[] args)
    {
        Test.PI = 3.1412 ; // Error : can't modify
    }
}
```

Note: We cannot update the functionality of “final object”, but we can access the complete functionality of Object.

```
final class FinalClass
{
    static final double PI = 3.142 ;
    final void fun()
    {
        System.out.println("final class final method... ");
    }
}
class Access
{
    public static void main(String args[ ])
```

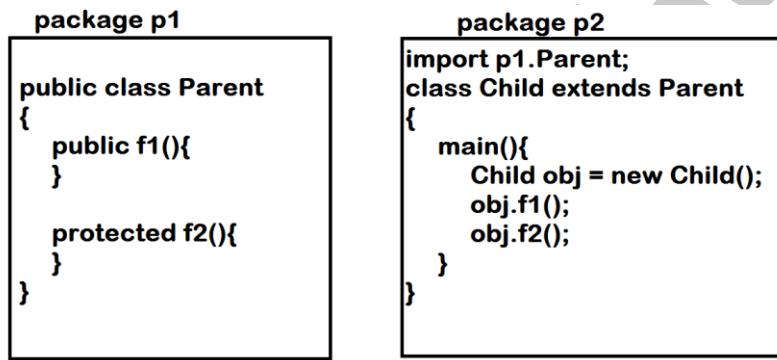
```

    {
        System.out.println("PI value : "+FinalClass.PI);
        FinalClass obj = new FinalClass();
        obj.fun();
    }
}

```

Accessing protected members of super class:

- Protected members of Parent Object must be done through Child Object.
- By just creating Object for the Parent class from the class, it is not possible to access protected members.
- JVM cannot understand the relation between Super and Sub Object if we create only Object of Super class.



```

package p1;
public class Parent
{
    public void f1()
    {
        System.out.println("Parent's public f1");
    }
    protected void f2()
    {
        System.out.println("Parent's protected f2");
    }
}

```

```

package p2;
import p1.Parent;
class Child extends Parent
{
    public static void main(String[] args)
    {
        //Parent p = new Parent(); //if we create Object of Parent class, it is allowed to access
        only public members of Parent class.
        p.f1(); //allowed
    }
}

```

```

        p.f2(); //CE : no relation between Parent & Child.

    Child p = new Child();
    p.f1();
    p.f2();
}
}

```

Question: Can we override the constructor in “Is-A” relation?

- Constructor cannot be overridden.
- Constructor definition belongs to a Particular class.
- Constructors name same as Class name. Hence we cannot override the Parent class constructor into Child class.

```

class Parent
{
    Parent()
    {
    }
}

class Child extends Parent
{
    Parent()
    {
    } ---> Not possible here
}

```

Question: Can we apply “final” modifier to constructor?

- We can't apply final modifier to constructor.
- “final” modifier restricts updating functionality
- As we are not updating constructor anyway, no need to apply final modifier to constructor.

```

class Test
{
    final Test()
    {
        System.out.println("final constructor..");
    }
}

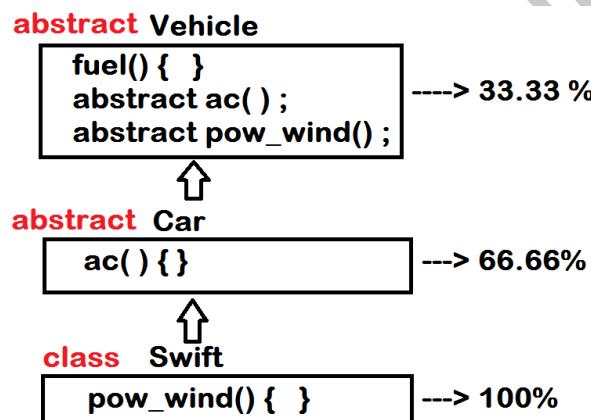
```

Abstraction

- Abstraction is the concept of hiding unnecessary details of Object and shows only essential features.
- Abstraction provides the information about "What object can do instead how it does it".
- Abstraction is the "General View" of Object.

Abstract class:

- If a class is not able to provide complete definition of Object referred as Abstract class.
- The class which is not providing complete definition of Object.
- "abstract" is a pre-defined keyword allowed to define abstract classes.
- Abstract class also saved with .java extension only.
- .class file will be generated when abstract class has compiled.



- “abstract” modifier is used to define abstract classes.
- Abstract class is allowed to define
 - Abstract methods(methods don’t have body)
 - Concrete methods(methods having body)

```
abstract class AbstractClass
{
    // abstract class contains
    void concreteMethod()
    {
        // empty
    }
    abstract void abstractMethod();
}
```

- Abstract class cannot be instantiated directly using “new” keyword, because it was not fully defined.
- For example, we can release any Mobile into the market after 100% manufactured.

```

abstract class AbstractClass
{
    void concreteMethod()
    {
        // empty
    }
    abstract void abstractMethod();
}
class Access
{
    public static void main(String args[ ])
    {
        new AbstractClass(); // Error :
    }
}

```

Question: can we define main() method inside abstract class ?

Answer: As main() method is static, JVM no need to instantiate abstract class to invoke main method.

```

abstract class AbstractClass
{
    public static void main(String args[])
    {
        System.out.println("Abstract class main method");
        AbstractClass.method();
    }
}

```

- We can access static members of abstract class using class name.
- Hence we can define main method (Static) inside the abstract class.

Question: How can we access non-static members of abstract class ?

- Every abstract class must be extended.
- Abstract class specifications (abstract methods) must be implemented in extended class.
- By creating object for child class, we can access the functionality of Parent class(abstract)

```

abstract class Parent
{
    void f1()
    {
        System.out.println("Concrete method....");
    }
    abstract void f2();
}

```

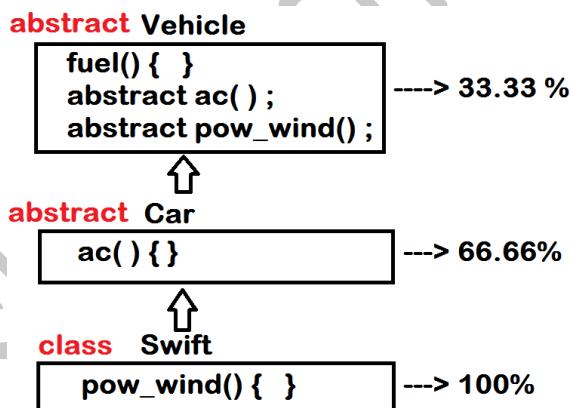
```

class Child extends Parent
{
    void f2()
    {
        System.out.println("abstract method implementation...");
    }
}
class Main
{
    public static void main(String[] args)
    {
        // Parent obj = new Parent(); // Error : we can't instantiate abstract class.....

        Child obj = new Child();
        obj.f1();
        obj.f2();
    }
}

```

Note: When any Child class of abstract class is unable to implement all the specifications of abstract class must be defined as abstract.



```

abstract class Vehicle
{
    void fuel()
    {
        System.out.println("fuel....");
    }
    abstract void ac();
    abstract void pow_wind();
}

abstract class Car extends Vehicle
{
    void ac()
    {
}

```

```

        System.out.println("ac.....");
    }
}
class Swift extends Car
{
    void pow_wind()
    {
        System.out.println("pow_windows....");
    }
}
class Main
{
    public static void main(String[] args)
    {
        Swift car = new Swift();
        car.fuel();
        car.ac();
        car.pow_wind();
    }
}

```

Question : Can we define constructor inside the abstract class ?

Answer : Yes allowed, when we instantiate child class, JVM implicitly creates abstract class object by invoking constructor using “super()” method....

```

abstract class Parent
{
    Parent() //concrete method
    {
        System.out.println("Abstract class instantiation");
    }
}
class Child extends Parent
{
    Child()
    {
        // super(); // implicit code...
        System.out.println("Child class instantiation");
    }
}
class Main
{
    public static void main(String[] args)
    {
        new Child();
    }
}

```

Question : Can we place non-static variables inside abstract class ?

- Yes allowed.
- Non-static variables belongs to Object
- We can't create object for abstract class directly but it is possible using Child class.
- Hence we provide initial values using super() method from Child class.

```
abstract class Parent
{
    int a;
    Parent(int x)
    {
        this.a = x;
    }
    abstract void details();
}

class Child extends Parent
{
    int a;
    Child(int x, int y)
    {
        super(x);
        this.a = y;
    }
    void details()
    {
        System.out.println("Parent's a : "+super.a);
        System.out.println("Child's a : "+this.a);
    }
}
class Main
{
    public static void main(String[] args)
    {
        int x, y ;
        java.util.Scanner sc = new java.util.Scanner(System.in);

        System.out.println("Enter Parent's a : ");
        x = sc.nextInt();

        System.out.println("Enter Child's a : ");
        y = sc.nextInt();

        Child obj = new Child(x, y);
        obj.details();
    }
}
```

Question: Can an abstract class be final?

Answer: No,

Abstract class must be extended where as
Final class cannot be extended.

```
final abstract AbstractClass{  
    abstract static void method();  
}
```

Note : A method cannot be abstract & final
A method cannot be abstract & static

```
class Test  
{  
    abstract final void f1(); // Error  
    abstract static void f2(); // Error  
}
```

Question: Why abstract method cannot be final?

Answer : Abstract method must be overridden in its sub class, but if method is final, we cannot override. Hence it is illegal combination.

Question: Why static method cannot be abstract?

- Common functionality of Object we define as static.
- Abstract methods are specific to particular object.
- Hence both static (common) and abstract (specific) are illegal combination.

Question: Can a final class extends Abstract class?

Answer: Yes,

but it has to implement all the specifications of abstract class
because another extension of final class is not allowed.

```
abstract class AbstractClass  
{  
    abstract void f1();  
    abstract void f2();  
}  
final class FinalClass extends AbstractClass  
{  
    void f1(){ }  
    void f2(){ }  
}
```

Question: Illegal combination of modifiers?

1. abstract & final

2. abstract & static

Interfaces

class: Complete representation of Object(100%)
(Only concrete methods)

abstract class: Partial representation of Object(.....%)
(Both Concrete & Abstract methods)

interface: Complete specification of Object (0%)
(Only abstract methods)

- “interface” is a pre-defined modifier is used to define set of specifications.
- “interface” definition allow only abstract methods.
- By default interface methods are “public & abstract”

```
interface Test
{
    void method()
    {
        // Error : interface methods can't have body
    }
}
```

```
interface Test
{
    void f1(); // by default public & abstract
    void f2();
}
```

- To check compiler added code to the class file we use javap command.

```
D:\>javac Test.java
D:\>javap Test                                         command used to check
Compiled from "Test.java"                           compiler written code
interface Test {
    public abstract void f1();
    public abstract void f2();
}
D:\>
```

compiler added code

- We can't instantiate interface.
- Interface is not fully defined.
- Interface doesn't allow constructors.
- We can't instantiate without constructor.

```
interface Test
{
    void f1();
    void f2();
}
class Main
{
    public static void main(String args[])
    {
        new Test(); // Error : can't instantiate
    }
}
```

Creating Object of	using "new" keyword	using child class
class	yes	yes
abstract class	no	yes
interface	no	no

Ques : Can we define main method inside interface ?

Ans : It is allowed since jdk 1.8

```
interface Test
{
    public static void main(String args[ ])
    {
        System.out.println("interface main method.....");
    }
}
```

- We can define static methods and can access directly using interface identity.
- Static members cannot be overridden because it is common functionality of Object.

- We cannot update static members in the hierarchy.
- If we want to update, we should modify in the top class of hierarchy.

interface Test

```
{
    static void m()
    {
        System.out.println("interface static method");
    }
    public static void main(String args[])
    {
        System.out.println("interface main method....");
        Test.m();
    }
}
```

- Only static method definitions allowed inside interface.
- Static methods cannot be overridden.
- Static methods are not specific.

Note : Any class can “**implements**” interface but not “**extends**”.

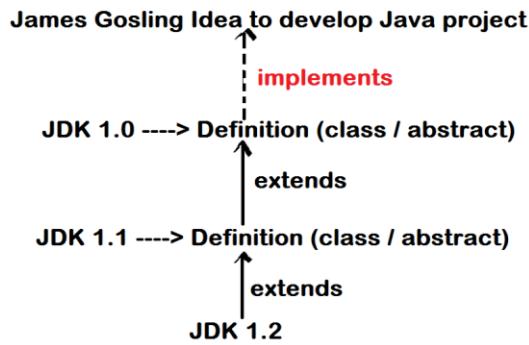
interface Test

```
{
    void f1(); // public method
    void f2();
}
class Main implements Test
{
    // void f1(){ } // Error : pls don't decrease privileges

    public void f1(){ } // must override as public
    public void f2(){ }
    public static void main(String args[])
    {
        Main obj = new Main();
        obj.f1();
        obj.f2();
    }
}
```

Ques : Why a class “**implements**” interface instead of “**extends**” ?

Ans : In case of class or abstract class, we can extend(update) existing functionality.
But in case of “**interface**” only specifications are available to implement.

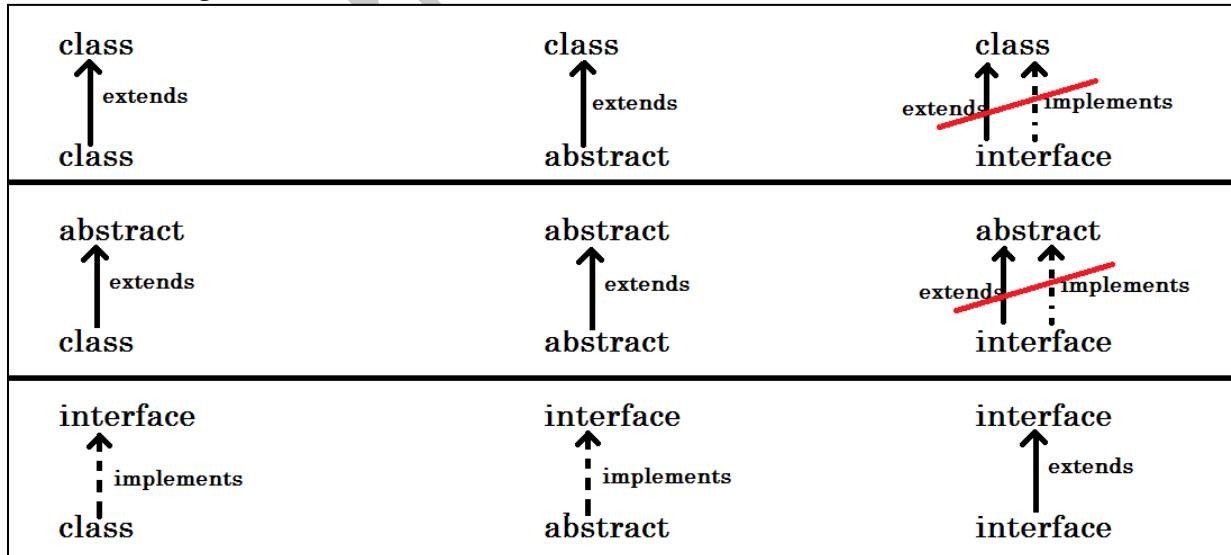


Note: If any class is unable to “implement” all the specifications of interface must be defined as “abstract class”.

```

interface Test
{
    void f1();
    void f2();
}
abstract class One implements Test
{
    public void f1(){ }
}
class Two extends One
{
    public void f2(){ }
}
  
```

Relations among class, abstract class & interface :



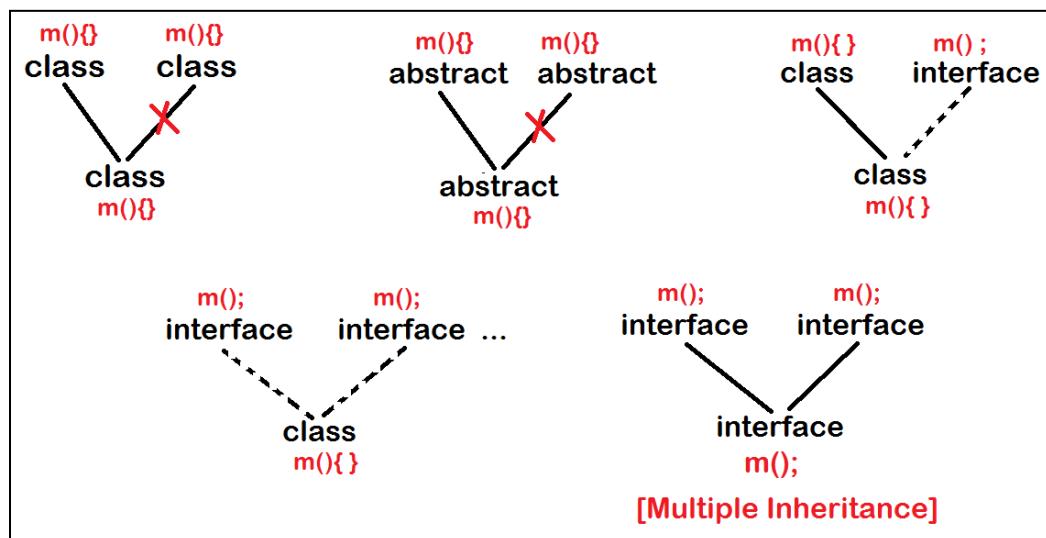
Note: We can implement “Multiple Inheritance” using interfaces in Java application.

One class can extends only one class.

One class can implement more than one interface.

One interface can extends many interfaces (Multiple inheritance).

Following diagram describes the relation between classes, abstract class and interfaces.



The following program describes all the above discussions are correct or not.

```
class A { /* logic */}  
class B { /* logic */}
```

```
abstract class C { /* logic */}  
abstract class D { /* logic */}
```

```
interface E { /* logic */}  
interface F { /* logic */}
```

```
class Diagram1 extends A, B  
{  
    // class extends only one class , Failed
```

```
}  
  
abstract class Diagram2 extends C,D  
{  
    // abstract class extends only one class, Failed
```

```
}  
  
class Diagram3 extends A implements E  
{  
    // allowed....
```

```

class Diagram4 implements E, F
{
    // allowed....
}

interface Diagram5 extends E, F
{
    // Multiple inheritance in Java
}

```

Observe the following Example :

```

interface In
{
    void f1(); //public abstract
    void f2();
}

class Main implements In
{
    public void f1(){ }
    public void f2(){ }
    public static void main(String[] args)
    {
        Main obj = new Main();
        obj.f1();
        obj.f2();
    }
}

```

Question : After implementation of interface, why we need to maintain interface in java application?
 Answer :

- Interface is a standard set of specifications.
- Every standard(interface) must be implemented with a Duplicate name(class)
- After their implementation with duplicate name (class name), we need set Standard (interface) name to that object.
- One standard object always tries to communicate with another standard object only in communication world.

```

interface Lenovo
{
    void processor();
    void motherBoard();
    void display();
}

class HarshaComputers implements Lenovo
{

```

```

public void processor(){}
public void motherBoard(){}
public void display(){}
public static void main(String[] args)
{
    Lenovo hc = new HarshaComputers();
    hc.processor();
    hc.display();
}
}

```

Note: If one object definition (AssembledComputer) based on multiple standards (Intel, AMD,LG), we cannot provide one standard name to the entire object.

```

interface Intel
{
    void processor();
}
interface AMD
{
    void motherBoard();
}
interface LG
{
    void display();
}

class AssembledComputer implements Intel, AMD , LG
{
    public void processor()
    {
        System.out.println("Intel's specification...");
    }
    public void motherBoard()
    {
        System.out.println("AMD's specification...");
    }
    public void display()
    {
        System.out.println("LG's specification...");
    }
    public static void main(String[] args)
    {
        AssembledComputer ac = new AssembledComputer();
        ac.processor();
        ac.motherBoard();
        ac.display();
    }
}

```

```
Intel i = new AssembledComputer();
i.processor();
i.display(); // Error : not specified by Intel.
}
}
```

Question: Why it is allowed to store Child object address into Parent reference variable?

Lenovo obj = new Computer();

Answer : Object casting concept.

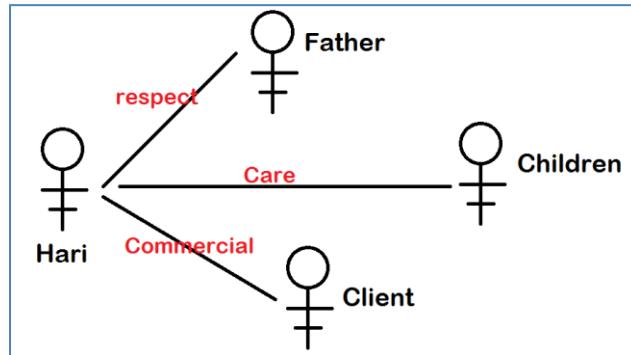
Naresh/Techologies

Polymorphism

Defining an Object(class) that shows different behavior(methods) with the same Identity.

Poly means "Many"

Morphed means "Forms"



Java supports 2 types of polymorphism:

1. Compile time polymorphism
2. Runtime polymorphism

Compile time polymorphism:

- Also called static binding.
- It is method overloading technique.
- Method overloading refers defining more than one method with the same name within the same class but with different signatures.
- It can be implemented in a single class.
- In static binding, compiler can understand which it has to bind on function call at compile time only. So at runtime, JVM no need to search for the function to be executed.

```
class Addition
{
    void add(int x, int y)
    {
        System.out.println("sum of 2 int's : "+(x+y));
    }
    void add(int x, int y, int z)
    {
        System.out.println("sum of 3 int's : "+(x+y+z));
    }
}
class StaticBinding
{
    public static void main(String[] args)
    {
        Addition obj = new Addition();
    }
}
```

```

        obj.add(10,20,30); // method call...
    }
}

```

Question: Can we overload Constructor?

Answer: Yes allowed. We have already seen in Constructor Chaining concept.

Question : Can we overload main method ?

Answer : Yes but JVM can access the main method only if it follows pre-defined prototype.

If we defined any main method explicitly, we must call from another location.

```

class MainOverLoad
{
    void main()
    {
        System.out.println("zero args...");
    }
    static void main(int x)
    {
        System.out.println("One args main.....");
    }
    public static void main(String[] args)
    {
        System.out.println("JVM main method....");
        MainOverLoad obj = new MainOverLoad();
        obj.main();
        obj.main(10);
    }
}

```

Runtime polymorphism:

- It is also called Dynamic binding.
- It is method overriding technique.
- Defining a method in Child class with the same name and same signature of its Parent class is called Method overriding.
- We can implement dynamic binding only in “Is-A” relation.
- More than one objects combined together to show Runtime polymorphism behavior.
- Allowed only in Super and Sub class.

Note: One advantage of Runtime Polymorphism is used to update Parent class functionality in Child class if it is not sufficient.

```

class SamsungGalaxy
{
    String camera(){
        return "12mp";
    }
}

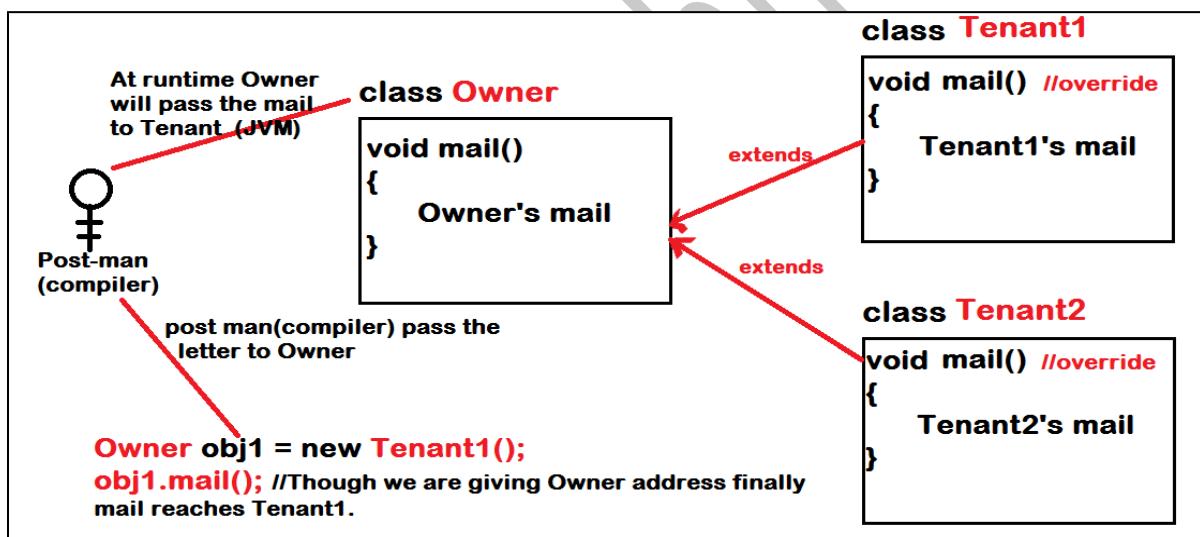
```

```

        String memory(){
            return "2gb"
        }
    }
    class SamsungEdge extends SamsungGalaxy
    {
        String camera(){ //overriding (updating functionality)
            return "16mp";
        }
        String memory(){
            return "4gb"
        }
    }
}

```

- The main advantage of Runtime polymorphism is “Accessing Child class object functionality using Parent object reference”.
- We know the importance of “C/O address” while addressing someone.
- The best example of runtime polymorphism as follows.



```

class Owner{
    void mail(){
        System.out.println("Owner received mail...");
    }
}
class Tenant1 extends Owner{
    void mail(){
        System.out.println("Tenant1 received mail...");
    }
}
class Tenant2 extends Owner{
    /*void mail()
    {
}

```

```

        System.out.println("Tenant2 received mail...");
    }*/  

}  
  

class RuntimeBinding {  

    public static void main(String[] args)  

    {  

        Owner obj1 = new Owner();  

        obj1.mail(); //Owner will receive as it was belongs Owner.  
  

        Owner obj2 = new Tenant1();  

        obj2.mail(); //Owner will pass mail to Tenant1 at Runtime.  
  

        Owner obj3 = new Tenant2();  

        obj3.mail(); //Owner will receive, as Tenant2 doesn't have functionality(we are writing  

in comments).  

    }  

}

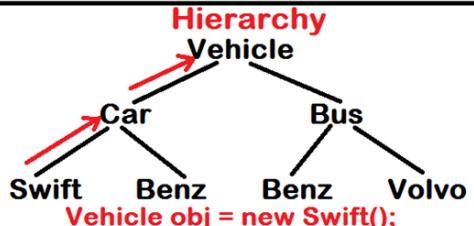
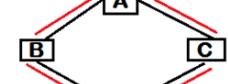
```

Question: why it is allowed to store sub class object into super class type variable directly?

Answer: Because of implicit object up casting.

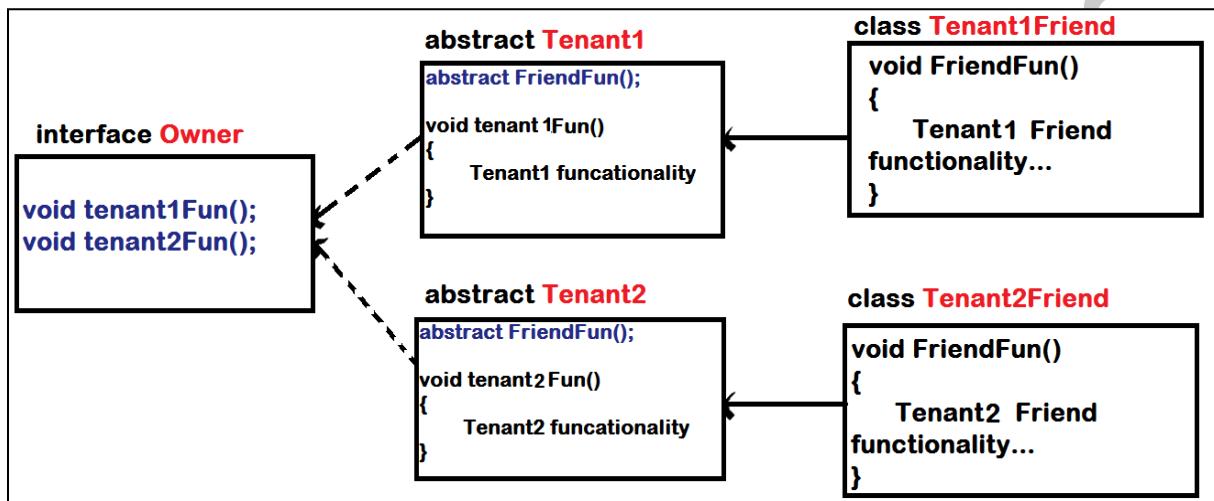
Question: why object up casting is implicit?

- Because multiple inheritance is not supported in java.
- No chance of deviation while up casting of object as there is only one parent for multiple child classes in the hierarchy.
- But down casting must be done explicitly.

Primitive cast	Object cast	Hierarchy
<pre>byte b ; b = 100 ; int i ; i = b ; //Implicit cast</pre>	<pre>Swift s ; s = new Swift(); Vehicle v ; v = s ; //implicit upcast</pre>	 Vehicle obj = new Swift(); While upcasting only 1 path is available. Hence no chance of deviation.
<pre>byte b1 ; b1 = i ; //Error b1 = (byte)i; //explicit</pre>	<pre>Swift s1 ; s1 = v ; //Error s1 = (Swift)v ; //Explicit downcast</pre>	 A obj = new D(); //2 paths are available from D-->A, chance of deviation.

- We can implement Runtime Polymorphism using Interfaces & Abstract classes.

- To connect runtime polymorphism Object, we can use simply specifications(abstract methods)
- As shown in the diagram,
 - We can access Tenant1 using Owner
 - We can't access Tenant1Friend functionality using Owner
 - Suppose if we pass a mail to Tenant1Friend using Owner, results Error.
 - If "Owner" pass that information to Tenant1 (Downcast), then accessing is possible.
 - Please go through diagram and implementation carefully.



```

interface Owner
{
    void tenantFun(); //public abstract
}
abstract class Tenant implements Owner
{
    public abstract void tenantFriendFun();
    public void tenantFun()
    {
        System.out.println("Accessing Tenant fun.....");
    }
}
class TenantFriend extends Tenant
{
    public void tenantFriendFun()
    {
        System.out.println("Accessing Tenant Friend fun....");
    }
}
class RuntimeBinding
{
    public static void main(String[] args)
    {
    }
}
  
```

```
Owner o = new TenantFriend();
o.tenantFun() ; //Using owner we can access Tenant
// o.tenantFriendFun(); //Error:we can't access Friend using Owner

Tenant t = (Tenant)o ; //Owner passing info to Tenant
t.tenantFriendFun();

}
```

Naresh Technologies

Exception Handling

During program Compilation and Execution there is a chance to get 3 kinds of Errors

1) Compile time errors:

- While writing programs using any High level languages, we need to follow set of language specification class syntactical rules.
- Violation of these rules produces, compile time errors.

Examples:

1. Cannot find symbol
2. "this" cannot be used in static context
3. Invalid method declaration.

2) Logical errors:

- Logical error is the compiler error and runtime error.
- It produces unexpected results when we run the application.

3) Runtime errors:

- Violation of JVM rules.
- As a programmer some of the errors we can analyze at the time of application development but we cannot handle.
- As a programmer we can write the logic but that executes only when problem has risen while application is running.

Exception : Runtime error occurs while application is running

Compiler error

```
class Program
{
    p.s.v.main()
    {
        static int a ;
    }
}
```

syntax rule : local
variable cannot
be static

Logical error

```
class Program
{
    p.s.v.main()
    {
        int a=5 , b=2 ;
        float c = a/b ;
        s.o.p(c);
    }
}
```

Excepted output : 2.5
But prints : 2.0

Runtime error

```
class Program
{
    p.s.v.main()
    {
        int a[] = new int[5];
        a[7] = 30 ;
    }
}
```

Array size is 5
Accessing 7th location
Runtime error : Exception

Exception Causes....

1. Abnormal termination of program
2. Informal information to End user
3. Improper shutdown of Resources.

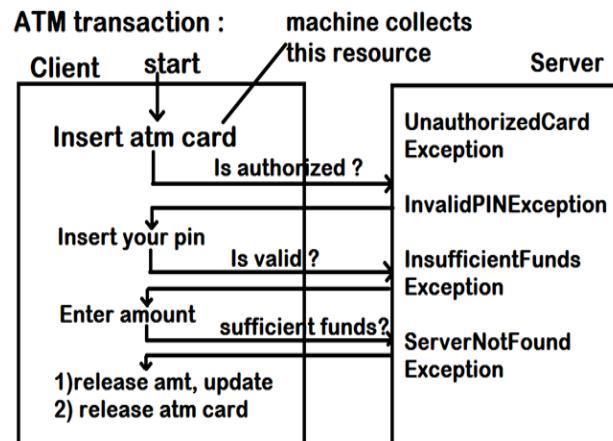
Note : As a programmer, we can analyze an error but we cannot handle at the time of application development.

for example,

- 1) inserting PIN number in atm transaction
- 2) playing videos in a computer
-

Hence we write Exception handling logic, that executes only if exception has raised @ runtime.

In a single ATM transaction, chance of getting many exceptions.

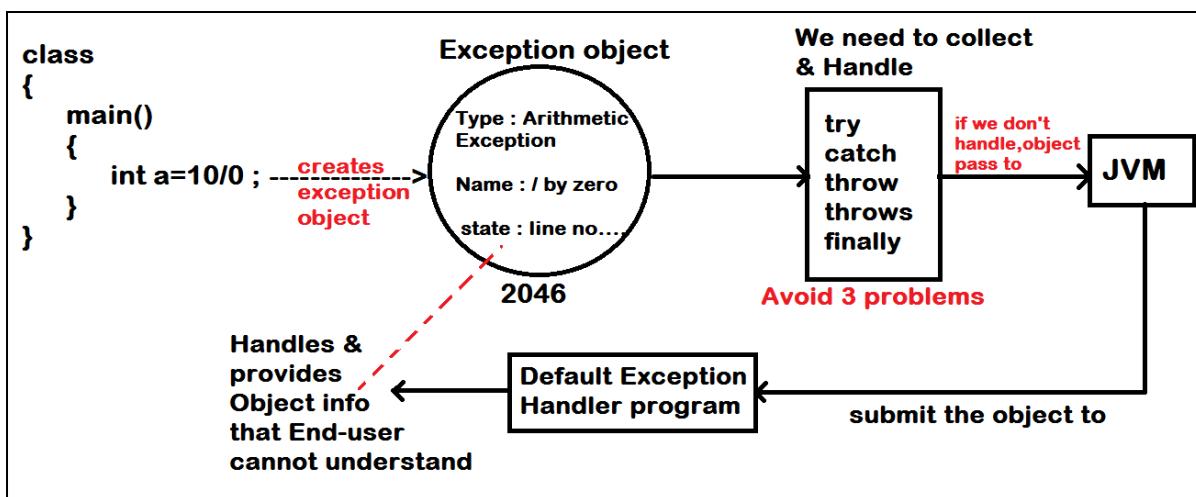


What happens when Exception has occurred?

- Exception occurs only either inside the block or method (because every statement must be placed inside the block or method only in java application).
 - When exception has raised, that block or method creates an exception object which contains the complete information of that error including....
1. Name of error
 2. Type of error
 3. State of error
- Once object has been created, it passes to the runtime system (JVM) then JVM handles that exception with the help of Default Exception Handler.
 - Hence mean while, we need to catch that exception and handle before the object reaches JVM, to avoid abnormal termination and continue with remaining statements execution in the application.

```

import java.util.Scanner;
class ExceptionDemo
{
    public static void main(String[] args)
    {
        int a,b,c;
        System.out.println("Enter two integers :");
        Scanner sc = new Scanner(System.in);
        a = sc.nextInt();
        b = sc.nextInt();
        c = a/b;
        System.out.println("result : "+c);
        System.out.println("continue.....");
    }
}
  
```



Chances to get exceptions:

- User has entered invalid arithmetic input causes “ArithmeticException”
- Accessing the data of array which is out of bounds causes “ArrayIndexOutOfBoundsException”.

```

class Test
{
    public static void main(String[] args)
    {
        int arr[ ] = new int[5];
        arr[7] = 30 ; //Exception :
    }
}

```

- Accessing the functionality of object using null pointer causes “NullPointerException”.

```

class Test
{
    static Test obj = null ;
    public static void main(String[] args)
    {
        Test.obj.check(); //Exception
    }
    void check()
    {
        ...logic
    }
}

```

- Invalid data conversion results “NumberFormatException”

```

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

```

```

        String s = "abcd";
        byte b = Byte.parseByte(s);
    }
}

```

- Downcasting the Object by providing another class name causes “ClassCastException”.
- Trying to access the data from the file which is not present in the file system “FileNotFoundException”
- Trying connect with database which is not ready “SQLException”
- Trying to retrieve IP address of system which is not connection the LAN “UnknownHostException”

Java API provides 5 keywords to handle exceptions which are

1. try
2. catch
3. finally
4. throws
5. throw

try:

- It is a pre-defined keyword
- Used to define a block of statements.
- Doubtful code that raises exception must be placed inside the try-block.
- if exception raised inside the block, it creates Exception Object and passes to "catch" block instead of JVM.

catch:

- It is a keyword
- used to define block which contains Exception handling code that raises in the corresponding try-block.

syntax:

```

try
{
    //doubtful code...
    // related info.....
}
catch(<Exception_type> <identifier>)
{
    //Handling code....
}

```

<Exception_type> specifies class name of Exception

<Identifier> specifies variable that hold exception object reference.

try - catch : try block creates exception object if any problem in logic. catch block collects that object and handles. Catch block will not execute if no exception in try-block

```
try
{
    1) doubtful code that raises exception
    2) Related info to exception
}
catch(Exception ref)
{
    Handling logic...
}
```

Exception class name Exception object reference variable

```
class Test
{
    main()
    {
        Read a, b ;
        try{
            int c = a/b ; //doubtful code
            print(c) ; //related info to exception
        }
        catch(ArithmeticException ae){
            print("b value not equal to 0");
        }
    }
}
```

```
class ExceptionDemo
{
    public static void main(String[] args)
    {
        int a,b,c;
        System.out.println("Enter two integers :");
        java.util.Scanner sc = new java.util.Scanner(System.in);
        a = sc.nextInt();
        b = sc.nextInt();
        try
        {
            c = a/b;
            System.out.println("result : "+c);//related information
        }
        catch (ArithmeticException ae)
        {
            System.out.println("Denominator should not be zero....");
        }
        System.out.println("end of program");
    }
}
```

Exception Hierarchy:

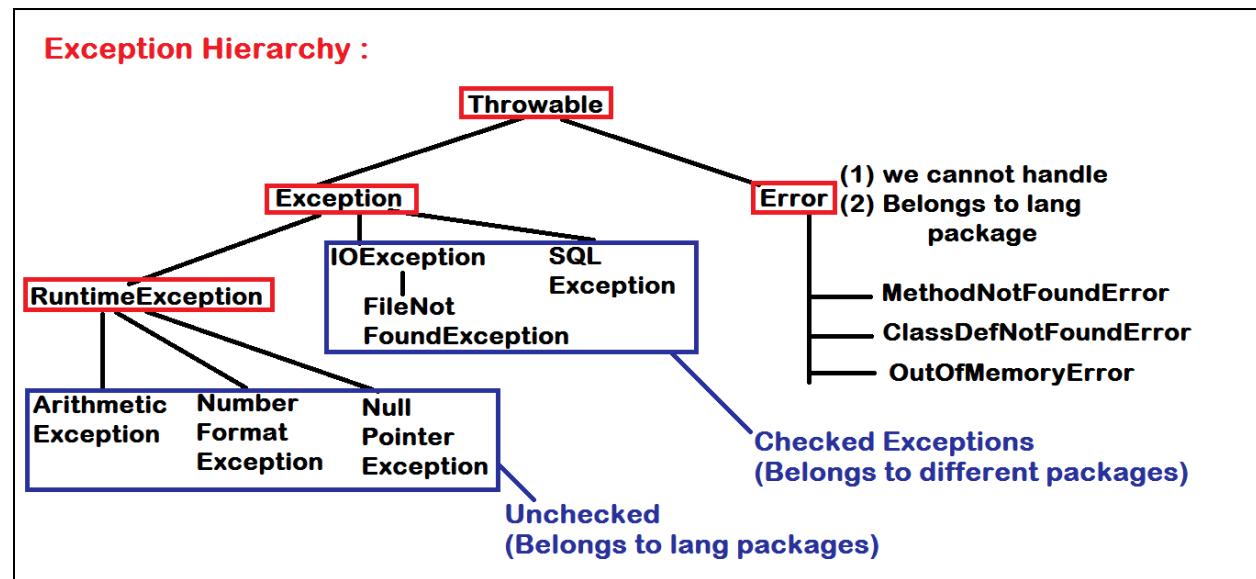
- java.lang.Throwable is the super class of all the Exceptions.
- It has 2 direct sub classes Exception & Error
- Errors can't be handled in java application.
- Sub classes of RuntimeException are called "Unchecked Exceptions" belongs to "java.lang" package.

Examples....

1. ArithmeticException
2. NumberFormatException

Sub classes of Exception are called "Checked Exceptions" belongs to corresponding package.

1. IOException (java.io)
2. SQLException(java.sql)



Note: It is also possible to handle an Exception by collecting the reference into its Super Exception type variable.

```
import java.util.Scanner;
class ExceptionDemo
{
    public static void main(String[] args)
    {
        try
        {
            int var = 10 / 0 ;
        }
        // catch (ArithmaticException ae){ }
        //catch (RuntimeException ae){ }
        //catch(Exception e){ }
        // catch(Throwable t){ }
        catch(java.io.IOException ie){ } // Error : try block never throws such type of
exception
    }
}
```

try with multiple catch blocks :

- We can define set of instructions in one try block.
- try with multiple catch blocks used to handle different exceptions occurred in different instructions of try block.

- We use try with multiple catch blocks when one task depends on more than one step of process and in each step there is a chance of getting exception.

Atm-transation :

- 1) Insert card --> if unauthorized --> Exception
- 2) Insert pin --> if invalid --> Exception
- 3) Enter amount --> if low balance --> Exception

Send a document through mail :

- 1) open file --> if not present --> Exception
- 2) write info --> if read only file --> Exception
- 3) send in network --> no connection --> Exception

syntax :

```
try
{
    ----- //chance to get Exception1
    .....
    ----- // chance to get Exception2
}
catch(Exception1 <var>)
{
    //Exception1 handling logic...
}
catch(Exception2 <var>)
{
    //Exception2 handling logic...
}
```

syntax :

```
try
{
    .....
    logic raises Exception1
    .....
    logic raises Exception2
}
catch(Exception1 e1)
{
    handling code...
}
catch(Exception2 e2)
{
    handling code
}
```

Ex :

```
try
{
    enter pin
    .....
    enter amt
    .....
}
catch(InvalidPINException e1)
{
    invalid pin, transaction failed
}
catch(LowBalanceException e2)
{
    low balance, transaction failed
}
```

1) Look at the example
2) If problem in first step,
after handling exception
we can't continue with
second step.

if pin is invalid, it
never ask to enter amount

3) Hence after handling
particular catch block.
control never back to
try block.

Note the followings...

- If Exception has risen in the try block, control terminates the execution of followed statements in try-block and executes the catch block.
- After execution of catch block, control will never back to try.
- If no exception in try block, no catch block will be executed.
- It is possible to handle one Exception at a time among multiple catch blocks.

```
/*
Task :
1) Read one argument from command line.
2) Convert into byte.

*/
class MultiCatch
{
    public static void main(String args[ ])
    {
        try
        {
            String input = args[0];
            System.out.println("Input value : "+input);

            byte output = Byte.parseByte(input);
            System.out.println("Converted value : "+output);

            System.out.println("Task completed successfully....");
        }
        catch (ArrayIndexOutOfBoundsException ae)
        {
            System.out.println("No input , task failed...");
        }
        catch(NumberFormatException ne)
        {
            System.out.println("Invalid input, task failed...");
        }
        System.out.println("End....");
    }
}
```

```

D:\>javac MultiCatch.java
D:\>java MultiCatch
No input , task failed...
End.....

D:\>java MultiCatch srinivas
Input value : srinivas
Invalid input, task failed...
End.....

D:\>java MultiCatch 13
Input value : 13
Converted value : 13
Task completed successfully.....
End.....

```

Note : We cannot handle same exception more than one time using one try block.

```

class Test
{
    main()
    {
        try{
        }
        catch(Exception e1){}
        catch(Exception e2){ } //Error :
    }
}

```

```

try{
    int a=10 , b=0 , c ;
    c = a/b ;
    int x=10 , y=0 , z ;
    z = x/y ;
}
catch(ArithmeticException e1){
    s.o.p("exception while a/b")
}
catch(ArithmeticException e2){ //Unreachable catch
    s.o.p("exception while x/y")
}

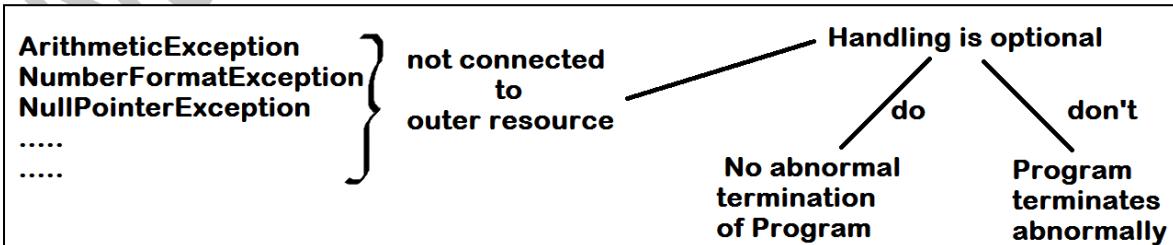
```

In both cases, control executes first catch block.

Unchecked v/s Checked Exceptions :

Unchecked:

- These types of exceptions occur in case of logical error.
- Handling of Unchecked exception takes care by programmer only.
- Compiler will not report though we don't handle unchecked exception in the application.
- But handling of unchecked exception avoids abnormal termination of program.



Checked Exception:

- While working with any outside resource (file, server, database, io devices.....), chance to get Checked Exception.
- After working with outside resource, it is mandatory to release (shutdown) the resource properly to avoid loss of data.
- Outside resource is not the property of Java application. Hence we must handle every checked exception in the application.
- If not, compiler reports error and class file will not be generated.

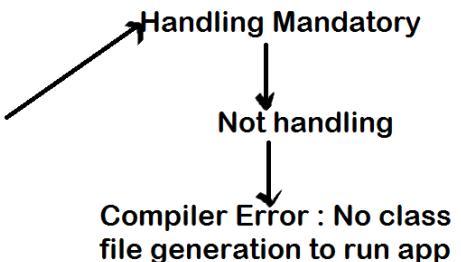
Checked Exceptions :

- 1) Not the sub class of RuntimeException.
- 2) Belongs to different packages.

Examples :

FNFE : Working with Files
IOE : I/O devices
SQLE : Database
ServletE : Servers

} Java App connected to outer resource



While using any method or constructor in the application, we need to consider the complete prototype of that method.

Following diagram describes an example with its prototype.

Prototype :

<access_modifier> <modifier> <modifier>
return_type Identity(arguments) throws **Exception**

Ex :

public static final float division(int , int) throws **ArithmeticException**

Unchecked
(Handling is optional)

Checked
(Handling is mandatory)

Unchecked : SubClass of RuntimeException

```
/*class FileInputStream
{
    public FileInputStream(String name) throws FileNotFoundException
```

```

        {
            opens a file in read mode....
        }
    }*/
import java.io.FileInputStream ;
import java.io.FileNotFoundException;
class CheckedException
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fis = new FileInputStream("g:/input.txt");
            System.out.println("File opened successfully...");
        }
        catch (FileNotFoundException fnfe)
        {
            System.out.println("file doesn't exist....");
        }
    }
}

```

- In the above application, we opened the File but we didn't close.
- Concept of CheckedException is releasing resource (File) properly.
- We can close any resource using finally-block.

finally:

- A block of instructions.
- Used to release a resource in Exception handling
- Only using in case Checked exceptions.
- finally block executes whether or not an Exception has raised in the try-block.
- finally block executes in case of abnormal termination of program also.

Note the followings...

1. try-block cannot be present without either catch-block or finally-block.
 - a. if try-catch is present, finally-block is optional
 - b. if try-finally is present, catch-block is optional
2. finally block gets execute in case of Abnormal termination of program also.
3. The order of 3 block as try-catch-finally.
4. No other statement is allowed in between try-catch-finally.

Tips :

- Try-block : Handling Exception.
- Catch block : Avoids abnormal termination.
- Finally block : Resource releasing code.

Case1 :

```
method(){  
    try{  
        } //only try-block is not allowed....  
    }  
}
```

Case2 :

```
method(){  
    try{  
        } //placing doubtful code  
    }  
    catch(){  
        //Handling UncheckedException  
        //No abnormal termination  
    }  
}
```

Case3 :

```
method(){  
    try{  
        } //doubtful code  
    }  
    finally{  
        //closing statements, working with CheckedException, Abnormal termination.  
    }  
}
```

Case 4 :

```
method(){  
    try{  
        } //doubtful code.....  
    }  
    catch(){  
        //Handling Checked Exception, No abnormal termination  
    }  
    finally{  
        //Releasing resources  
    }  
}
```

try block cannot be present without either catch or finally	if try-catch is present, finally is optional	if try-finally is present catch is optional	try-catch-finally must be placed in the following order	No other statement is allowed in between try -catch - finally
<pre>try { }</pre> 	<pre>try { } catch() { }</pre> <p style="text-align: center;">↓</p> <p>1) working with unchecked Exception 2) No abnormal termination</p>	<pre>try { } finally { }</pre> <p style="text-align: center;">↓</p> <p>1) working with checked Exception 2) abnormal termination</p>	<pre>try { } catch() { } finally { }</pre> <p>1) working with checked 2) no abnormal termination</p>	<pre>try { } s.o.p(); catch() { }</pre> 

The following program describes how finally-block executes in all the situations.

class Exceptions

```
{
    public static void main(String[] args)
    {
        try
        {
            String s = "100";
            byte b = Byte.parseByte(s);
            System.out.println("b value : "+b);
        }
        catch (NumberFormatException nfe)
        {
            System.out.println("Exception caught....");
        }
        finally
        {
            System.out.println("finally block....");
        }
    }
}
```

Following program explains how finally block executes in case of abnormal termination of program.

class Exceptions

```
{
    public static void main(String[] args)
    {
        try
        {
            String s = "150";

```

```

        byte b = Byte.parseByte(s);
        System.out.println("b value : "+b);
    }
    /*catch (NumberFormatException nfe)
    {
        System.out.println("Exception caught....");
    }*/
    finally
    {
        System.out.println("finally block....");
    }
}
}

```

Program explains how to use finally block to close a resource(file).

```

/*class FileInputStream
{
    public FileInputStream(String name) throws FileNotFoundException
    {
        opens a file in read mode....
    }
    public void close() throws IOException
    {
        Closes this file input stream
    }
}*/

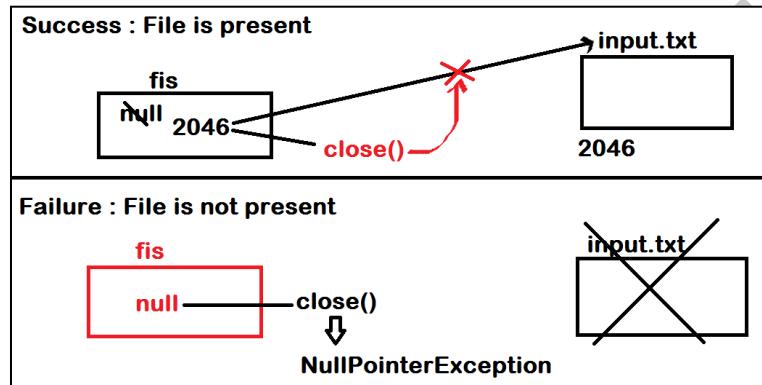
```

```

import java.io.FileInputStream ;
import java.io.FileNotFoundException;
import java.io.IOException ;
class CheckedException
{
    public static void main(String[] args)
    {
        FileInputStream fis = null ;
        try
        {
            fis = new FileInputStream("g:/input.txt");
            System.out.println("File opened successfully...");
        }
        catch (FileNotFoundException fnfe)
        {
            System.out.println("file doesn't exist....");
        }
        finally
        {
            try

```

```
        fis.close();
        System.out.println("File closed successfully....");
    }
    catch (IOException io)
    {
        System.out.println("IO error....");
    }
}
```



```
/*class FileInputStream
{
    public FileInputStream(String name) throws FileNotFoundException
    {
        opens a file in read mode....
    }
    public void close() throws IOException
    {
        Closes this file input stream
    }
}*/
```

```
import java.io.FileInputStream ;
import java.io.FileNotFoundException;
import java.io.IOException ;
class CheckedException
{
    public static void main(String[] args)
    {
        FileInputStream fis = null ;
        try
        {
            fis = new FileInputStream("g:/input1.txt");
            System.out.println("File opened successfully...");
```

```

        }
        catch (FileNotFoundException fnfe)
        {
            System.out.println("file doesn't exist....");
        }
        finally
        {
            try
            {
                fis.close();
                System.out.println("File closed successfully....");
            }
            catch (IOException io)
            {
                System.out.println("IO error....");
            }
            catch(NullPointerException npe)
            {
                System.out.println("Couldn't close file...");
            }
        }
    }
}

```

throws:

- If any method is unable to handle an exception object, it can pass to next level.
- It is recommended to handle every exception in any level of java application.
- If we don't handle an exception in any level, finally JVM handles using Default Exception handler.

```

import java.io.*;
class CheckedException
{
    public static void main(String[] args) throws FileNotFoundException, IOException
    {
        FileInputStream fis = new FileInputStream("g:/input.txt");
        System.out.println("File opened successfully...");

        fis.close();
        System.out.println("File closed successfully....");
    }
}

```

We can handle all the exceptions using “Exception” type as follows.

```

import java.io.*;
class CheckedException
{

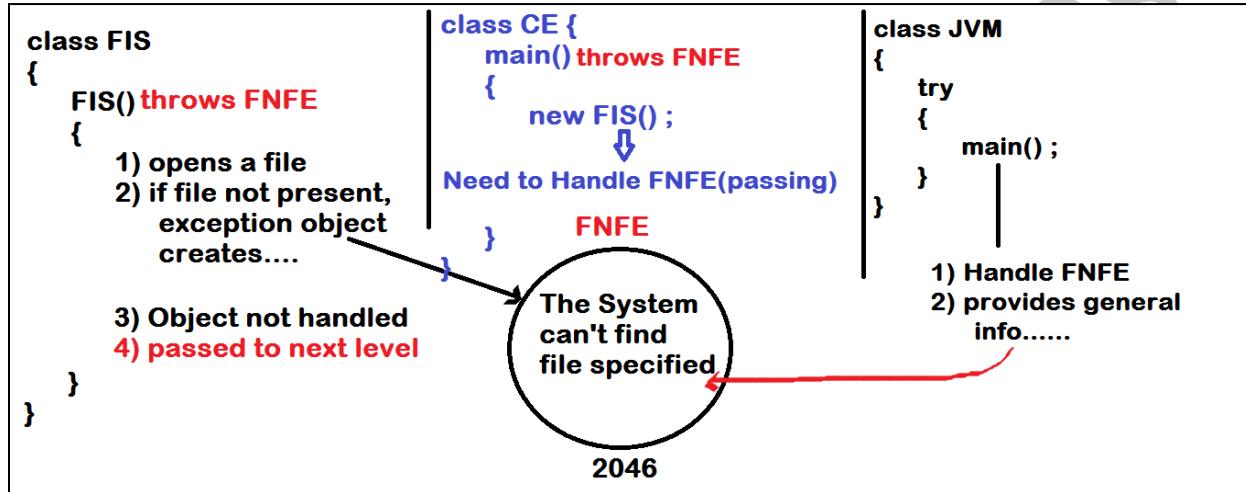
```

```

public static void main(String[] args) throws Exception
{
    FileInputStream fis = new FileInputStream("g:/input.txt");
    System.out.println("File opened successfully...");

    fis.close();
    System.out.println("File closed successfully....");
}
}

```



throw:

- Exception is a class.
- It is either pre-defined or user-defined.
- It is allowed to create user exceptions by extending functionality from pre-defined exception.
- User exception can be either Checked or Unchecked.
- **Most of the User exceptions are “checked”.**

```

class UserException extends RuntimeException
{
    User defined Unchecked exception
}

```

```

class UserException extends Exception
{
    User defined Checked Exception
}

```

- “throw” keyword is used to throw an exception object.
- In case of pre-defined exceptions, if exception has risen, JVM creates and throws.
- But in case of User exception, we need to create and throw.

```

class Test
{
    public static void main(String[] args)
    {
        int var = 10 / 0 ;
        /*
            1) creates ArithmeticException object
            2) place error message inside object
            3) throws that object.
        */
    }
}

```

In the above application, what is happening internally when exception rises, now we are doing explicitly.

Creating & Throwing pre-defined Unchecked exception object :

For example, take ArithmeticException class.

```

/*class ArithmeticException
{
    public ArithmeticException(String s)
    {
        Constructs an ArithmeticException with the specified detail message.
    }
}/*
class Test
{
    public static void main(String[] args)
    {
        ArithmeticException obj = new ArithmeticException("User_msg");
        throw obj;
    }
}

```

Another example:

```

class Test
{
    public static void main(String[] args)
    {
        NumberFormatException nfe = new NumberFormatException("Invalid data
conversion....");
        throw nfe;
    }
}

```

Creating & Throwing pre-defined Checked exception object :

```

/*class IOException
{
    public IOException(String s)
    {
        Constructs an IOException with the specified detail message.
    }
    public String getMessage()
    {
        Returns the detail message string of this throwable.
    }
}*/



import java.io.IOException;
class Test
{
    public static void main(String[] args)
    {
        try
        {
            IOException obj = new IOException("IO error");
            throw obj;
        }
        catch (IOException io)
        {
            System.out.println("Exception : "+io.getMessage());
        }
    }
}

```

Throwing pre-defined checked exception and using “throws” also.

```

import java.io.IOException;
class Test
{
    static void method() throws IOException
    {
        IOException obj = new IOException("IO error");
        throw obj;
    }
    public static void main(String[] args)
    {
        try
        {
            Test.method();
        }
        catch (IOException io)
        {
            System.out.println("Exception : "+io.getMessage());
        }
    }
}

```

```

        }
    }

Create User defined exception object and throw :
class MyException extends Exception
{
    String name ;
    MyException(String name)
    {
        this.name = name ;
    }

    String getErrorMessage()
    {
        return this.name ;
    }
}

class Test
{
    static void method() throws MyException
    {
        MyException obj = new MyException("Message....");
        throw obj ;
    }
}

class Main
{
    public static void main(String[] args)
    {
        try
        {
            Test.method();
        }
        catch (MyException me)
        {
            System.out.println("Exception : "+me.getErrorMessage());
        }
    }
}

```

Usage of Exception in General applications:

- In a bank application, when we perform withdraw operation, chance of getting different kinds of exception.
 - InvalidPINException
 - UnauthorizedCardException
 - **InsufficientFundsException**

```

import java.util.Scanner ;
class InsufficientFundsException extends Exception
{
    String name ;
    InsufficientFundsException(String name)
    {
        this.name = name ;
    }

    String getErrorMessage()
    {
        return this.name ;
    }
}

class Account
{
    private int balance ;
    Account(int balance)
    {
        this.balance = balance ;
    }
    public int getBalance()
    {
        return this.balance ;
    }
    public void withdraw(int amount) throws InsufficientFundsException
    {
        System.out.println("Trying to withdraw : "+amount);
        System.out.println("Avail balance : "+this.balance);
        if(amount <= this.balance)
        {
            this.balance = this.balance - amount ;
            System.out.println("Collect cash : "+amount);
        }
        else
        {
            InsufficientFundsException ife = new InsufficientFundsException("Low
balance");
            throw ife ;
        }
    }
}

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

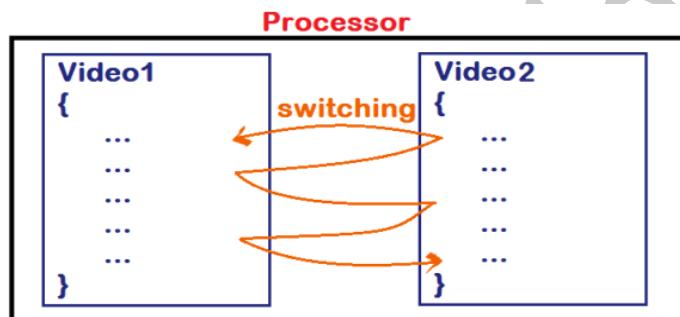
```

```
{  
    Scanner sc = new Scanner(System.in);  
    int amount ;  
  
    System.out.print("Enter initial amount : ");  
    amount = sc.nextInt();  
    Account acc = new Account(amount);  
    System.out.println("Balance in account : "+acc.getBalance());  
  
    System.out.print("Enter amount to withdraw : ");  
    amount = sc.nextInt();  
    try  
    {  
        acc.withdraw(amount);  
    }  
    catch (InsufficientFundsException ife)  
    {  
        System.out.println("Exception : "+ife.getErrorMessage());  
    }  
    System.out.println("Final Balance in account : "+acc.getBalance());  
}  
}
```

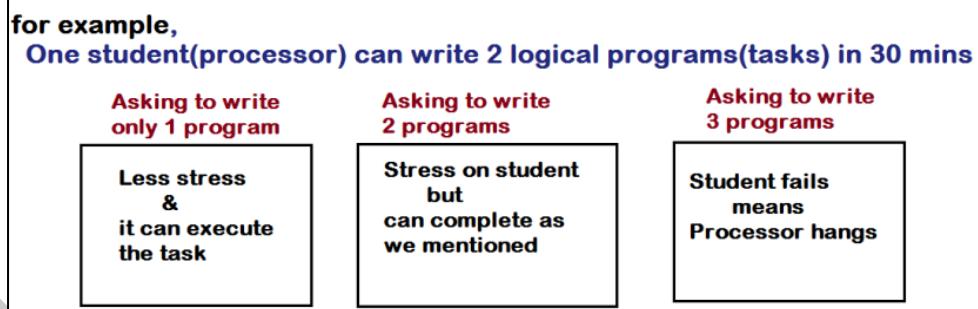
Multi threading

Multitasking:

- Performing more than one task simultaneously using single processor.
- Any processor can execute a single instruction at a time.
- When we execute more than one task, process space will be shared.
- While performing multi tasking, the control switches between the contexts and executes instructions according to Time specification.
- Multitasking is the concept of optimum utilization of CPU.
- Control unit uses two OS concepts to perform Multi tasking.
 - Time slicing
 - Context switching.

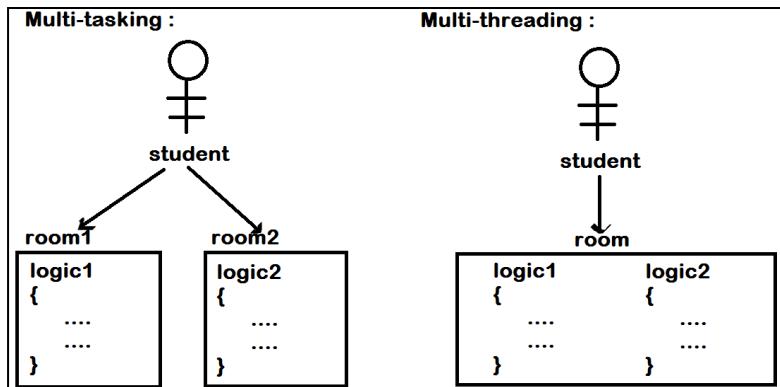


- If the processor is not capable of performing Multi tasking results abnormal termination (hang).



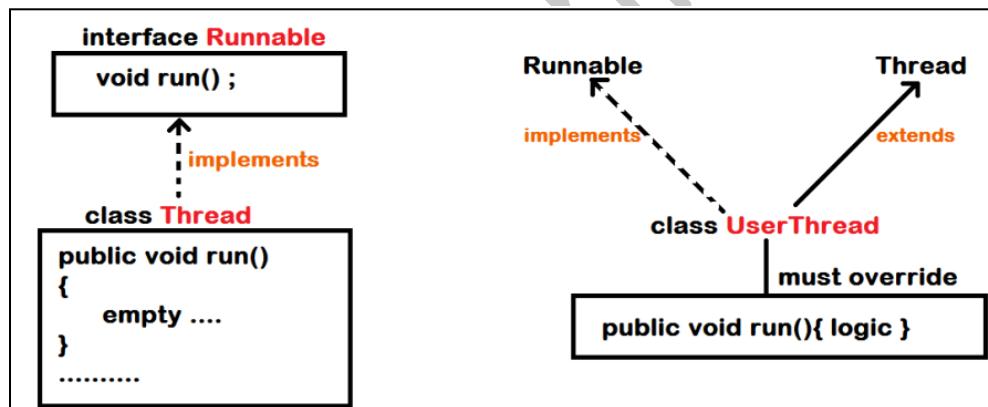
Multi threading:

- The only difference between Multi tasking and Multi threading is,
 - In Multi tasking, "n" process spaces required to perform "n" tasks.
 - In Multi threading, "n" threads execute in a single process space.
- Multi tasking takes care by Operating System. No programming effort is required.
- To implement Multi threading, we need to write program.
- Multi threading results, less stress on the Processor, because no need to switch between the contexts to perform Multi tasking.



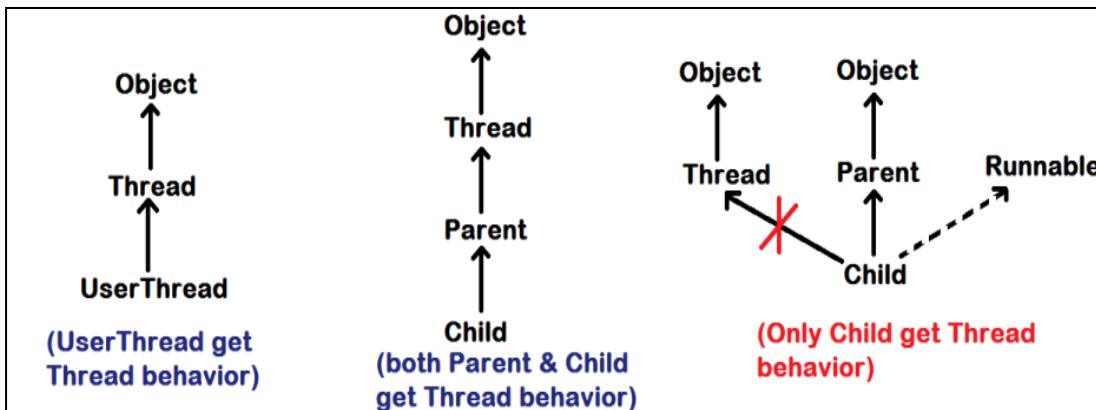
Creating a Thread:

- java.lang package contains pre-defined class and interface to create User threads.
- A thread can be created in 2 ways
 1. Extending from java.lang.Thread class
 2. Implementing from java.lang.Runnable interface
- Every Thread class must override "public void run()" method in its definition.
- run() method contains the logic that has to execute parallel.
- Pre-defined run method in the java.lang.Thread class is having empty definition.



When we use Runnable interface?

- We can use Thread class to apply thread behavior to any Object.
- Using Thread class, we can apply thread behavior to both Parent & Child classes in “Is-A” relation.
- **We must use Runnable interface only when we want to apply thread functionality only to Child class in “Is-A” relation.**



Life Cycle of Thread:

In the Life cycle process of Thread, Programmer & Scheduler will participate.

Creation: Programmer has created Thread class either by extending or by implementing...

Instantiation: Thread Object will be created to participate in the communication only when it is instantiated by the programmer.

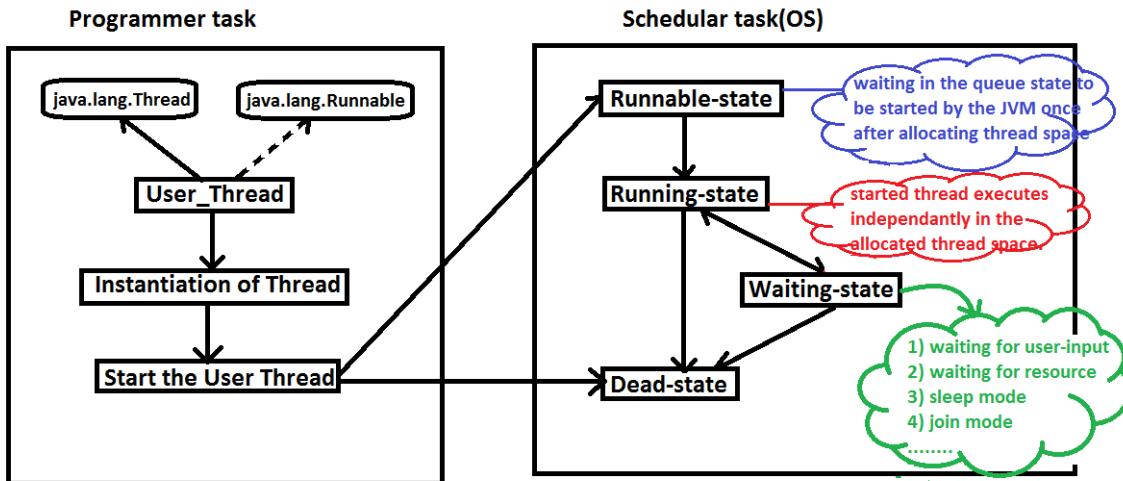
Start: As soon as Thread Object is ready, it has to be started by the Programmer.

Runnable state: The started thread by the programmer need to wait in the Queue until Scheduler allocates Thread space in the processor to be executed.

Running state: Logic execution state

Waiting state: Another Queue state into which thread enters from running state for different reasons....

Dead state: The last stage of Thread into which Thread enters either by completing all the thread instruction or in case of abnormal termination.

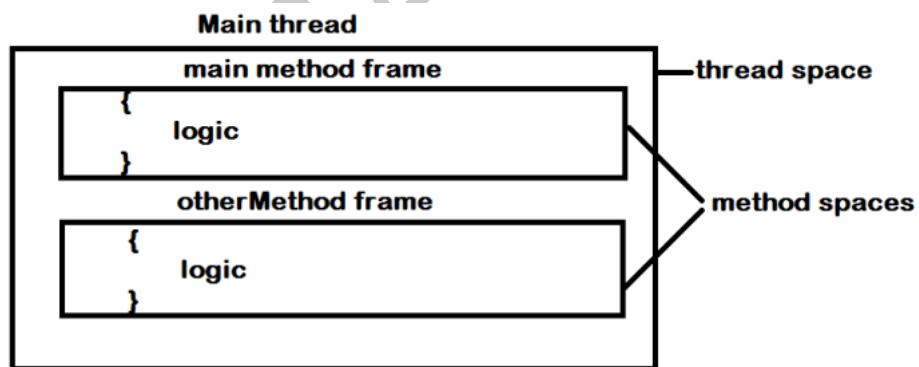


Types of Thread Applications:

1. Single Threaded Application
2. Multi Threaded Application

Single Threaded Application:

- When we invoke java application, JVM implicitly creates a thread is called "main thread".
- In single threaded application, execution starts at main thread and end at the same thread.
- One Thread space can have “n” number of method spaces to execute the thread logic.
- All the methods of single thread executes sequentially.



```

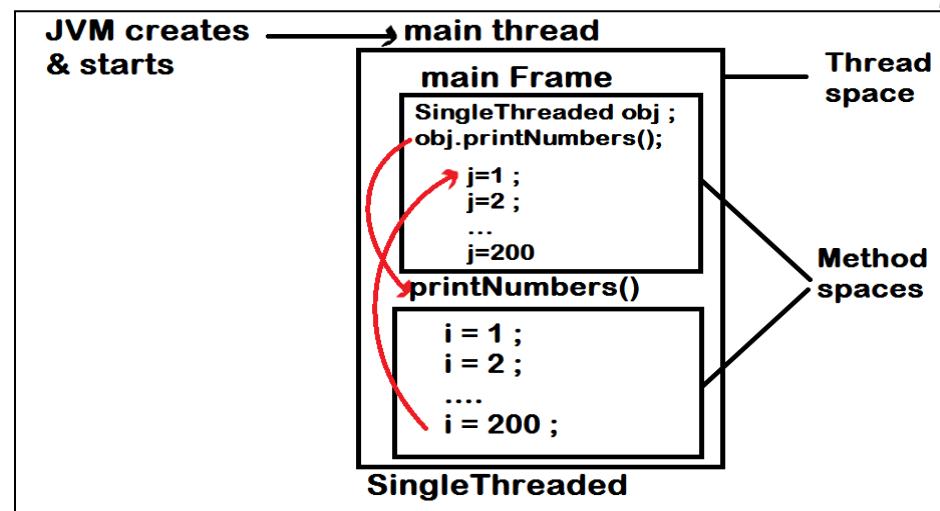
class SingleThreaded
{
    public static void main(String[] args)
    {
        SingleThreaded obj = new SingleThreaded();
        obj.printNumbers();

        for(int j=1 ; j<=200 ; j++)
        {
            System.out.print("j : "+j+"\t");
        }
    }
}
  
```

```

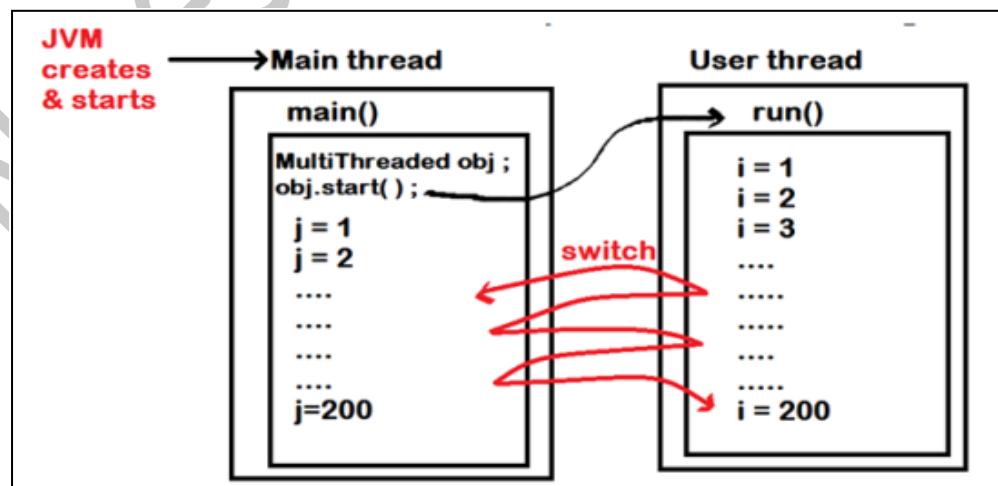
void printNumbers()
{
    for(int i=1; i<=200 ; i++)
    {
        System.out.print("i: "+i+"\t");
    }
}

```



Multi threaded Application:

- Creating a user thread from main thread referred as Multi threaded application.
- Multi threaded application execution starts at main thread only.
- In multi threaded application, main-thread can complete its execution before any child thread.
- Program execution completes, when all the running threads moved to dead state.
- All the threads in the application get independent memory allocation and execute parallel.



```

class MultiThreaded extends Thread
{
    public static void main(String[] args)
    {
        MultiThreaded obj = new MultiThreaded();
        obj.start();
        for(int j=1; j<=200 ; j++)
        {
            System.out.print("j : "+j+"\t");
        }
    }
    public void run()
    {
        for(int i=1; i<=200 ; i++)
        {
            System.out.print("i : "+i+"\t");
        }
    }
}

```

sleep() method:

```

/*class Thread
{
    public static void sleep(long millis) throws InterruptedException
    {
        Causes the currently executing thread to sleep (temporarily cease execution) for the
        specified number of milliseconds.
    }
}*/

```



The following application describes the use of sleep method and practically proves that main thread can complete before execution of any Child thread:

```

class MultiThreaded extends Thread
{
    public static void main(String[] args) throws InterruptedException
    By Srinivas (C/DS/Java trainer)

```

```

{
    MultiThreaded obj = new MultiThreaded();
    obj.start();
    for(int j=1; j<=10 ; j++)
    {
        System.out.println("j : "+j);
        Thread.sleep(500);
    }
    System.out.println("main exiting...");
}
public void run() // throws InterruptedException
{
    for(int i=1; i<=10 ; i++)
    {
        System.out.println("i : "+i);
        Try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException ie){ }
    }
    System.out.println("Child exiting...");
}
}

```

Question: why run() method cannot throws InterruptedException?

- Java syntactical rule describes, it is mandatory to override the method in the sub class how it has specified in its super-class.
- run() method is the overridden method, hence it is not possible to modify its prototype.

Pre-defined thread Identities:

- Every thread is having identity.
- JVM invokes the threads using their identities only.
- Every thread is initially identified by default name.
- Default names set by Thread class default constructor in Thread creation process.
- The following method is used to check identity of thread.
public final String getName()

```

/*class Thread{
    public Thread()
    {
        create thread and set default identity to that thread.
    }
}*/
class Identities extends Thread
{
    /*Identities()

```

```

{
    super();
} */
public static void main(String[] args)
{
    for(int i=1; i<=5 ; i++)
    {
        Identities obj = new Identities();
        System.out.println("Name : "+obj.getName());
    }
}

```

Setting Identity to the Thread explicitly:

It is possible to set the identity to thread in 2 ways

- 1) In the process of Construction of Thread using Thread class constructor.
- 2) In the process of Execution of Thread, we can rename the thread.

Using constructor:

```
public Thread(String name)
```

It sets the name to the current user thread.

```

class Threads extends Thread
{
    Threads(String name)
    {
        super(name);
    }
    public void run()
    {
        System.out.println("Name : "+this.getName());
    }
    public static void main(String[] args)
    {
        Threads t = new Threads("Child");
        t.start();
    }
}

```

Using setName() method : Used to rename the thread.

```
public final void setName(String name)
```

```

class Threads extends Thread
{
    Threads()
    {
        System.out.println("Default thread name : "+this.getName());
    }
}

```

```

    }
    public void run()
    {
        System.out.println("New Name : "+this.getName());
    }
    public static void main(String[] args)
    {
        Threads t = new Threads();
        t.setName("Child");
        t.start();
    }
}

```

join() method:

- A pre defined method used to implement inter thread communication.
- We can join the threads using this method.
- We use join() method in case of one thread execution should resume only when another thread moved to dead state.

Join : Thread execution stops until another thread execution completes.



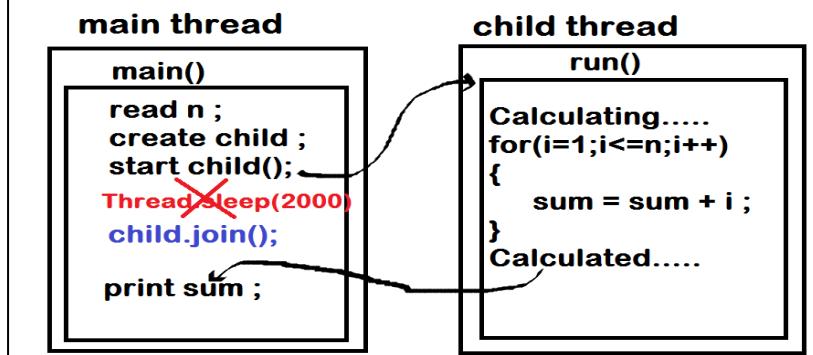
In the above diagram car(thread) should stops until train(thread) leaves completely

```

/*class Thread
{
    public void join() throws InterruptedException
    {
        Waits for "this" thread to die
    }
}*/

```

SUM OF FIRST "N" NUMBERS :



```
class JoinDemo extends Thread
{
    static int n , sum = 0 ;
    public static void main(String[] args) throws InterruptedException
    {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        System.out.println("/**SUM OF FIRST N NUBERS**/");
        System.out.print("Enter n value : ");
        JoinDemo.n = sc.nextInt();

        JoinDemo child = new JoinDemo();
        child.start();

        //Thread.sleep(2000);
        child.join(); //main thread wait until child moved to dead state.
        System.out.println("sum of first "+JoinDemo.n+ " numbers is :" +JoinDemo.sum);
    }
    public void run()
    {
        System.out.println("Calculation starts... ");
        for(int i=1 ; i<=JoinDemo.n ; i++)
        {
            JoinDemo.sum = JoinDemo.sum + i;
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException ie){}
        }
        System.out.println("Calculation end");
    }
}
```

Execution time of Single threaded application:

By Srinivas (C/DS/Java trainer)

Page 194

```

/*class System
{
    public static long currentTimeMillis()
        Returns the current time in milliseconds.
}*/

class PrintNumbers
{
    void print1to50() throws InterruptedException
    {
        for(int i=1 ; i<=50 ; i++)
        {
            System.out.print(i+"\t");
            Thread.sleep(100);
        }
    }

    void print50to1() throws InterruptedException
    {
        for(int i=50 ; i>=1 ; i--)
        {
            System.out.print(i+"\t");
            Thread.sleep(100);
        }
    }
}

class TimeCheck
{
    public static void main(String[] args) throws InterruptedException
    {
        PrintNumbers pn = new PrintNumbers();
        long start = System.currentTimeMillis();
        pn.print1to50();
        pn.print50to1();
        long end = System.currentTimeMillis();
        System.out.println("Time consumed : "+(end-start)/1000+" seconds");
    }
}

```

Execution time of Multi threaded application:

```

class PrintNumbers
{
    void print1to50() throws InterruptedException
    {
        for(int i=1 ; i<=50 ; i++)
        {
            System.out.print(i+"\t");
            Thread.sleep(100);
        }
    }
}

```

```

        }
    }
    void print50to1() throws InterruptedException
    {
        for(int i=50 ; i>=1 ; i--)
        {
            System.out.print(i+"\t");
            Thread.sleep(100);
        }
    }
}
class TimeCheck extends Thread
{
    static PrintNumbers pn = new PrintNumbers();
    public void run()
    {
        try
        {
            TimeCheck.pn.print1to50();
        }
        catch (InterruptedException ie){}
    }
    public static void main(String[] args) throws InterruptedException
    {
        TimeCheck t = new TimeCheck();
        long start = System.currentTimeMillis();
        t.start();
        TimeCheck.pn.print50to1();
        t.join(); //first child should move to dead state, then only we can take the end time
        long end = System.currentTimeMillis();
        System.out.println("Time consumed : "+(end-start)/1000+" seconds");
    }
}

```

Question: Why can't we call run() method directly ?

- It is allowed to call run() method explicitly instead of start() method.
- If we invoke run() method directly, the logic executes only in existing thread.
- Pre-defined start() method is having logic to allocate the separate thread space and executes run() method logic in the allocated space.
- As a programmer we can't allocate the thread space explicitly, hence we depends on start() method to create threads in application.

Question: Can we call start() method more than one time on the same thread Object at the same time?

Answer: not allowed, results IllegalThreadStateException. Violation rule of Synchronization.

```
class Threads extends Thread
```

```

{
    public void run()
    {
        for(int i=1; i<=5 ; i++)
        {
            System.out.println("run is executing.....");
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e){ }
        }
    }
    public static void main(String[] args)
    {
        Threads t = new Threads();
        t.run(); //sequential execution
        t.start(); //executes independantly
        t.start(); //if causes abnormal termination of current Thread(main)
        System.out.println("end of main....");
    }
}

```

Priority of a Thread:

- Every thread can have the priority by which Operating System schedules the thread when it is in waiting (queue) state.
- Threads having higher priority will be processed first.
- Every thread is having default priority.
- +66java.lang.Thread class contains Fields(pre-defined variables) to describe priorities of Thread.

```

/*class Thread{
    public static final int MAX_PRIORITY
        The maximum priority that a thread can have.
    public static final int NORM_PRIORITY
    public static final int MIN_PRIORITY
}
*/
class ThreadPriorities
{
    public static void main(String[] args)
    {
        System.out.println("min priority : "+Thread.MIN_PRIORITY);
        System.out.println("default priority : "+Thread.NORM_PRIORITY);
        System.out.println("max priority : "+Thread.MAX_PRIORITY);
    }
}

```

Finding the information of current Thread:

```
/*
    public static Thread currentThread()
        Returns a reference to the currently executing thread object.
*/
class CurrentThreadInfo extends Thread
{
    public void run()
    {
        System.out.println("Child name : "+this.getName());
        System.out.println("Child priority : "+this.getPriority());
    }
    public static void main(String[] args)
    {
        CurrentThreadInfo t = new CurrentThreadInfo();
        t.start();

        Thread m = Thread.currentThread();

        System.out.println("Parent name : "+m.getName());
        System.out.println("Parent priority : "+m.getPriority());
    }
}
```

Types of Threads:

Java application is allowed to define 2 types of Threads.

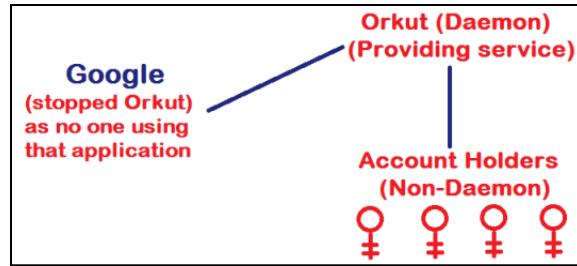
- 1) Non-Daemon threads
- 2) Daemon threads.

Non-Daemon threads:

- Non-daemon threads execute front end logic of application.
- Every thread is having Non-Daemon behavior as soon as it has created.

Daemon-threads:

- Thread executes behind the application.
- It is called service thread.
- It executes back ground logic of application.
- It is possible to change the behavior of Non-Daemon to Daemon.
public final void setDaemon(boolean on)
- The JVM implicitly stops all the daemon threads execution if all the Non-Daemon threads moved to dead state.
- It is possible to change the behavior of Thread into Daemon either in the process of Thread construction or before start.



```

class Threads implements Runnable
{
    static Thread t;
    Threads(String name)
    {
        Threads.t = new Thread(this , name);
        Threads.t.setDaemon(true); //Child thread behaves like Daemon thread....
    }
    public void run()
    {
        for(int i=1 ; i<=10 ; i
        {
            System.out.println(Threads.t.getName()+" : "+i);
            Try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException ie)
            {
                System.out.println("Exception caught...");
            }
        }
    }
    public static void main(String[] args) throws InterruptedException
    {
        Threads runnable = new Threads("Daemon");
        Threads.t.start();
        Thread m = Thread.currentThread();
        m.setName("Non-Daemon");
        for(int i=1 ; i<=10 ; i++)
        {
            System.out.println(m.getName()+" : "+i);
            Thread.sleep(500);
        }
    }
}

```

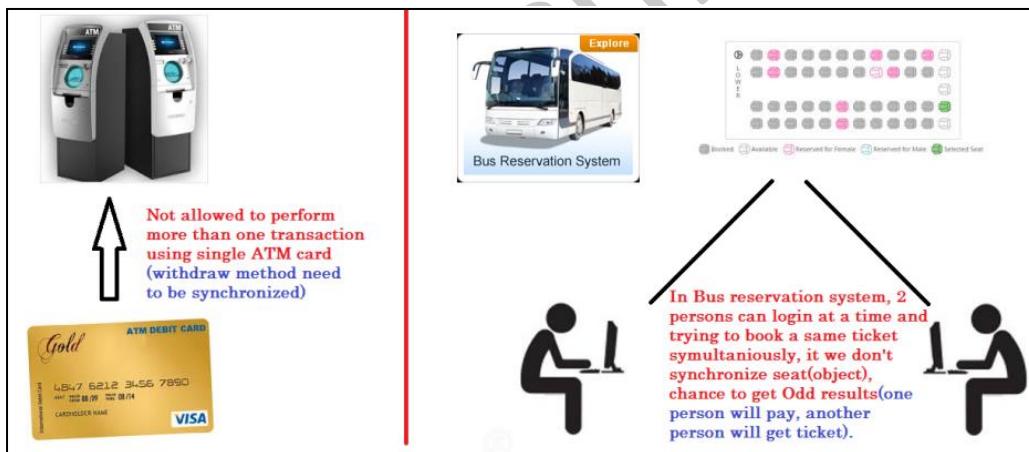
- It is not possible to change the behavior of Thread once it has been started(in running state).

- Hence main-thread always behaves like Non-Daemon, because it starts by the JVM implicitly.

```
class Threads extends Thread
{
    public void run()
    {
        this.setDaemon(true); //Thread behaviour can't be changed while it is running....
    }
    public static void main(String[] args)
    {
        new Threads().start();
    }
}
```

Thread synchronization:

- The concept of synchronization is, when multiple threads are trying to access a single resource, only one thread is allowed and all the remaining threads moved to queue (waiting) state.
- Complete class is not possible to synchronize in the java application.
- A method or block can be synchronized.



```
class InsufficientFundsException extends Exception
{
    String name;
    InsufficientFundsException(String name)
    {
        this.name = name;
    }
    String getErrorMessage(){
        return this.name;
    }
}
```

```

class Account{
    private int balance;
    Account(int balance)
    {
        this.balance = balance ;
    }
    public int getBalance()
    {
        return this.balance;
    }
    synchronized void withdraw(int amount) throws InsufficientFundsException
    {
        System.out.println("Trying to withdraw : "+amount);
        System.out.println("Balance in account : "+this.balance);

        if(amount <= this.balance)
        {
            System.out.println("Collect the cash : "+amount);
            this.balance = this.balance - amount ;
        }
        else
        {
            throw new InsufficientFundsException("No funds.....");
        }
    }
}
class AccountThread extends Thread
{
    Account atm ;
    int amt ;
    AccountThread(Account atm , int amt)
    {
        this.atm = atm ;
        this.amt = amt ;
    }
    public void run()
    {
        try
        {
            atm.withdraw(amt);
        }
        catch (InsufficientFundsException ie)
        {
            System.out.println("Error : "+ie.getMessage());
        }
    }
}
class ATMAApplication

```

By Srinivas (C/DS/Java trainer)

```

{
    public static void main(String[] args) throws InterruptedException
    {
        Account atm = new Account(6000);
        System.out.println("Initial amount in the account : "+atm.getBalance());
        AccountThread t1 = new AccountThread(atm, 4000);
        AccountThread t2 = new AccountThread(atm, 3000);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Final amount in the account : "+atm.getBalance());
    }
}

```

Block synchronization :

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- Generally we synchronize methods as shown in the above discussion.
- Block synchronization is used to synchronize a set of lines of code in a method.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

```

class PrintNumers
{
    void print(int n)
    {
        for(int i=1; i<=5 ; i++)
        {
            System.out.println("Parallel : "+i);
            try
            {
                Thread.sleep(500);
            }
            catch (Exception e){ }

        synchronized(this)
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("Sequential of "+n+" multiples : "+n*i);
                try
                {
                    Thread.sleep(400);
                }
                catch(Exception e)
                {

```

```

        System.out.println(e);
    }
}
}

class MyThread1 extends Thread
{
    PrintNumers pn;
    MyThread1(PrintNumers pn)
    {
        this.pn = pn;
    }
    public void run()
    {
        this.pn.print(5);
    }
}
class MyThread2 extends Thread
{
    PrintNumers pn;
    MyThread2(PrintNumers pn)
    {
        this.pn = pn;
    }
    public void run()
    {
        this.pn.print(100);
    }
}
class SynchronizedBlock
{
    public static void main(String args[])
    {
        PrintNumers obj = new PrintNumers();
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}

```

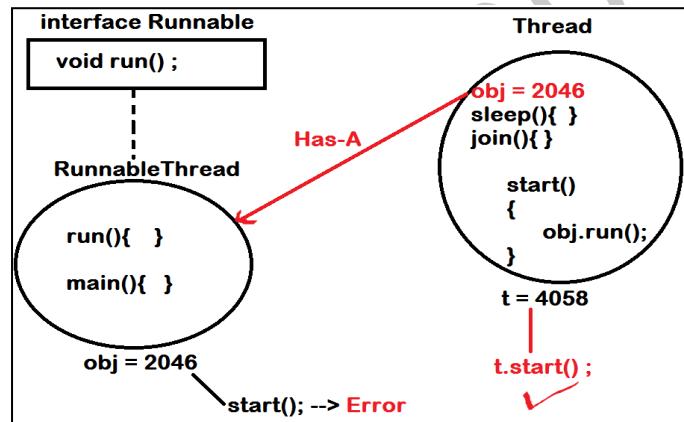
Creating Thread using Runnable interface:

- When a class is implementing Runnable interface, it has to implement all the specifications of Runnable

- public void run() is the only specification in Runnable interface....
- Runnable Object cannot be started directly. It is not the Thread.
- Hence, it is mandatory to create Thread Object using Runnable Object.

```
public Thread(Runnable target)
Allocates a new Thread object.
```

```
class Threads implements Runnable {
    public void run(){
        System.out.println("Child is created.....");
    }
    public static void main(String[] args){
        Threads runnable = new Threads(); //Runnable Object
        //runnable.start(); //CE : cannot start....
        Thread t = new Thread(runnable);
        t.start();
    }
}
```



Program to set the Identity of Thread that has created using Runnable.....

```
public Thread(Runnable target , String name)
```

```
class Threads implements Runnable
{
    static Thread t;
    Threads(String name)
    {
        Threads.t = new Thread(this , name);
        Threads.t.start();
    }
    public void run()
    {
        System.out.println("Name of Thread : "+Threads.t.getName());
    }
    public static void main(String[] args)
    {
    }
```

```

        Threads runnable = new Threads("Child");
    }
}

```

Has-A relation:

- One Object holding the reference of another Object.
- We can implement Has-A relation using Non-static variables because only non-static variables get memory allocation inside the object.
- As we shown in the diagram.... Human(Object) is holding the Reference(Controller) of another Object(Helicopter) , hence Human can access the complete functionality of Helicopter.



```

class Helicopter
{
    void accelerate()
    {
        System.out.println("Speed increasing");
    }
    void moveLeft()
    {
        System.out.println("Left turn....");
    }
    void moveRight()
    {
        System.out.println("Right turn....");
    }
    void moveForward()
    {
        System.out.println("Moving forward...");
    }
    void moveBackward(){}
}

```

```

class Human
{

```

By Srinivas (C/DS/Java trainer)

```
Helicopter heli ; //Human "Has-a" Helicopter
Human(Helicopter heli)
{
    this.heli = heli ;
}
void controlHelicopter()
{
    this.heli.accelerate();
    this.heli.moveLeft();
    this.heli.moveRight();
}
}

class MainClass
{
    public static void main(String args[ ])
    {
        Helicopter heli = new Helicopter();
        Human h = new Human(heli);
        h.controlHelicopter();
    }
}
```

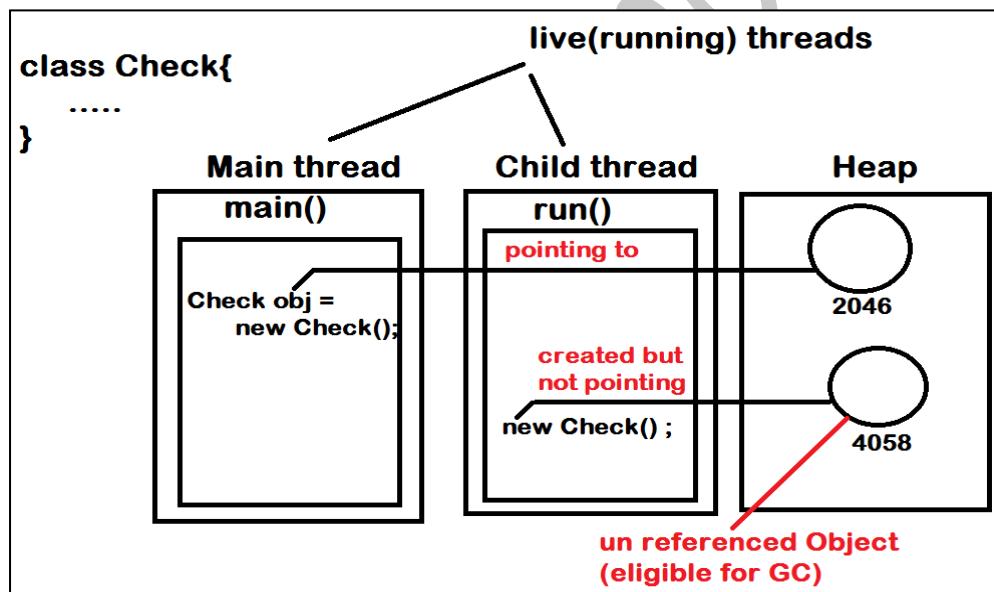
NaresII

Garbage Collection

- Objects will be created in Heap memory.
- Garbage Collection is a mechanism of re-acquiring the Heap space by destroying the Objects which are eligible for Garbage Collection.
- In other languages such as C or C++, dynamically allocated memory must be released explicitly
 - in C ---> *using free() function*
 - in C++ ---> *either delete or destructor*
- Java Objects deletes implicitly.

Question: When an Object is eligible for GC?

Answer: When multiple threads are running in the application, any thread can create object in the Heap area. When no live thread is referencing an Object in the Heap area is said to be Garbage Collection.



Garbage Collector:

- It is a Daemon thread.
- Daemon threads always run behind the application and provide service to the Non-Daemon thread.
- Every child thread is by default Non-Daemon when it has created including main thread.
- It is possible to change the behavior of Non-Daemon thread to Daemon thread at the time of thread creation only.
- Once the thread has been start, it is impossible to change its behavior.

Types of Threads:

Java application is allowed to define 2 types of Threads.

By Srinivas (C/DS/Java trainer)

Page 207

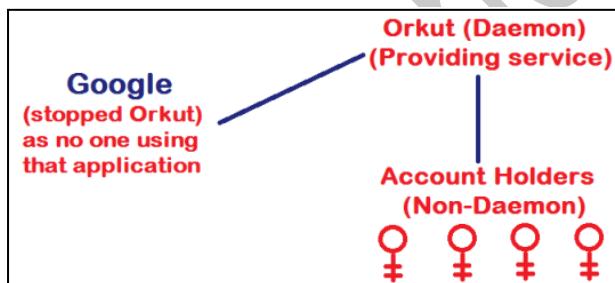
- 1) Non-Daemon threads
- 2) Daemon threads.

Non-Daemon threads:

- Non-daemon threads execute front-end logic of application.
- For example
 - Reading input
 - Displaying output
- Every thread is having Non-Daemon behavior as soon as it has created.

Daemon-threads:

- Thread executes behind the application.
 - It is called service thread.
 - It executes back ground logic of application.
 - It is possible to change the behavior of Non-Daemon to Daemon.
- public final void setDaemon(boolean on)**
- The JVM implicitly stops all the daemon threads execution if all the Non-Daemon threads moved to dead state.



This program explains JVM stops Daemon thread when Non-daemon threads moved to dead state:

```

class Demo extends Thread
{
    public void run()
    {
        for(int i=1; i<=10 ; i++)
        {
            System.out.println("Daemon : "+i);
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e){}
        }
        System.out.println("Daemon exiting...");
    }
}
  
```

```

public static void main(String[] args)
{
    Demo child = new Demo();
    child.setDaemon(true); // Converts Non-daemon behavior to Daemon....
    child.start();

    for(int i=1; i<=10 ; i++)
    {
        System.out.println("Non-Daemon : "+i);
        try
        {
            Thread.sleep(500);
        }
        catch (Exception e){ }

    }
    System.out.println("Non-Daemon exiting...");
}
}

```

- It is possible to change the behavior of Thread into Daemon either in the process of Thread construction or before start.
- If we try to convert while thread is running, results IllegalThreadStateException

```

class Demo extends Thread
{
    public static void main(String[] args)
    {
        Demo child = new Demo();
        child.start();
        child.setDaemon(true); // Exception :
    }
}

```

Question : Can we change the behavior of main thread to Daemon?

Ans : Not possible, we cannot change the behavior of thread to Daemon while it is running.

Main-thread implicitly starts by the JVM to execute application.

What happens when an Object is eligible for GC?

- JVM creates and starts GC thread.
- GC thread searches for un-referenced Object in the heap area.
- If found, it invokes **finalize()** method on that Object before destruction.
- **finalize()** method is working like finally block to place re-source releasing code of a particular Object.
- GC thread has to shutdown resources which are connected to object before destruction.
- **finalize()** method is available in **java.lang.Object** class as follows

protected void finalize()

```

    {
        //finalizing code....
    }

```

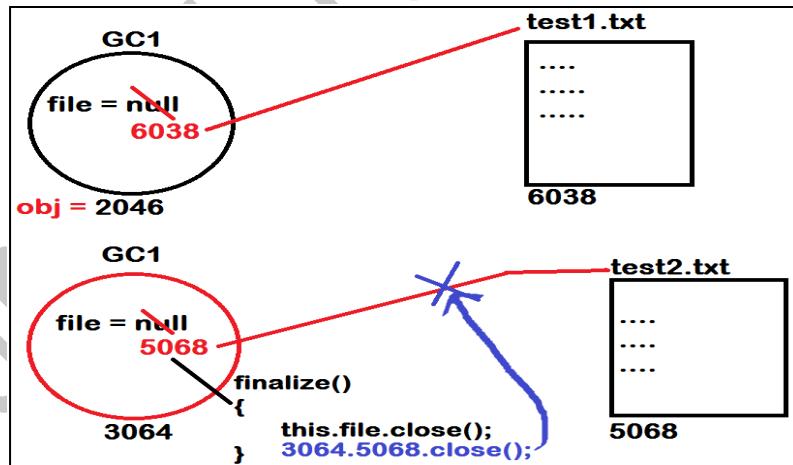
- finalize method working like destructor in C++.

The following program describes how objects are garbage collected using finalize() method.
Remember this program is used to understand the concept. It doesn't compile and execute.

```

class GC1
{
    File file; //non-static variable must be initialized using constructor.
    GC1(File file)
    {
        this.file = file;
    }
    public static void main(String[] args)
    {
        GC1 obj = new GC1(new File("test1.txt"));
        new GC1(new File("test2.txt")); // eligible for GC
    }
    protected void finalize()
    {
        this.file.close();
    }
}

```



```

/*class Object
{
    protected void finalize()
    {
        empty.....
    }
}*/

```

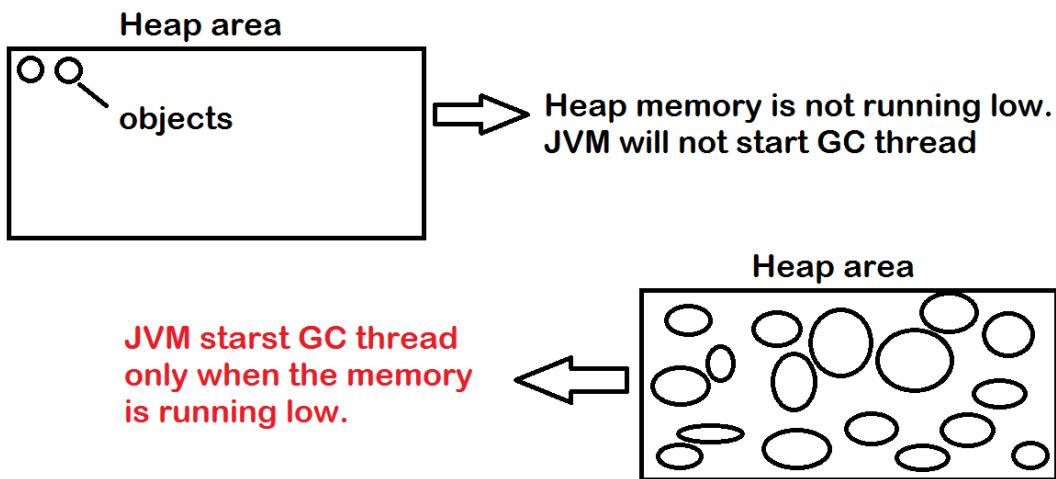
```

class GCDemo // extends Object
{
    GCDemo()
    {
        System.out.println(this+" has been created...");
    }
    protected void finalize()
    {
        System.out.println(this+" has been deleted...");
    }
    public static void main(String[] args)
    {
        new GCDemo();
        new GCDemo(); //un referenced objects, eligible for GC
    }
}

```

Question: Why Objects are not garbage collected in the above application?

- JVM starts the GC thread only when heap memory is running low.
- Multi tasking applies stress on the processor.
- If JVM runs GC thread continuously in the application that decreases application performance.
- Hence JVM starts GC thread only when required.



```

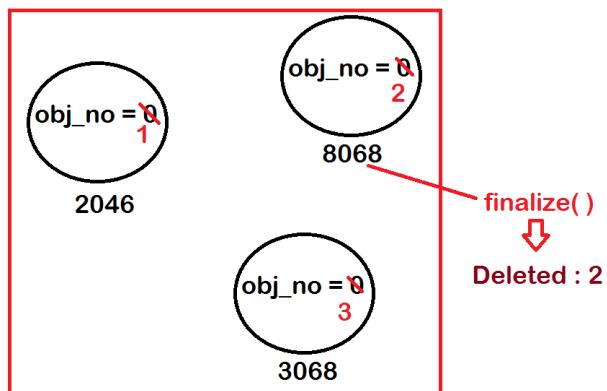
class GCDemo
{
    int obj_no;
    GCDemo(int obj_no)
    {
        this.obj_no = obj_no ;
        System.out.println("Created : "+this.obj_no);
    }
    public static void main(String[] args)
}

```

```

{
    for(int i=1; i<=16000 ; i++)
    {
        new GCDemo(i);
    }
}
protected void finalize()
{
    System.out.println("Deleted : "+this.obj_no);
}
}

```



Note: It is possible to request the JVM to start the GC thread using pre-defined method `System.gc()`. But it is not guarantee that JVM will start the Thread.

- If JVM executes GC-thread continuously in the application, that decreases application performance.
- Calling `gc()` method describes, program is responsible for application performance.

Question: Why request to start GC-thread?

Answer: To use the heap space quickly in the application.

```

/*class System
{
    public static void gc()
    {
        Runs GC thread.
    }
}*/



class GCDemo
{
    GCDemo()
    {
        System.out.println(this + " Created");
    }
}

```

```

    }
    public static void main(String[] args)
    {
        new GCDemo();
        new GCDemo();

        for(int i=1; i<=100 ; i++)
        {
            System.gc();
        }
    }
    protected void finalize()
    {
        System.out.println(this + " Deleted");
    }
}

```

Note : Generally in Real time application, we request to start GC-thread continuously using a thread with infinite loop as follows.

```

class GCDemo extends Thread
{
    int obj_no ;
    GCDemo()
    {
        //empty...
    }
    GCDemo(int obj_no)
    {
        this.obj_no = obj_no ;
        System.out.println("Created : "+this.obj_no);
    }

    public static void main(String[] args) throws Exception
    {
        GCDemo thread = new GCDemo();
        thread.start();

        for(int i=1; i<=100 ; i++)
        {
            new GCDemo(i);
            Thread.sleep(100);
        }

        thread.stop();
    }

    protected void finalize()
    {
    }
}

```

```

        System.out.println("Deleted : "+this.obj_no);
    }

    public void run()
    {
        while(true)
        {
            System.gc(); // infinite loop.....
        }
    }
}

```

Factory Class & Singleton Class :

- Class must be public
- Properties must be private
- We cannot instantiate these classes directly.
- Using factory method we can get instance of these class.
- “Factory method” is a static method available in these classes.
- “Factory method” creates object and returns from its definition.

Note : The only difference between Factory class and Singleton class is, “Factory class” can be instantiated many times where as “Singleton” class only once.

```

public class FactoryClass
{
    private FactoryClass()
    {
        // Other class cannot instantiate directly
    }

    public static FactoryClass factoryMethod()
    {
        FactoryClass obj = new FactoryClass();
        return obj;
    }
}

public class Singleton
{
    private static Singleton obj = null;
    private Singleton()
    {
        // logic
    }

    public static Singleton factoryMethod()
    {

```

```

        if(Singleton.obj == null)
    {
        Singleton.obj = new Singleton();
        return Singleton.obj ;
    }
    else
    {
        return Singleton.obj ;
    }
}

class Access
{
    public static void main(String[] args)
    {
        // FactoryClass fc = new FactoryClass(); // Error :
        FactoryClass fc1 = FactoryClass.factoryMethod();
        FactoryClass fc2 = FactoryClass.factoryMethod();

        System.out.println("fc1 : "+fc1);
        System.out.println("fc2 : "+fc2);

        // Singleton s = new Singleton(); // Error :
        Singleton s1 = Singleton.factoryMethod();
        Singleton s2 = Singleton.factoryMethod();

        System.out.println("s1 : "+s1);
        System.out.println("s2 : "+s2);
    }
}

```

Runtime class:

- Available in `java.lang` package.
- since jdk 1.0
- Runtime class represents JVM.
- The current runtime (virtual environment - JVM) can be obtained from the `getRuntime()` method.
- It is a Singleton class.

Question : Why Runtime class is Singleton?

- Runtim class represents JVM
- To run many class files in java application, there is only one JVM.
- JVM instance is sharable in java application but cannot be duplicated.

```

class JVMAccess
{
    public static void main(String[] args)
    {
        // Runtime jvm = new Runtime(); // Error :

        Runtime jvm1 = Runtime.getRuntime();
        Runtime jvm2 = Runtime.getRuntime();

        System.out.println("jvm1 : " + jvm1);
        System.out.println("jvm2 : " + jvm2);
    }
}

```

Requesting GC using Runtime gc() method:

```

class RuntimeDemo {
    protected void finalize(){
        System.out.println("finalized.....");
    }
    public static void main(String[] args) {
        new RuntimeDemo();
        Runtime r = Runtime.getRuntime();
        r.gc();
    }
}

/*public Process exec(String command) throws IOException
   Executes the specified string command in a separate process.
*/
class RuntimeDemo {
    public static void main(String[] args) throws java.io.IOException{
        Runtime r = Runtime.getRuntime();
        Process p1 = r.exec("notepad.exe");
        Process p2 = r.exec("mspaint.exe");
    }
}

```

Finding the memory size of Heap and JVM.

```

/*
long maxMemory() : returns JVM size in bytes
long totalMemory() : returns Heap size in bytes
long freeMemory() : available Heap memory in bytes
*/
class Memory {
    public static void main(String[] args) {
        long mm, tm, fm;
        Runtime r = Runtime.getRuntime();

```

```
        mm = r.maxMemory();
        tm = r.totalMemory();
        fm = r.freeMemory();
        System.out.println("JVM size : "+mm/(1024*1024));
        System.out.println("Heap size : "+tm/(1024*1024));
        System.out.println("free Heap size : "+fm/(1024*1024));
    }
}
```

Increasing Heap size and JVM size dependant to Main memory (RAM).

Commands to increase the size:

- Xms --> to set Heap size
- Xmx --> to set JVM size

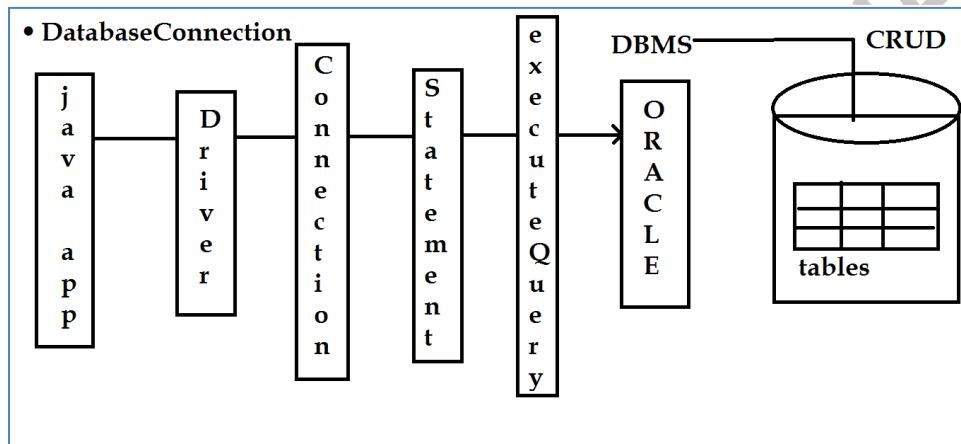
Note: size must be represents in mega bytes.

G:\>javac Memory.java

```
G:\>java -Xms250m -Xmx800m Memory
JVM size: 773
Heap size: 241
Free Heap size: 240
```

Inner Classes

- Defining a class within another class.
- It is also called nested class.
- Inner classes have clearly two benefits, name control & access control.
- In Java, this benefit is not as important because Java packages give the name control.
- Setting dependencies on objects to access their functionality is the advantage of Inner classes.
- The following diagram describes how to set dependencies on Objects.



The syntax for inner class is as follows:

```
[modifiers] class OuterClassName  
{  
    code...  
    [modifiers] class InnerClassName  
    {  
        code....  
    }  
}
```

Types of inner classes:

1. Static inner classes
2. Non-static inner classes
3. Local inner classes
4. Anonymous inner classes

Class files generation for inner classes :

```
class Outer  
{  
    // ....code  
    class Inner  
    {  
        // ....code  
    }
```

```
    }
}
```

class-files : Outer.class
Outer\$Inner.class

Accessing static members of static inner class:

- Static members of a class can be accessed directly using class Identity.

```
class Outer
{
    static void f1()
    {
        System.out.println("Outer class static method");
    }
    static class Inner
    {
        static void f2()
        {
            System.out.println("Inner class static method");
        }
    }
    public static void main(String args[])
    {
        Outer.f1();
        // Outer.f2(); //Error : Not defined in Outer class

        Outer.Inner.f2();
    }
}
```

Accessing non-static members of non-static inner class:

- Generally accessing non-static members of class is possible using object reference.
- We can't access inner class non-static functionality using Outer class Object.
- We need to instantiate inner class using Outer class object address.
- It is the example of setting dependencies on objects.

```
class Outer
{
    void f1()
    {
        System.out.println("Outer class non-static method");
    }
    class Inner
    {
        void f2()
        {

```

```

        System.out.println("Inner class non-static method");
    }
}
public static void main(String args[ ])
{
    Outer obj1 = new Outer();
    obj1.f1();
    // obj1.f2(); //Error : not defined in Outer class

    Outer.Inner obj2 = obj1.new Inner();
    obj2.f2();
}
}

```

Accessing non-static members of static inner class:

static inner classes can be instantiated using outer class name instead of object.

```

class Outer
{
    static class Inner
    {
        void fun()
        {
            System.out.println("Inner class non-static method");
        }
    }
    public static void main(String args[ ])
    {
        Outer.Inner obj = new Outer.Inner();
        obj.fun();
    }
}

```

Accessing static members of non-static inner class: It is not allowed to define static members(free accessible) inside the non-static context(restricted access). Hence non-static inner class doesn't have static declarations.

```

class Outer
{
    class Inner
    {
        static void fun() //Error :
        {
            System.out.println("Inner class static method");
        }
    }
}

```

For example, placing box-office(static) inside theatre(non-static).

```
class ShoppingMall
{
    class Theatre
    {
        static void boxOffice() //Error :
        {
            System.out.println("Inner class static method");
        }
    }
}
```

Note : private members of one class cannot be accessed from another class even in Parent-Child relation also

```
class First
{
    private static int a=100;
}
class Second extends First
{
    public static void main(String[] args)
    {
        System.out.println("a value : "+First.a);
    }
}
```

But an Outer class can access even private members of inner class:

```
class Outer
{
    class Inner
    {
        private void method()
        {
            System.out.println("Inner class private method");
        }
    }
    public static void main(String args[ ])
    {
        Outer obj1 = new Outer();
        Outer.Inner obj2 = obj1.new Inner();
        obj2.method();
    }
}
```

In the above program we can instantiate Inner class using Outer object as follows...

```
class Outer
```

```

{
    class Inner
    {
        private void method()
        {
            System.out.println("Inner class private method");
        }
    }
    public static void main(String args[ ])
    {
        new Outer().new Inner().method();
    }
}

```

Static Members of Outer class are directly visible to inner class including private:

```

class Outer
{
    private static int x = 100 ;
    class Inner
    {
        private void method()
        {
            System.out.println("Outer's x val : "+Outer.x);
        }
    }
    public static void main(String args[ ])
    {
        new Outer().new Inner().method();
    }
}

```

Note : non-static members of outer class including private can be accessed from the inner class as follows.....

<outer_class_name>.this.<outer_class_non-static-member>

```

class Outer
{
    private int x ;
    Outer(int x)
    {
        this.x = x ;
    }
    class Inner
    {
        int x ;
        Inner(int x)

```

```

    {
        this.x = x ;
    }
    private void method()
    {
        System.out.println("Outer's x val : "+Outer.this.x);
        System.out.println("Inner's x val : "+this.x);
    }
}
public static void main(String args[ ])
{
    Outer obj1 = new Outer(100);
    Outer.Inner obj2 = obj1.new Inner(200);
    obj2.method();
}
}

```

Local inner classes :

- Defining a class inside the block or method or constructor.
- access modifiers cannot be applied to the local inner classes.
- using Local inner classes we can achieve name control benefit among the classes.
- It is allowed to define more than one class with the same name in two different blocks of same Outer class.
- Local inner class cannot be static.

```

class Outer
{
    static
    {
        class LocalInner
        {
            //Outer$1LocalInner.class
        }
    Outer
    {
        class LocalInner
        {
            //Outer$2LocalInner.class
        }
    }
}

```

Scope of local inner class is always restricted to that block where it has defined

```
class Outer
{
```

```
    static
```

```

{
    class LocalInner
    {
        void check()
        {
            System.out.println("Class inside block...");
        }
    }
    new LocalInner().check();
}
Outer()
{
    class LocalInner
    {
        void check()
        {
            System.out.println("Class inside constructor...");
        }
    }
    new LocalInner().check();
}
public static void main(String args[ ])
{
    new Outer();
}
}

```

More than one level of local inner classes(as shown below) creates complex user defined data types.

```

class Outer
{
    Outer()
    {
        class LocalInner
        {
            void check()
            {
                System.out.println("Class inside constructor...");
                class InnerInner
                {
                    void check()
                    {
                        System.out.println("Class inside method....");
                    }
                }
                new InnerInner().check();
            }
        }
    }
}

```

```

        new LocalInner().check();
    }
    public static void main(String args[ ])
    {
        new Outer();
    }
}

```

Anonymous inner class :

- Defining the class with no name is called Anonymous inner class.
- It is also one type of Local inner class.
- Constructors are not allowed to define because no class name.
- Anonymous inner classes are mainly used to implement interfaces.
- Providing definition for a standard(interface) directly inside a method.

**Here static method returns reference of Connection interface.
Now we implement this concept using anonymous inner class.**

Connection con = DriverManager.getConnection();

Interface class static method

```

interface Connection
{
    void check();
}
class DriverManager
{
    public static Connection getConnection()
    {
        Connection con = new Connection()
        {
            public void check()
            {
                System.out.println("anonymous.....");
            }
        };
        return con;
    }
}
class DatabaseConnection
{
    public static void main(String args[])
}

```

```

    {
        Connection con = DriverManager.getConnection();
        con.check();
    }
}

```

- Returning interface as an argument to method using anonymous inner class.
- In the above application, a method is returning Connection object as it was getter method.
- Using setter method, if we want to pass Connection object as a parameter is as follows.

```

interface Connection
{
    void check();
}
class DriverManager
{
    public static void setConnection(Connection con)
    {
        con.check();
    }
}
class DatabaseConnection
{
    public static void main(String args[])
    {
        DriverManager.setConnection(new Connection()
        {
            public void check()
            {
                System.out.println("anonymous.....");
            }
        });
    }
}

```

Generally we implements Runnable interface with any duplicate identity(class)

```

class RunnableObject implements Runnable
{
    public void run()
    {
        System.out.println("Thread logic....");
    }
}
class ThreadClass
{
    public void main(String args[ ])
    {

```

```

        RunnableObject obj = new RunnableObject();
        Thread thread = new Thread(obj);
        thread.start();
    }
}

```

- Implementing Runnable interface using anonymous inner class :
- Implementing interface with duplicate class is bit complex when compare with anonymous inner class.
- Anonymous inner class approach is an easy way to implement an interface.

```

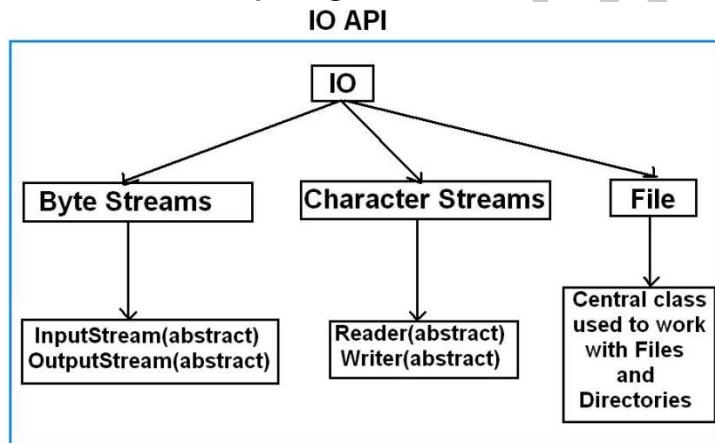
class RunnableAnonymous
{
    public static void main(String[] args)
    {
        Thread t = new Thread(new Runnable()
        {
            public void run()
            {
                System.out.println("anonymous....");
            }
        });
        t.start();
    }
}

```

I/O Streams

- In general, Stream is nothing but flow of something, for example water.
- In java, a stream represents a sequence of objects (usually bytes, characters, etc.), which can be accessed in sequential order.
- An I/O Stream represents an input source or an output destination
- Typical operations on a stream:
 1. **Byte Streams:** used to read and write stream byte by byte
 2. **Character Streams:** allows the programmer to read character by character from the stream.
 3. **Buffered Streams:** used to read group of bytes or characters from the desired stream to reduce the number of calls to the stream.
 4. **Data Streams:** handle binary I/O of primitive data types.
 5. **Object Streams:** defined to work with object stream like in serialization.

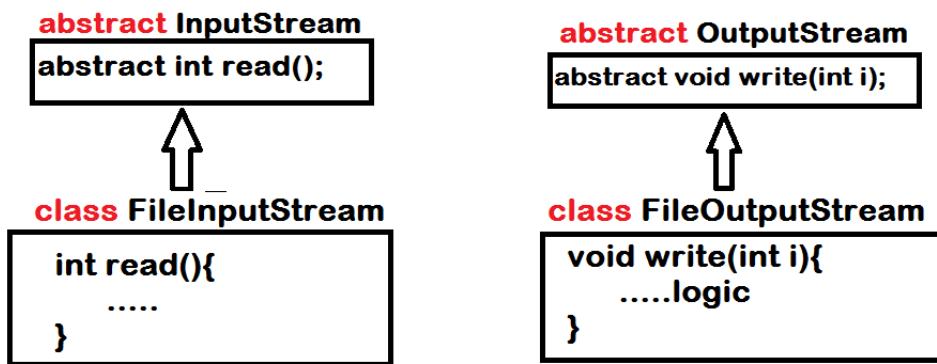
The following diagram describes how IO packages classes to be used.



Byte Streams

- Byte streams are those in which the data will be transferred one byte at a time between primary memory to secondary memory and secondary memory to primary memory either locally or globally.
- `java.io.*` package contains some set of classes and interfaces to perform byte stream operations.
- Programs use byte streams to perform input and output operations of 8-bit data.
- All byte stream classes are descended from `InputStream` and `OutputStream` classes
- The two main classes used to perform Byte stream operations are `InputStream` & `OutputStream`.
- These classes are abstract classes.

- We use extensions such as FileInputStream & FileOutputStream classes to implements programming.



FileInputStream class:

- This is the concrete (which we can create an object or it contains all defined methods) sub-class of all InputStream class.
 - **public class FileInputStream extends InputStream**
- This class is always used to open the file in read mode.
- Opening the file in read mode is nothing but creating an object of FileInputStream class.
FileInputStream (String fname) throws FileNotFoundException
FileInputStream fis=new FileInputStream ("abc.txt");
- If the file name abc.txt does not exist then an object of FileInputStream fis is null and hence we get FileNotFoundException.

FileOutputStream class:

- This is the concrete sub-class of all OutputStream classes.
public class FileOutputStream extends OutputStream
- This class is always used for opening the file in write mode is nothing but creating an object of FileOutputStream class.
FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
- If the flag is true the data will be appended to the existing file else if flag is false the data will be overlapped with the existing file data.
- If the file is opened in write mode then the object of FileOutputStream will point to that file which is opened in write mode.
- If the file is unable to open in write mode an object of FileOutputStream contains null.

```

/*class FileInputStream
{
    public FileInputStream(String name) throws FileNotFoundException
    {
        opens a file in read mode.....
    }
    public void close() throws IOException
}
  
```

```

    {
        closes the file
    }
    public int read()
    {
        on success, returns ASCII value of character
        on failure, returns -1 .
    }
}
*/
import java.io.*;
class ByteStreams
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis = null ;
        try
        {
            fis = new FileInputStream("d:/ByteStreams.java");
            System.out.println("File opened....");

            int ch ;
            while((ch=fis.read()) != -1)
            {
                System.out.print((char)ch);
            }
        }
        finally
        {
            if(fis != null)
            {
                fis.close();
                System.out.println("File closed...");
            }
        }
    }
}

```

Writing data into file:

- We need to represent output file in the construction of FileOutputStream class.
- If file is present with the specified name, it opens the file and deletes the existing content.
- If file is not present, it creates the new file with specified name.

Append mode :

- Used to append the data.
- If file is present, it opens and place the cursor at the end of the file to add new contents.
- If file is not present, it creates new file.

```

/*class FileOutputStream
{
    FileOutputStream(String s)
    {
        open a file in write mode....
    }
    FileOutputStream(String s, boolean append)
    {
        append = true ;
        opens a file in append mode....
    }
}

import java.io.*;
class ByteStreams
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis = null ;
        FileOutputStream fos = null ;
        try
        {
            fis = new FileInputStream("d:/ByteStreams.java");
            fos = new FileOutputStream("d:/Output.txt");

            int ch ;
            while((ch=fis.read()) != -1)
            {
                fos.write(ch);
            }
            System.out.println("File copied....");
        }
        finally
        {
            if(fis != null)
            {
                fis.close();
            }
            if(fos != null)
            {
                fos.close();
            }
        }
    }
}

```

Append data program :

```

import java.io.*;
class ByteStreams
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis = null ;
        FileOutputStream fos = null ;
        try
        {
            fis = new FileInputStream("d:/ByteStreams.java");
            fos = new FileOutputStream("d:/Output.txt" , true); // append mode...

            int ch;
            while((ch=fis.read()) != -1)
            {
                fos.write(ch);
            }
            System.out.println("Content appended.....");
        }
        finally
        {
            if(fis != null)
            {
                fis.close();
            }
            if(fos != null)
            {
                fos.close();
            }
        }
    }
}

```

Note : Byte streams are mainly used to work with binary files such as images, audio files....

```

import java.io.*;
class CopyImage
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis = null ;
        FileOutputStream fos = null ;
        try
        {
            fis = new FileInputStream("g:/input.jpg");
            fos = new FileOutputStream("g:/output.jpg");

```

```

        int ch;
        while((ch=fis.read()) != -1)
        {
            fos.write(ch);
        }
        System.out.println("Image copied successfully...");
    }
    finally
    {
        if(fis != null)
        {
            fis.close();
        }
        if(fos != null)
        {
            fos.close();
        }
    }
}
}

```

Character Streams

- Prior to JDK 1.1, the input and output classes (mostly found in the `java.io` package) supported only 8-bit "byte" streams.
- In JDK 1.1 the concept of 16-bit Unicode "character" streams was introduced.
- While the byte streams were supported via the `java.io.InputStream` and `java.io.OutputStream` classes and their subclasses, character streams are implemented by the `java.io.Reader` and `java.io.Writer` classes and their subclasses.
- For example, to read files using character streams, you'd use the `java.io.FileReader` class; to read using byte streams you'd use `java.io.FileInputStream`.
- Unless you're working with binary data such as image and sound files, you should use readers and writers to read and write information for the following reason:
- **Programs that use character streams are easier to internationalize because they're not dependent upon a specific character encoding.**
- To bridge the gap between the byte and character-stream classes, Java provides the `java.io.InputStreamReader` and `java.io.OutputStreamWriter` classes.
- The only purpose of these classes is to convert byte data into character-based data according to a specified (or the platform default) encoding.

FileReader:

- Inherited from `InputStreamReader`.
- Used to read character by character from any underlying stream.
- We can't read like images, videos, sound files as these are not come under byte streams.

FileWriter:

- Inherited from `OutputStreamWriter`.
- Used to Write character by character to any kind of file or output device.

```

import java.io.*;
class CharacterStreams
{
    public static void main(String[] args) throws IOException
    {
        FileReader fr = null ;
        FileWriter fw = null ;
        try
        {
            fr = new FileReader("g:/input.jpg");
            fw = new FileWriter("g:/output.jpg");
            int ch ;
            while((ch = fr.read()) != -1)
            {
                fw.write(ch);
            }
            System.out.println("copied....");
        }
        finally
        {
            if(fr != null)
            {
                fr.close();
            }
            if(fw != null)
            {
                fw.close();
            }
        }
    }
}

```

Object streams (Serialization)

- Serialization is the concept of converting Object state into Persistent(permanent) state.
- Generally in Java application, objects will be created in Heap area and will be deleted as soon as execution completes.
- For every application it is mandatory to maintain the data permanently.
- Java API providing classes and interfaces to perform serialization.
 - a. java.io.Serializable(interface)
 - b. java.io.ObjectInputStream(class)
 - c. java.io.ObjectOutputStream(class)

Note :

- class must implements Serializable interface to serialize that object.
- In the process of serialization, **transient** variables will not participate.

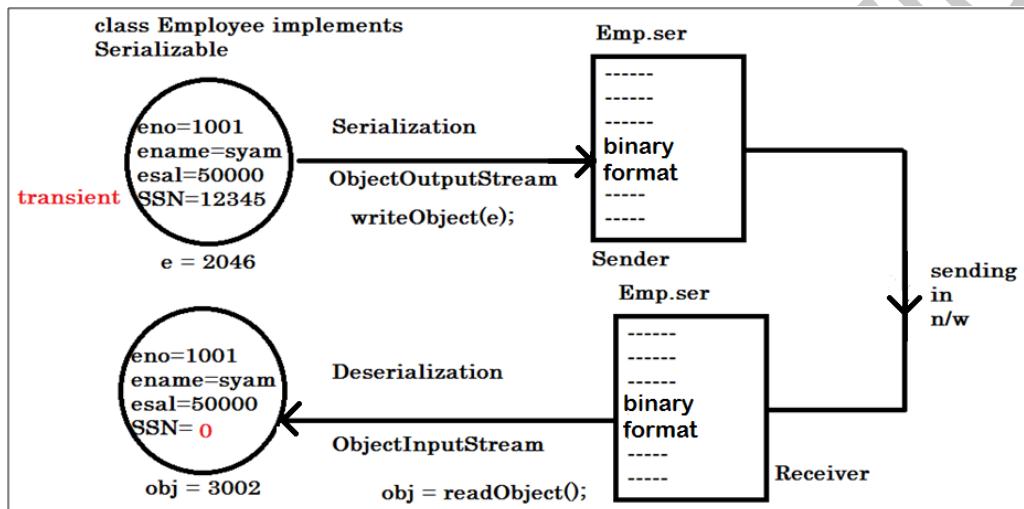
- Serializable information must be stored into a file with .ser extension.
- Serializable file will be in encrypted format, hence we can transfer the file over the network.
- In the network, receiver should de-serialize the file to check the information.

Serialization : Convert Object --> File

De-Serialization : Convert File --> Object

The following example code shows how to serialize and de-serialize of an object, it includes three programs,

1. Defining a class that includes some properties of an employee.
2. Serializing Employee object.
3. De-serializing Employee object.



```
class Employee implements java.io.Serializable
{
    int eno ;
    String ename ;
    double esal ;
    transient int SSN ;
    Employee(int eno, String ename, double esal, int SSN)
    {
        this.eno = eno ;
        this.ename = ename ;
        this.esal = esal ;
        this.SSN = SSN ;
    }
}
```

Serialization:

```
import java.io.*;
class Serialization
{
    public static void main(String[] args) throws Exception
```

```

{
    Employee e = new Employee(101,"Syam",50000,12345);
    FileOutputStream fos = null ;
    ObjectOutputStream oos = null ;
    try
    {
        fos = new FileOutputStream("d:/Emp.ser");
        oos = new ObjectOutputStream(fos);
        oos.writeObject((Object)e);
        System.out.println("Serialized.....");
    }
    finally
    {
        if(oos != null)
        {
            oos.close();
        }
        if(fos != null)
        {
            fos.close();
        }
    }
}
}

```

De Serialization:

```

import java.io.*;
class DeSerialization
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis = null ;
        ObjectInputStream ois = null ;
        try
        {
            fis = new FileInputStream("d:/Emp.ser");
            ois = new ObjectInputStream(fis);
            Employee e1 = (Employee)ois.readObject();
            System.out.println("De-Serialized.....");

            System.out.println("Eno : "+e1.eno+"\nEname : "+e1.ename+"\nEsal :
"+e1.esal+"\nSSN : "+e1.SSN);
        }
        finally
        {
            if(ois != null)
            {

```

```
        ois.close();  
    }  
    if(fis != null)  
    {  
        fis.close();  
    }  
}
```

transient keyword:

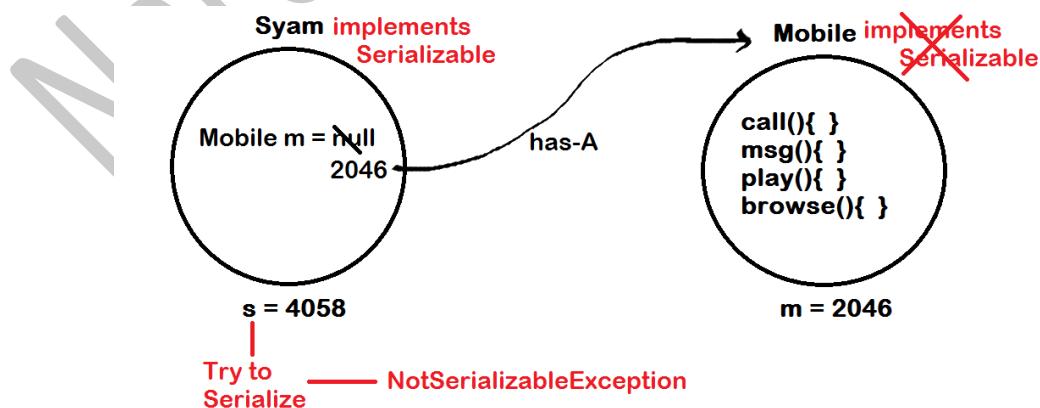
- The keyword transient in Java used to indicate that the variable should not be serialized.
 - By default all the variables in the object is converted to persistent state.
 - In some cases, you may want to avoid some variables (confidential data) because you don't have the necessity to transfer across the network.
 - So, you can declare those variables as transient. If the variable is declared as transient, then it will not be persisted means will not be participated in Serialization process.

How many methods Serializable has? If not, then what is the purpose of Serializable interface?

- Serializable interface doesn't have any method.
 - It is empty interface.
 - It is also called Marker Interface or Tagged interface in Java.
 - Marker interface is used as a information tag about an object.
 - It adds special behavior to Object of class which is implementing it.
 - Examples
 - java.io.Serializable interface
 - java.lang.Cloneable interface

Can we Serialize an object which is pointing to a non-serializable object?

- One class(implements Serializable) pointing to(has-A relation with) another object which is not Serializable.
 - If we try to serialize the object results Exception.



```

import java.io.*;
class Car // implements java.io.Serializable
{
    private Vehicle model_no;
    private int bhp;
    public Car(Vehicle model, int bhp )
    {
        model_no=model;
        this.bhp=bhp;
    }
}
class Vehicle implements java.io.Serializable
{
    private int model_no;
    public Vehicle( int model_no)
    {
        this.model_no=model_no;
    }
}
class Serialize
{
    public static void main( String args[])
    {
        Vehicle v = new Vehicle(1134);
        Car c= new Car( v, 256);
        try
        {
            FileOutputStream fs = new FileOutputStream( "Serialize.ser" );
            ObjectOutputStream os = new ObjectOutputStream( fs );
            os.writeObject( c );
            os.close();
        }
        catch( FileNotFoundException fnf )
        {
            fnf.printStackTrace();
        }
        catch( IOException i )
        {
            i.printStackTrace();
        }
    }
}

```

If a class is Serializable but its super class is not, what will be the state of the instance variables inherited from super class after de serialization?

- Java serialization process only continues in object hierarchy till the class is Serializable i.e. implements Serializable interface in Java and values of the instance variables inherited from

super class will be initialized by calling constructor of Non-Serializable Super class during de serialization process.

- Once the constructor chaining will start it wouldn't be possible to stop, hence even if classes higher in hierarchy not implements Serializable interface, their constructor will be executed.

```
class NotSerializable{  
    int a;  
}  
public class Employee extends NotSerializable implements java.io.Serializable{  
    public String name;  
    public String address;  
    public transient int SSN;  
    public static int number;  
}
```

Which kind of variables is not serialized during Java Serialization?

- static and transient variables.
- Since static variables belong to the class and not to an object they are not the part of the state of object so they are not saved during Java Serialization process.
- As Java Serialization only persist state of object and not object itself.
- Transient variables are also not included in java serialization process and are not the part of the object's serialized state.

POJO v/s Bean class:

The only difference between POJO & Bean class is,

POJO class doesn't implement Serializable interface where as Bean class does it.

```
public class POJO  
{  
    All the POJO rules  
}  
  
public class Bean implements java.io.Serializable  
{  
    All the POJO rules  
}
```

Buffered Streams :

- Buffer is a temporary storage area while processing the information.
- In some of the applications we use Buffers to process the data instead of fetching information from the secondary memory every time.
- Processed information will be stored in Buffer temporarily.
- Once we close(save) the Buffer, the info saved permanently in Secondary memory.

BufferedReader:

BufferedReader buffer character to read characters, arrays and lines. It read text from a character-input stream.

```
import java.io.*;
class BufferedReaderDemo
{
    public static void main(String[] args)
    {
        try
        {
            FileReader f=new FileReader("IN.txt");
            BufferedReader br = new BufferedReader(f);
            String str;
            while ((str = br.readLine()) != null)
            {
                System.out.println(str);
            }
        } catch (IOException e) {}
    }
}
```

```
import java.io.*;
class BufferedStreams
{
    public static void main(String[] args) throws Exception
    {
        FileReader fr = null ;
        FileWriter fw = null ;
        BufferedWriter bw = null ;
        try{
            fr = new FileReader("d:/BufferedStreams.java");
            fw = new FileWriter("g:/buffer.txt");
            bw = new BufferedWriter(fw);

            int ch;
            while( (ch=fr.read()) != -1 ){
                bw.write(ch);
            }
        }
        finally{
            if(bw != null){
                bw.close();
            }
            if(fr != null){
                fr.close();
            }
            if(fw != null){
                fw.close();
            }
        }
    }
}
```

```
        }
    }
}
```

StringTokenizer class :

- Available in util package.
- Since jdk 1.0.
- Used to split a string into Tokens(words).

```
import java.util.StringTokenizer;
class Tokenizer
{
    public static void main(String[] args)
    {
        String line = "this is a test";
        StringTokenizer st = new StringTokenizer(line);
        while (st.hasMoreTokens())
        {
            System.out.println(st.nextToken());
        }
    }
}
```

WAP to count number of words in a given file :

```
import java.io.*;
import java.util.StringTokenizer ;
class WordCount
{
    static int count = 0 ;
    public static void main(String[] args) throws Exception
    {
        FileReader fr = null ;
        BufferedReader br = null ;
        Try
        {
            fr = new FileReader("d:/WordCount.java");
            br = new BufferedReader(fr);

            String line ;
            while( (line = br.readLine()) != null)
            {
                StringTokenizer st = new StringTokenizer(line);
                while (st.hasMoreTokens())
                {
                    System.out.println(st.nextToken());
                    count++;
                }
            }
        }
    }
}
```

```

        }
        System.out.println("Word count is :" + count);
    }
    finally{
        if(br != null){
            br.close();
        }
        if(fr != null){
            fr.close();
        }
    }
}
}
}

```

BufferedInputStream

- BufferedInputStream adds extra functionalities to another InputStream.
- BufferedInputStream creates internal array with buffer size on the machine to read and write group of bytes.
- We can skip the data while reading from the internal buffer array using pre-defined skip() method.
- We can also reset the position to be read using two pre-defined functions mark() and reset().

Copying image file:

```

import java.io.*;
class BufferedInputStreamDemo
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis=new FileInputStream("sonoo.jpg");
        BufferedInputStream bis=new BufferedInputStream(fis);
        FileOutputStream fos=new FileOutputStream("result.jpg");
        int ch;
        while((ch=bis.read())!=-1)
        {
            fos.write(ch);
        }
    }
}

```

Use of skip() method:

```

import java.io.*;
class BufferedInputStreamDemo
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis=new FileInputStream("BufferedInputStreamDemo.java");
        BufferedInputStream bis=new BufferedInputStream(fis);

```

```

        FileOutputStream fos=new FileOutputStream("out.txt");
        bis.skip(200);
        int ch;
        while((ch=bis.read())!=-1)
        {
            fos.write(ch);
        }
    }
}

```

Use of skip(), reset() and mark() methods:

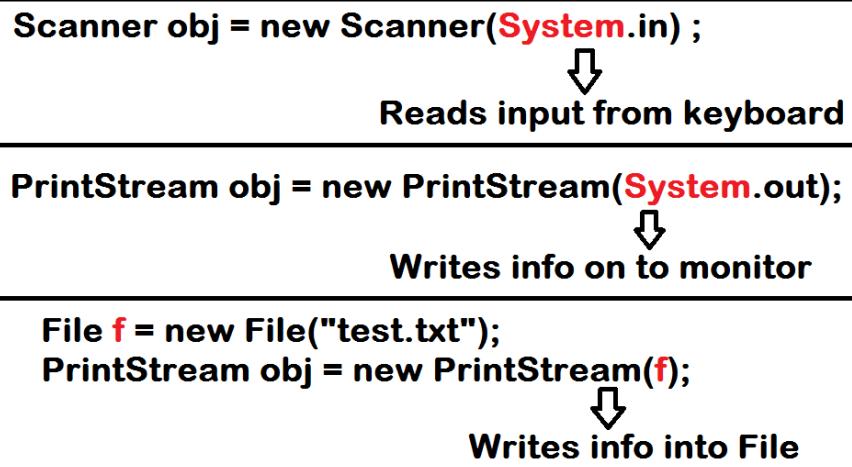
```

import java.io.*;
class BufferedInputStreamDemo
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis=new FileInputStream("BufferedInputStreamDemo.java");
        BufferedInputStream bis=new BufferedInputStream(fis);
        FileOutputStream fos=new FileOutputStream("out.txt");
        bis.mark(1);
        bis.skip(200);
        bis.reset();
        int ch;
        while((ch=bis.read())!=-1)
        {
            fos.write(ch);
        }
    }
}

```

PrintStream class :

- The PrintStream class is obtained from the FilterOutputStream class that implements a number of methods for displaying textual representations of Java primitive data types.
- Unlike other output streams, a PrintStream never throws an IOException and the data is flushed to a file automatically i.e. the flush() method is automatically invoked after a byte array is written.
- The constructor of the PrintStream class is written as:
PrintStream (java.io.OutputStream out); //create a new print stream
- The print() and println() methods of this class give the same functionality as the method of standard output stream.



Simple java program that prints hello world:

```

import java.io.PrintStream;
class Example
{
    public static void main(String[] args)
    {
        PrintStream ps=new PrintStream(System.out);
        ps.println("Hello World!");
    }
}

```

Example:

```

import java.io.*;
public class PrintStreamDemo
{
    public static void main(String[] args)
    {
        try
        {
            FileOutputStream out = new FileOutputStream("out.txt");
            PrintStream p = new PrintStream(out);
            p.println("this text will be written into out.txt after run");
            System.out.println("text has been written, go and check out.txt in the
current directory");
            p.close();
        } catch (Exception e){}
    }
}

```

ByteArrayInputStream

- It is an implementation of an input stream that uses a byte array as the source.
- This class has two constructors, each of which requires a byte array to provide the data source:

ByteArrayInputStream(byte array[])
ByteArrayInputStream(byte array[], int start, int numBytes)

- Here, array is the input source.
- The second constructor creates an InputStream from a subset of your byte array that begins with the character at the index specified by start and is numBytes long.

```
import java.io.*;
class ByteArrayInputStreamDemo
{
    public static void main(String args[]) throws IOException
    {
        String tmp = "abcdefghijklmnopqrstuvwxyz";
        byte b[] = tmp.getBytes();
        ByteArrayInputStream bais1 = new ByteArrayInputStream(b);
        ByteArrayInputStream bais2= new ByteArrayInputStream(b,3,3);

        ByteArrayOutputStream baos1= new ByteArrayOutputStream();
        ByteArrayOutputStream baos2= new ByteArrayOutputStream();
        int ch;
        while((ch=bais1.read())!=-1){
            baos1.write(ch);
        }
        while((ch=bais2.read())!=-1) {
            baos2.write(ch);
        }
        byte b1[]=baos1.toByteArray();
        byte b2[]=baos2.toByteArray();

        System.out.println("b1 array contents");
        for(int i=0;i<b1.length;i++){
            System.out.print((char)b1[i]);
        }
        System.out.println("b2 arrray contents :");
        for(int i=0;i<b2.length;i++){
            System.out.print((char)b2[i]);
        }
    }
}
```

DataInputStream

- Is used to read java primitive data types from an underlying input stream.
- It is inherited from FilterInputStream class.
- It has only one constructor, and it takes byte stream object as an argument.

String readLine() :

- Deprecated. This method does not properly convert bytes to characters.

- As of JDK 1.1, the preferred way to read lines of text is via the BufferedReader.readLine() method.
- Programs that use the DataInputStream class to read lines can be converted to use the BufferedReader class by replacing code of the form.

```
import java.io.*;
public class DataInputStreamDemo
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis=new FileInputStream("DataInputStreamDemo.java");
        DataInputStream dis = new DataInputStream(fis);
        while (dis.available() != 0)
        {
            System.out.println(dis.readLine());
        }
        dis.close();
    }
}
```

Reading and Writing Primitive types:

```
import java.io.*;
public class DataInputStreamDemo
{
    public static void main(String[] args)
    {
        int eno=1001;
        String ename="srinivas";
        try {
            FileOutputStream fos = new FileOutputStream("emp.txt");
            DataOutputStream dos = new DataOutputStream(fos);
            dos.writeInt(eno);
            dos.writeUTF(ename);
            dos.close();
            FileInputStream fis = new FileInputStream("emp.txt");
            DataInputStream dis = new DataInputStream(fis);
            int id= dis.readInt();
            System.out.println("Id: " + id);
            String name = dis.readUTF();
            System.out.println("Name: " + name);
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Working with Files

- File is a class, available in java.io directory used to work with Files and Directories.
- The instance of File class represents a File or Directory.
- To work with files or directories we must create object of file class.
- When you create object for File class it will not search for that file or directory, externally we should call some pre-defined methods to perform operations on such file/directory.
- Thus the statement can be written as:

```
File f = new File("<filename>");
```

Method	Description
f.exists()	Returns true if file exists.
f.isFile()	Returns true if this is a normal file.
f.isDirectory()	true if "f" is a directory.
f.getName()	Returns name of the file or directory.
f.isHidden()	Returns true if file is hidden.
f.lastModified()	Returns time of last modification.
f.length()	Returns number of bytes in file.
f.getPath()	path name.
f.delete()	Deletes the file.
f.renameTo(f2)	Renames f to File f2. Returns true if successful.
f.createNewFile()	Creates a file and may throw IOException.

Ques : How File class represents both File & Directory ?

While instantiating File class, if we specify extension, it is referred as file or else it is directory.

```
File file = new File("Sample.txt");
```

```
File directory = new File("Check");
```

Creating a File:

```
/*class File
{
    public boolean exists()
    {
        Tests whether the file or directory is present
    }
    public boolean createNewFile() throws IOException{
        creates a file in specified path...
    }
}*/
```

```

import java.io.File ;
class CreateFile
{
    public static void main(String[] args) throws Exception
    {
        File f = new File("g:/test.txt");
        if(f.exists())
        {
            System.out.println("File is present");
        }
        else
        {
            f.createNewFile();
            System.out.println("File created...");
        }
    }
}

```

Creating directories:

```

import java.io.*;
class.CreateDirectory
{
    public static void main(String[] args)
    {
        String s1 = "d:/test";
        String s2 = "d:/test1/test2/test3";
        File d1 = new File(s1);
        File d2 = new File(s2);
        if(d1.mkdir())
            System.out.println(s1+" created...");
        if(d2.mkdirs())
            System.out.println(s2+" created...");
    }
}

```

Deleting Files & Directories:

```

import java.io.File ;
class Deletion
{
    public static void main(String list[ ])
    {
        if(list.length == 0)
        {
            System.out.println("no input.....");
        }
        else
        {

```

```

        for(int i=0 ; i<list.length ; i++)
    {
        Deletion.fileRemove(list[i]);
    }
}

static void fileRemove(String name)
{
    File target = new File(name);
    if(target.exists())
    {
        if(target.delete())
            System.out.println(name+" deleted...");
        else
            System.out.println("failed in deletion of "+name);
    }
    else
    {
        System.out.println(name+" not present");
    }
}
}

```

Renaming the file:

```

import java.io.File;
import java.io.IOException;
public class Rename
{
    public static void main(String[] argv) throws IOException
    {
        File f = new File("F:/in.txt");
        f.renameTo(new File("out.java"));
    }
}

```

File copy:

```

import java.io.*;
public class Copy
{
    public static void main(String[] args) throws IOException
    {
        File inputFile = new File("Copy.java");
        File outputFile = new File("OutCopy.java");
        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);
        int c;
        while ((c = in.read()) != -1)
            out.write(c);
    }
}

```

```

        in.close();
        out.close();
    }
}

```

Constructing a File Name path

In Java, it is possible to set dynamic path, using the static constant **File.separator** or **File.separatorChar** to specify the file name in a platform-independent way. If we are using Windows platform then the value of this separator is '\'.

```

import java.io.File;
public class Main
{
    public static void main(String[] args)
    {
        String filePath = File.separator + "Java" + File.separator + "IO";
        File file = new File(filePath);
        System.out.println(file.getPath());
    }
}

```

Different Operating Systems recognize paths differently as follows...

windows : String s2 = dir1/dir2/dir3

Unix : String s2 = dir1\dir2\dir3

Global path : String s2 = dir1\\dir2
or
String s2 = dir1//dir2

```

import java.io.File;
public class Main {
    public static void main(String[] args)
    {
        String str="f:/Dir1/dir2/dir3";
        boolean isDirectoryCreated = (new File(str).mkdirs());
        if (isDirectoryCreated)
        {
            System.out.println("successfully created");
        }
        else
        {
            System.out.println("creation failed");
        }
    }
}

```

Check if the file permission allows:

```
boolean canExecute(); – return true, file is executable; false is not.  
boolean canWrite(); – return true, file is writable; false is not.  
boolean canRead(); – return true, file is readable; false is not.
```

Set the file permission:

```
boolean setExecutable(boolean); – true, allow execute operations; false to disallow it.  
boolean setReadable(boolean); – true, allow read operations; false to disallow it.  
boolean setWritable(boolean); – true, allow write operations; false to disallow it.
```

```
import java.io.File;  
public class SetWritableTest  
{  
    public static void main(String[] args) throws SecurityException  
    {  
        File file = new File("ddd.txt");  
        if (file.exists())  
        {  
            boolean bval = file.setWritable(false);  
            System.out.println("set the owner's write permission: " + bval);  
        }  
        else  
        {  
            System.out.println("File cannot exists: ");  
        }  
    }  
}
```

Hidden file: a file is considered to be hidden, if it's marked as hidden in the file properties.

```
import java.io.File;  
import java.io.IOException;  
public class FileHidden  
{  
    public static void main(String[] args) throws IOException  
    {  
        File file = new File("FileHidden.java");  
        if(file.isHidden())  
        {  
            System.out.println("This file is hidden");  
        }  
        else  
        {  
            System.out.println("This file is not hidden");  
        }  
    }  
}
```

Reflection API

- Reflection API is a powerful technique (that provides the facility) to find-out its environment as well as to inspect the class itself.
- Reflection API was included in Java 1.1.
- The classes of Reflection API are the part of the package `java.lang.reflect`.
- It allows the user to get the complete information about interfaces, classes, constructors, fields and various methods being used.
- It also provides an easy way to create a Java Application that was not possible before Java 1.1.

Need of reflection:

- Reflection is a feature in the Java programming language.
- It allows an executing Java program to examine or "introspect" upon itself, and manipulates internal properties of the program.
- For example, it's possible for a Java class to obtain the names of all its members and display them.
- For example, there is no way in a Pascal, C, or C++ program to obtain information about the functions defined within that program.
- The reflection API represents, or reflects, the classes, interfaces, and objects in the current Java Virtual Machine.
- With the reflection API you can:
 - Determine the class of an object.
 - Get information about a class's modifiers, fields, methods, constructors, and super classes.
 - Find out what constants and method declarations belong to an interface.
 - Create an instance of a class whose name is not known until runtime.
 - Get and set the value of an object's field, even if the field name is unknown to your program until runtime.
 - Invoke a method on an object, even if the method is not known until runtime.
 - Create a new array, whose size and component types is not known until runtime, and then modify the array's components.

Class information of specified Object :

```
class Example
{
    //empty.....
}
class ClassInfo
{
    static Example fun()
    {
        return new Example();
    }
}
```

```

        }
    }
class Reflection
{
    public static void main(String[] args)
    {
        Object obj = ClassInfo.fun();
        Class c = obj.getClass();
        String name = c.getName();
        System.out.println("obj-class name : "+name);
    }
}

```

finding super class information :

```

import java.awt.Checkbox;
class SuperClassInfo
{
    public static void main(String[] args)
    {
        Checkbox obj = new Checkbox();
        Class curr = obj.getClass();
        Class sup = curr.getSuperclass();
        String curr_name = curr.getName();
        String sup_name = sup.getName();
        System.out.println(curr_name+" extends "+sup_name);
    }
}

```

Interface info : One class can extends only one class, but can implements more than one interface.

```

import java.util.ArrayList;
class InterfacesInfo
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        Class c = al.getClass();
        Class in[ ] = c.getInterfaces();
        System.out.println(c.getName()+" implementing.....");
        for(int i=0 ; i<in.length ; i++)
        {
            System.out.println(in[i].getName());
        }
    }
}

```

Example using User defined interfaces :

```
interface In1{ }
class Ex implements In1{ }
interface In2{ }
interface In3 extends In2{ }
interface In4{ }
class Example extends Ex implements In3,In4 { }
class InterfacesInfo{
    public static void main(String[] args) {
        Example e = new Example();
        Class in[] = e.getClass().getInterfaces();
        for(int i=0 ; i<in.length ; i++){
            System.out.println(in[i].getName());
        }
    }
}
```

Finding Fields Info : Properties of an Object (either static or non-static) referred as fields.

```
import java.lang.reflect.Field;
class FieldsInfo
{
    public static void main(String[] args)
    {
        Thread t = new Thread();
        Class c = t.getClass();
        Field f[] = c.getFields();
        System.out.println(c.getName()+" variables set..");
        for(int i=0 ; i<f.length ; i++)
        {
            System.out.println(f[i]);
        }
    }
}

import java.lang.reflect.Field;
class FieldsInfo
{
    public static void main(String[] args)
    {
        //System s = new System(); //CE : can't be instantiated.
        Class s = java.lang.System.class;
        System.out.println("fields of System class are.....");
        Field f[] = s.getFields();
        for(int i=0 ; i<f.length ; i++)
            System.out.println(f[i]);
    }
}
```

Finding Methods Info :

```
public Method[] getMethods() throws SecurityException
    Returns an array containing Method objects reflecting all the public member
methods of the class or interface represented by this Class object.

public class Example
{
    public void f1(){}
    private void f2(){}
    void f3(){}
    private void f4(){}
}

import java.lang.reflect.Method;
class MethodInfo
{
    public static void main(String[] args)
    {
        Example e = new Example();
        Class c = e.getClass();
        Method m[] = c.getMethods();
        System.out.println(c.getClass()+" containing following methods.....");
        for(int i=0 ; i<m.length ; i++)
        {
            System.out.println("method "+(i+1)+" : "+m[i].getName());
        }
    }
}
```

Loading a class at runtime :

```
/*class Class
{
    public static Class<?> forName(String className) throws ClassNotFoundException
    {
        loads the class by checking the details at runtime
    }
}
*/
class DynamicLoad
{
    public static void main(String[] args)
    {
        try
        {
            Class obj1 = Class.forName("Sample");
            System.out.println("Sample is loaded....");

            Class obj2 = Class.forName("Example");
        }
    }
}
```

```
        System.out.println("Example is loaded....");
    }
    catch (ClassNotFoundException obj)
    {
        System.out.println("Class not found : "+obj.getMessage());
    }
}

class DynamicLoading
{
    public static void main(String[] args)
    {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        System.out.print("Enter one class name : ");
        String name = sc.next();
        try{
            Class.forName(name);
            System.out.println("class is loading successfully.....");
        }
        catch (ClassNotFoundException cn){
            System.out.println("class doesn't exist.....");
        }
    }
}
```

String Handling

- String is a sequence of characters.
- String is a single dimensional character array.
- In Java programming language, strings are objects.
`String str = "Hello world!";`
- Whenever it encounters a string literal in your code, String object will be created with its value in this case, "Hello world!".
- It is also possible to construct the object in different ways using pre-defined constructors.

```
class Test
{
    public static void main(String[] args)
    {
        String s1 = "one";
        String s2 = new String("two");

        char c[] = {'t','h','r','e','e'};
        String s3 = new String(c);

        byte b[] = {65,66,67,68};
        String s4 = new String(b);

        System.out.println("S1 : "+s1+"\nS2 : "+s2+"\nS3 : "+s3+"\nS4 : "+s4);
    }
}
```

String length:

- Finding length of a string is possible in two ways.
- Using `length()` in string class we can find length of the String
- Using Array class “length” property we can find out number of strings inside the array.

```
class StringDemo
{
    public static void main(String args[])
    {
        String palindrome = "Dot saw I was Tod";
        String set[] = {"one", "two", "three"};
        int len1 = palindrome.length();
        int len2 = set.length;
        System.out.println( "Palindrome Length is : " + len1);
        System.out.println( "Number of strings in set : " + len2);
    }
}
```

String concatenation:

- Merging of two strings is nothing but string concatenation.
- We can concatenate two strings in two ways.
- As “+” operator exhibits polymorphism in java, we can concatenate two string using “+” operator.
- By using concat() method also we can perform concatenation operation.

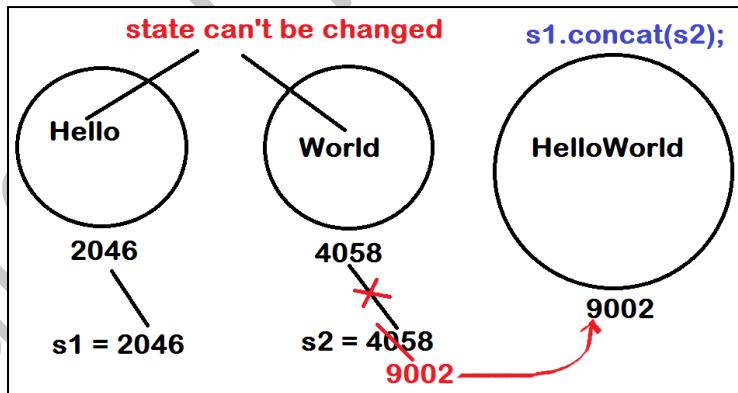
```

/*class String{
    String concat(String str)
    {
        Concatenates the specified string to the end of this string.
    }
}*/

class StringDemo
{
    public static void main(String[] args)
    {
        String s1 = "Hello";
        String s2 = "World";
        System.out.println("Before Merge : \nS1 : "+s1+"\nS2 : "+s2+"\n");

        // s1.concat(s2);
        s2 = s1.concat(s2);
        System.out.println("After Merge : \nS1 : "+s1+"\nS2 : "+s2+"\n");
    }
}

```



What are mutable objects and immutable objects?

- An Immutable object is a kind of object whose state cannot be modified after it is created.
- This is as opposed to a mutable object, which can be modified after it is created.
- In Java, objects are referred by references.
- If an object is said to be immutable, the object reference can be shared.
- For example, Boolean, Byte, Character, Double, Float, Integer, Long, Short, and String are immutable classes in Java, but the class StringBuffer is a mutable object in addition to the immutable String.

- While printing object reference of general objects, it prints address(hexa string).
- While printing Immutable object reference, it prints data(value) inside object.
- Reason, Immutable object always holds a single element.

Question: How to print address of Immutable objects ?

```
/*class Object{
    int hashCode(){
        // logic....
    }
}*/
class StringDemo
{
    public static void main(String[] args)
    {
        String s1 = new String("Hello");
        String s2 = new String("World");
        System.out.println("s1 : "+s1.hashCode());
        System.out.println("s2 : "+s2.hashCode());
    }
}
```

Note : String objects create in String pool area.

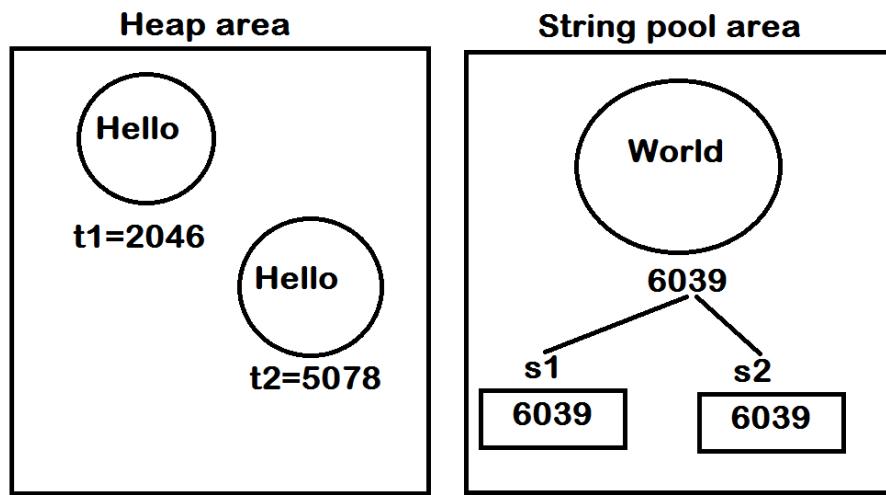
Duplicate objects not allowed in pool area.

```
class Test{
    String s ;
    Test(String s){
        this.s = s ;
    }
}
class StringDemo{
    public static void main(String[] args) {
        Test t1 = new Test("Hello");
        Test t2 = new Test("Hello");

        System.out.println("t1 : "+t1.hashCode());
        System.out.println("t2 : "+t2.hashCode());

        String s1 = new String("World");
        String s2 = new String("World");

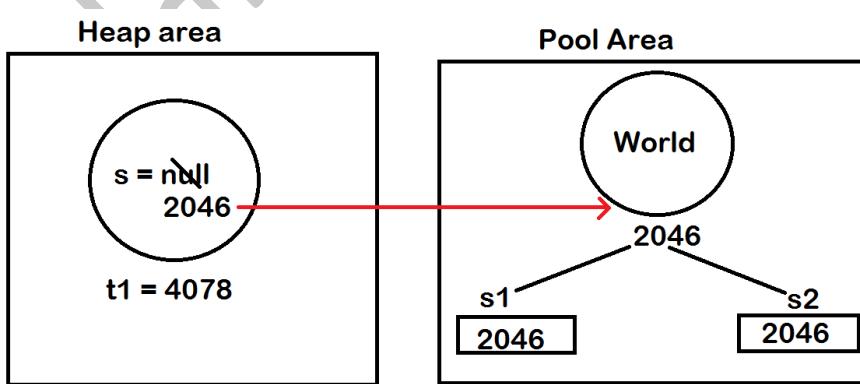
        System.out.println("s1 : "+s1.hashCode());
        System.out.println("s2 : "+s2.hashCode());
    }
}
```



```

class Test
{
    String s;
    Test(String s)
    {
        this.s = s;
    }
}
class StringDemo
{
    public static void main(String[] args)
    {
        String s1 = new String("World");
        String s2 = new String("World");
        Test t1 = new Test("World");

        System.out.println("s1 : "+s1.hashCode());
        System.out.println("s2 : "+s2.hashCode());
        System.out.println("t1.s : "+t1.s.hashCode());
    }
}
  
```



Question : In the above application if any variable(t1.s , s1 or s2) modifies, will it effect remaining object information?

Ans : No, With modified content new object will be created because Strings are immutable.

How to create an immutable class?

1. Class must be public
2. Properties must be final and private
3. Initialization of Object must be using constructor
4. Public setter methods not allowed
5. Public getters allowed to access the information of Immutable object.

```
public final class FinalPersonClass {  
    private final String name;  
    private final int age;  
  
    public FinalPersonClass(final String name, final int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public int getAge() {  
        return age;  
    }  
    public String getName() {  
        return name;  
    }  
}
```

Char at position:

```
public class StringDemo {  
    public static void main(String args[]) {  
        String s = "Naresh technologies";  
        char result = s.charAt(8);  
        System.out.println(result);  
        System.out.printf("%c",s.charAt(7)); //using format specifiers  
    }  
}
```

String comparison:

```
public class StringDemo{  
    public static void main(String args[]) {  
        String str1 = "Strings are immutable";  
        String str2 = "Strings are immutable";  
        String str3 = "Integers are not immutable";  
  
        int result = str1.compareTo( str2 );  
        System.out.println(result);  
    }  
}
```

```

        result = str2.compareTo( str3 );           //using compareTo(Object)
        System.out.println(result);               //returns ascii difference

        result = str3.compareTo((String)str1);   //using compareTo(String)
        System.out.println(result);
    }
}

```

Comparing the strings without considering the case:

```

public class StringDemo {
    public static void main(String args[]){
        String Str1 = new String("hello");
        String Str2 = Str1;
        String Str3 = new String("Hello");
        boolean retVal;

        retVal = Str1.equals( Str2 );
        System.out.println("Returned Value = " + retVal );

        retVal = Str1.equals( Str3 );
        System.out.println("Returned Value = " + retVal );

        retVal = Str1.equalsIgnoreCase( Str3 );
        System.out.println("Returned Value = " + retVal );
    }
}

```

Trim(): (to delete spaces in the beginning and ending of a string)

```

public class StringDemo{
    public static void main(String args[]){
        String s1 = new String("      Welcome to naresh tech      ");
        System.out.println("before trim the length is :" + s1.length());
        String s2=s1.trim();
        System.out.println("Return Value :" + s2);
        System.out.println("after trim the length is :" + s2.length());
    }
}

```

toString() and valueOf() :

```

public class StringDemo{
    public static void main(String args[]){
        Byte b=10;
        String s1=b.toString();           //converts obj to string
        System.out.println(s1);

        int i=100;
        String s2=Integer.toString(i);   //converts primitive to string
    }
}

```

```

        System.out.println(s2);

        //byte res1=String.valueOf(s1);//cannot converts string to primitive
        //int res2=String.valueOf(s2);//it converts String obj to corresponding primitive type
    (so res1 and res2 of type String only here in this example) only
        byte res1=Byte.valueOf(s1);
        int res2=Byte.valueOf(s2);
        System.out.println("\n"+res1+"\n"+res2);
    }
}

```

Difference between String and StringBuffer/StringBuilder in Java

- The main difference between String and StringBuffer/StringBuilder in java is that String object is immutable whereas StringBuffer/StringBuilder objects are mutable.
- Immutable means the value stored in the String object cannot be changed.

If String is immutable then how can I able to change the contents of the object?

- If you made any changes that will not directly reflect on old string. Internally a new String object is created to do the changes.
- So suppose you declare a String object:
`String myString = "Hello";`
- Next, you want to append “Guest” to the same String. What do you do?
`myString = myString + "world";`
- When you print the contents of myString the output will be **“Hello world”**.
- Although we made use of the same object (myString), internally a new object was created in the process.
- So, if you were to does some string operation involving append or trim or some other method call to modify your string object, you would really be creating those many new objects of class String.

How do you make your string operations efficient?

- Using StringBuffer or StringBuilder instead of String class to manipulate strings.
- As StringBuffer/StringBuilder objects are mutable, we can make changes to the value stored in the object.

What's the difference between StringBuffer and StringBuilder?

- StringBuffer and StringBuilder have the same methods with one difference and that's of synchronization.
- StringBuffer is synchronized (which means it is thread safe and hence you can use it when you implement threads for your methods) whereas StringBuilder is not synchronized (which implies it isn't thread safe).
- So, if you aren't going to use threading then use the StringBuilder class as it'll be more efficient than StringBuffer due to the absence of synchronization.

Note: StringBuilder was introduced in Java 1.5 (so if you happen to use versions 1.4 or below you'll have to use StringBuffer)

Code1:

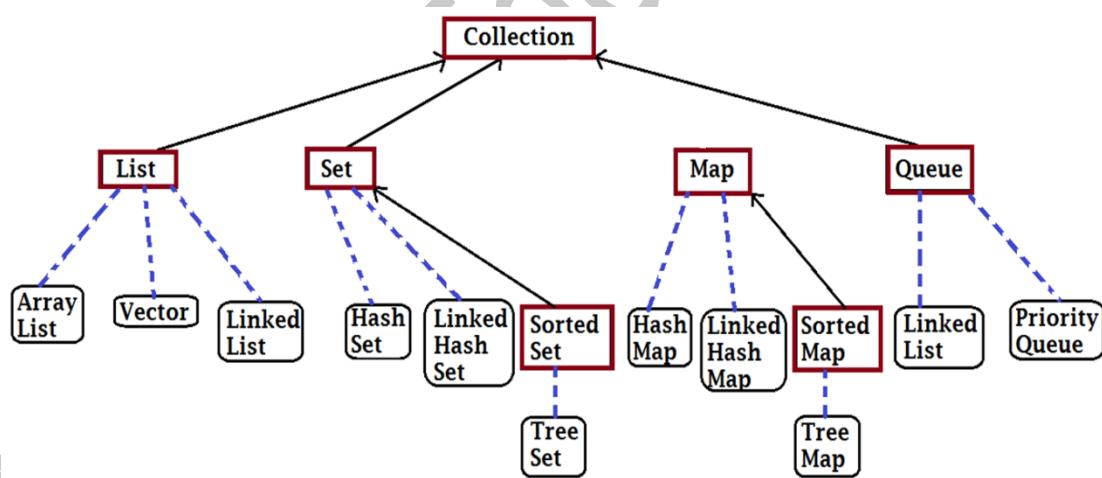
```
class Example {  
    public static void main(String[] args) {  
        String s1="naresh";  
        s1=s1+"tech";  
        System.out.println(s1);  
  
        StringBuffer sb1=new StringBuffer("santosh");  
        //sb1=sb1+"tech";  
        //System.out.println(sb1);  
  
        StringBuffer sb2=new StringBuffer("tech");  
        StringBuffer sb3;  
        sb3=sb1.append(sb2);  
        System.out.println(sb3);  
    }  
}
```

Code2:

```
class Example {  
    public static void main(String[] args) {  
        String s1="naresh";  
        s1=s1+"tech";  
        System.out.println(s1);  
  
        StringBuffer sb1=new StringBuffer("santosh");  
        //sb1=sb1+"tech";  
        //System.out.println(sb1);  
  
        StringBuffer sb2=new StringBuffer("tech");  
        //StringBuffer sb3;  
        sb1=sb1.append(sb2);  
        System.out.println(sb1);  
    }  
  
    class Example {  
        public static void main(String[] args) {  
            StringBuilder sb1=new StringBuilder("naresh tech");  
            System.out.println(sb1);  
            sb1.deleteCharAt(2);  
            sb1.insert(2,'gm');  
            System.out.println(sb1);  
        }  
    }  
}
```

Collection Framework

- For every computer application maintaining and managing the data is mandatory.
- While storing the data into secondary storage device like database, if we arrange the data in a particular structure, we can access effectively at later time.
- To structure the data number of algorithms were proposed.
- Some of the examples are :
 1. **stack (Last In First Out)**
 2. **queue (First In First Out)**
 3. **arrays (Stores info in sequential order)**
 4. **linked lists(Stores info in the form of nodes)**
 5. **trees**
 6. **graphs.**
- In other languages like C and C++, we need to implement proposed algorithms to structure the data in any computer application.
- But Java API provides interfaces and classes to structure the data which are implementations of those algorithms.
- Mainly used interfaces as given below.
- Collection is the super interface of all.
- No direct implementation of collection interface.



Array:

- They are fixed in size.
- With respect to memory arrays are not preferable.
- Arrays shows very good Performance(index based)
- Can hold only homogeneous data elements.
- Arrays can hold both primitive data types and objects
- They do not have any underlying data structures algorithms.
- Arrays can hold duplicate elements.

```

class ArrayDemo
{
    public static void main(String[] args)
    {
        //int arr[5]; //not allowed

        int arr[ ] = new int[5]; //memory will be allocated using "new" operator at runtime
        arr[0] = 10 ; //primitive value
        arr[1] = new Integer(20) ; //using wrapper classes, we are storing Integer object
        arr[2] = 10 ;

        //arr[3] = 34.56 ; //CE : allows only homogenous elements
        arr[7] = 40 ; //RE : out of bounds.....
```

Java Technologies

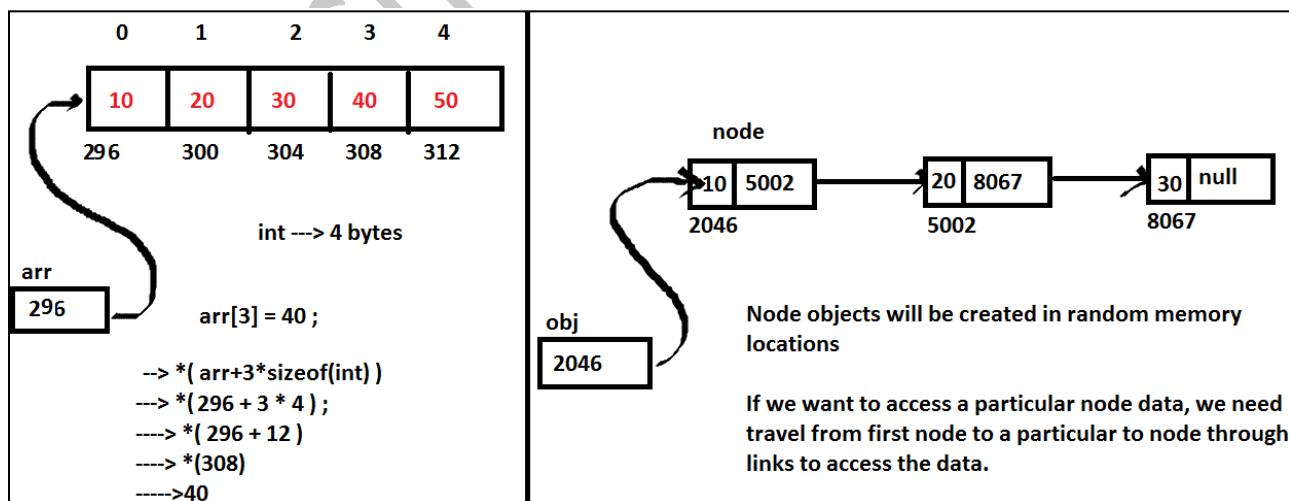
```

        System.out.println("Array elements are....");
        for(int i=0 ; i<=arr.length ; i++)
        {
            System.out.println(arr[i]);
        }
    }
}

```

Collections :

- Grow able in nature.
- With respect to memory, collections are recommended.
- It shows poor Performance(All the collections are not index based, for example Linked List).
- Can hold both homogenous and Heterogeneous.
- They have underlying data structure.
- Collections hold only Objects.



The Collections Framework consists of three parts

- Algorithms (rules).

- Interfaces (abstract data types).
- Implementations (concrete versions of these interfaces).
- All implementations are Serializable and Cloneable.
- All implementations support having null elements.

Interface	Ordered	Allow duplicates
Collection	yes/no	yes/no
List Queue	yes	yes
Set Map	no	no
SortedSet SortedMap	yes	no

List interface :

ArrayList :

- 1) available in java.util package.
- 2) since jdk 1.2
- 3) ordered
- 4) allows duplicates.
- 5) initial capacity ($n=10$)
- 5) incrementing capacity ($3n/2$) that means it increases half of its previous capacity.
- 6) ArrayList is not synchronized by default but can be synchronized explicitly as follows.....
`Collections.synchronizedList(new ArrayList(...));`

```
import java.util.ArrayList;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(new Integer(10)); //boxing
        al.add(20); //auto boxing (since jdk 1.5)
        al.add(10); //duplicates allowed
        al.add("amar");//heterogenous data allowed.
        System.out.println(al);
    }
}
```

Question: why the compiler giving warnings when we compile the above program?

- In arrays, it is possible to set restrictions about what type of data is allowed to store(homogenous), but it is not possible in collections upto jdk 1.4.
- In jdk 1.5, they introduced the concept of Generics by which we can set restrictions on Collection Object.

```
import java.util.ArrayList;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(new Integer(10));
        al.add(new Float(34.56)); //CE : not allowed
    }
}
```

Operations on Array List using pre-defined methods:

```
import java.util.ArrayList;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList al1 = new ArrayList();
        for(int i=10 ; i<=50 ; i+=10)
        {
            al1.add(new Integer(i)); //Appends the element
        }
        System.out.println(al1);
        al1.add(2,100); //inserts the element @ specified index number
        System.out.println("al1 elements : "+al1);
        //ArrayList al2 = al1.clone(); //CE :
        ArrayList al2 = (ArrayList)al1.clone();
        System.out.println("al2 cloned elements : "+al2);
        al1.addAll(4,al2);
        System.out.println("al1 elements : "+al1);
    }
}
```

- ArrayList elements we can print directly by printing Object reference of that List, because internal iterator will access the elements.
- Using internal elements only we can print the elements but we can't process those elements.
- To process elements number of explicit Iterators are available in the Collection API.
 1. **For-each loop**
 2. **Enumeration**
 3. **Iterator**

for-each loop (enhanced for loop) :

- used to iterate Array or Collection.
- since jdk 1.5
- can iterate the elements without taking the information of source & destination bounds.
- can move only in forward direction.
- can access element by element.

syntax :

```
for(<data_type> <var> : <array>or<collection>)
{
    Processing logic...
}
```

Example:

```
class ForEachExample
{
    public static void main(String[] args)
    {
        int arr[ ] = {10,20,30,40,50};
        System.out.println("Array elements are : ");
        for(int ele : arr)
        {
            System.out.println(ele);
        }
    }
}
```

Program:

```
class ForEachLoop
{
    public static void main(String[] args)
    {
        int arr[ ] = {10,20,30,40,50};
        int i=0 ;
        for( int ele : arr )
        {
            arr[i] = ele + 100 ;
            i++;
        }
        System.out.println("Array elements are : ");
        for( int ele : arr ){
            System.out.println(ele);
        }
    }
}
```

Processing Array elements using for-each loop:

```
import java.util.ArrayList;
import java.util.Random;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
        Random r = new Random();
        for(int i=1 ; i<=5 ; i++)
        {
            al.add(new Integer(r.nextInt(100)));
        }
        System.out.println("elements are : ");
        for(Integer i : al)
        {
            System.out.println(i);
        }
    }
}
```

protected void removeRange(int fromIndex, int toIndex)

Removes from this list all of the elements whose index is between fromIndex(inclusive) and toIndex(exclusive). Shifts any succeeding elements to the left (reduces their index).

```
import java.util.ArrayList;
class ArrayListDemo extends ArrayList
{
    public static void main(String[] args)
    {
        ArrayListDemo al = new ArrayListDemo();
        for(int i=10 ; i<=100 ; i+=10)
        {
            al.add(new Integer(i));
        }
        System.out.println(al);
        al.removeRange(4,8); //removes the elements from index 4 to 7
        System.out.println(al);
    }
}
```

public E remove(int index)

Removes the element at the specified position in this list.

```
/*
public E remove(int index)
{
    import java.util.ArrayList;
    class ArrayListDemo
```

```

public static void main(String[] args)
{
    ArrayList al = new ArrayList();
    for(int i=10 ; i<=30 ; i+=10)
    {
        al.add(new Integer(i));
        al.add(new Integer(2));
    }
    System.out.println(al);
    al.remove(2); //deletes element @ index 2
    System.out.println(al);
}

/*
public boolean remove(Object o)
    Removes the first occurrence of the specified element from this list, if it is present
*/
import java.util.ArrayList;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        for(int i=10 ; i<=30 ; i+=10)
        {
            al.add(new Integer(i));
            al.add(new Integer(2));
        }
        System.out.println(al);
        al.remove(new Integer(2)); //deletes first occurrence of 2
        System.out.println(al);
    }
}

```

Different operations on ArrayList collection object :

```

import java.util.ArrayList;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList al1 = new ArrayList();
        al1.add(10); //Appends the element
        al1.add(30);
        al1.add(20);
        System.out.println(al1);
        al1.add(1,100); //inserts element @ index pos
    }
}

```

```

        System.out.println(al1);

        ArrayList al2 = new ArrayList();
        al2.add(30);
        al2.add(10);

        al1.addAll(al2); //Appends al2 elements to al1
        System.out.println(al1);

        al1.addAll(2,al2);//inserts elements of al2 @index pos of al1
        System.out.println(al1);

        System.out.println("ele @ pos 3 : "+al1.get(3));
        System.out.println("no of elements : "+al1.size());

        al1.set(4 , 200);//replaces element @pos4 with 200
        System.out.println(al1);

        al1.clear();//clears the list
        System.out.println(al1);

        if(al1.isEmpty())
            System.out.println("al1 list is empty");
        else
            System.out.println("al1 contains elements");
    }
}

```

Find out the output:

```

import java.util.ArrayList;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList list = new ArrayList();
        for(int i=10 ; i<=100 ; i=i+10)
        {
            list.add(new Integer(i));
        }
        for(int i=0 ; i<list.size() ; i++)
        {
            list.remove(i);
        }
        System.out.println("final list : "+list);
    }
}

```

Vector :

- 1) available in java.util package.
- 2) since jdk 1.0
- 3) ordered
- 4) allows duplicates
- 5) initial capacity n=10
- 6) incrementing capacity (2n)
- 7) Vector is synchronized by default.

Note: In ArrayList it is not possible to check the internal capacity(no such method). But in Vector we can check internal capacity and incrementing capacity using pre-defined method.

Tip : Only the collection classes which are from jdk 1.0 are synchronized by default. Most of the Collection classes are introduced from jdk 1.2 which are not synchronized by default but can be synchronized explicitly user pre-defined static functions available in Collections class for example.....

```
Collections.synchronizedList();
Collections.synchronizedSet();
Collections.synchronizedMap();
Collections.synchronizedSortedSet();
Collections.synchronizedSortedMap();
```

/* public Enumeration<E> elements()

Returns an enumeration of the components of this vector.

boolean hasMoreElements()

Tests if this enumeration contains more elements.

Object nextElement()

Returns the next element of this enumeration if this enumeration object */

```
import java.util.Vector;
import java.util.Enumeration;
class VectorDemo
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        System.out.println("initial size : "+v.size());//returns number of elements
        System.out.println("initial capacity : "+v.capacity());
        for(int i=10 ; i<=100 ; i+=10)
        {
            v.add(new Integer(i));
        }
        System.out.println("after 10 insertions size : "+v.size());
```

```

        System.out.println("after 10 insertions capacity : "+v.capacity());

        v.add(110);
        System.out.println("after 11 insertions size : "+v.size());
        System.out.println("after 11 insertions capacity : "+v.capacity());

        Enumeration e = v.elements();
        while(e.hasMoreElements())
        {
            System.out.println(e.nextElement());
        }
    }

import java.util.Vector;
import java.util.Enumeration; //interface
class VectorDemo
{
    public static void main(String[] args)
    {
        Vector v = new Vector(3,2);
        System.out.println("initial capacity : "+v.capacity());
        v.add(10);
        v.add(20);
        v.add(30);
        v.add(40);
        System.out.println("after 4 insertions, capacity : "+v.capacity());

        Enumeration e = v.elements();
        System.out.println("Vector elements are : ");
        while(e.hasMoreElements())
        {
            System.out.println(e.nextElement());
        }
    }
}

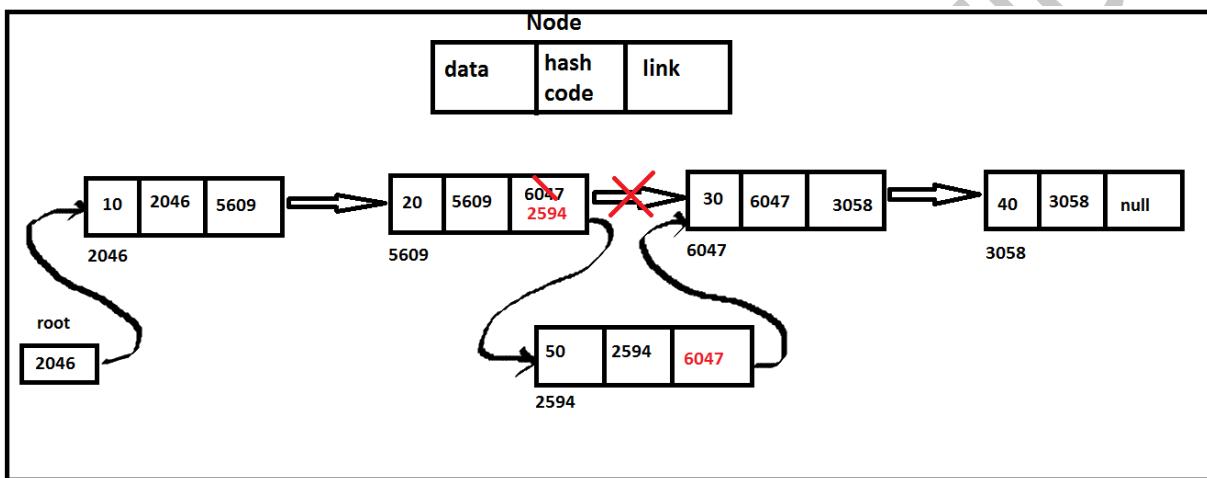
```

LinkedList:

- 1) Available in util package.
- 2) since jdk 1.2
- 3) ordered
- 4) allows duplicates
- 5) initial capacity n=0
- 6) incrementing capacity ++n
- 7) not synchronized by default.

LinkedList representation:

- 1) In the linked list, elements are stored in the form of nodes.
- 2) Each node having three fields....
 - a) data field
 - b) hashCode field
 - c) address of next node.
- 3) Linked list occupies much memory than arraylist.
- 4) creation time is higher than ArrayList.
- 5) **insertions and deletions are faster.**
- 6) searching is slow(not index based).
- 7) Using hashCode linked list maintains the order of elements.



```
import java.util.ArrayList ;
import java.util.LinkedList ;
class ConstructionTime
{
    private static Object arr[ ];
    static
    {
        arr = new Object[1000000];
        for(int i=0 ; i<arr.length ; i++)
        {
            arr[i] = new Object();
        }
    }
    public static void main(String[] args)
    {
        long start, end ;

        ArrayList list1 = new ArrayList();
        start = System.currentTimeMillis();
        for(Object ele : arr)
        {
    }
```

```

        list1.add(ele);
    }
    end = System.currentTimeMillis();
    System.out.println("ArrayList construction time : "+(end-start));

    LinkedList list2 = new LinkedList();
    start = System.currentTimeMillis();
    for(Object ele : arr)
    {
        list2.add(ele);
    }
    end = System.currentTimeMillis();
    System.out.println("LinkedList construction time : "+(end-start));
}

```

Set interface: Implementations are.....

1. HashSet
2. LinkedHashSet
3. TreeSet

Note : HashSet & LinkedHashSet stores the data into collection Object according to HashTable algorithm.

How the elements store into HashTable ?

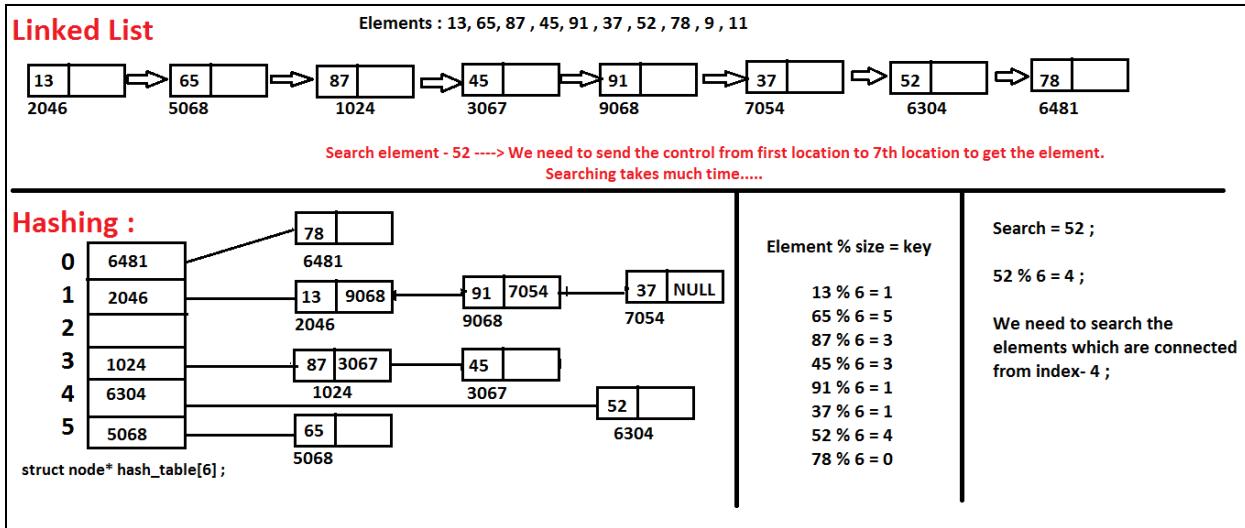
- Every Hash table has a fixed size by default.
- The size may varies depends on number of elements stored in the Collection.
- Hash table implementation is bit advanced to linked list representation.
- In Hashtable also, elements will be stored in the form of nodes only.
- Searching of element is faster than Linkedlist in Hashtable.

structure format :

- Every element in the Hashtable stores according to key element and size of Hashtable only.
- But in Set interface, elements uses only Hashtable size to be stored but not key elements.

HashSet :

- available in util package.
 - since from jdk 1.2
 - not ordered
 - not allowed duplicates.
 - initial capacity is 16
 - default load factor is 0.75
 - HashSet is not Synchronized by default, can be synchronized explicitly as follows...
- Collections.synchronizedSet(new HashSet(...));**



```

import java.util.Random;
class RandomNumbers
{
    public static void main(String[] args)
    {
        Random r = new Random();

        int x = r.nextInt(); // generate in integer limits
        System.out.println("x value : "+x);

        int y = r.nextInt(100); // generate in 0-100
        System.out.println("y value : "+y);
    }
}

import java.util.*;
class HashSetDemo
{
    public static void main(String[] args)
    {
        ArrayList list = new ArrayList();
        Random r = new Random();
        for(int i=1; i<=10; i++)
        {
            int ele = r.nextInt(5);
            list.add(new Integer(ele));
        }
        System.out.println("List : "+list); // allow duplicates

        HashSet set= new HashSet(list); //Converting List --> Set
        System.out.println("Set : "+set); // doesn't allow duplicates
    }
}

```

```

    }
}

```

Note : Some of the implementations rejects the element when it is duplicated in the Collection.
for example... Set interface rejects the duplicate elements.

Map interface replaces the duplicate elements.

```

import java.util.*;
class HashSetDemo
{
    public static void main(String[] args)
    {
        HashSet set = new HashSet();
        set.add("amar");

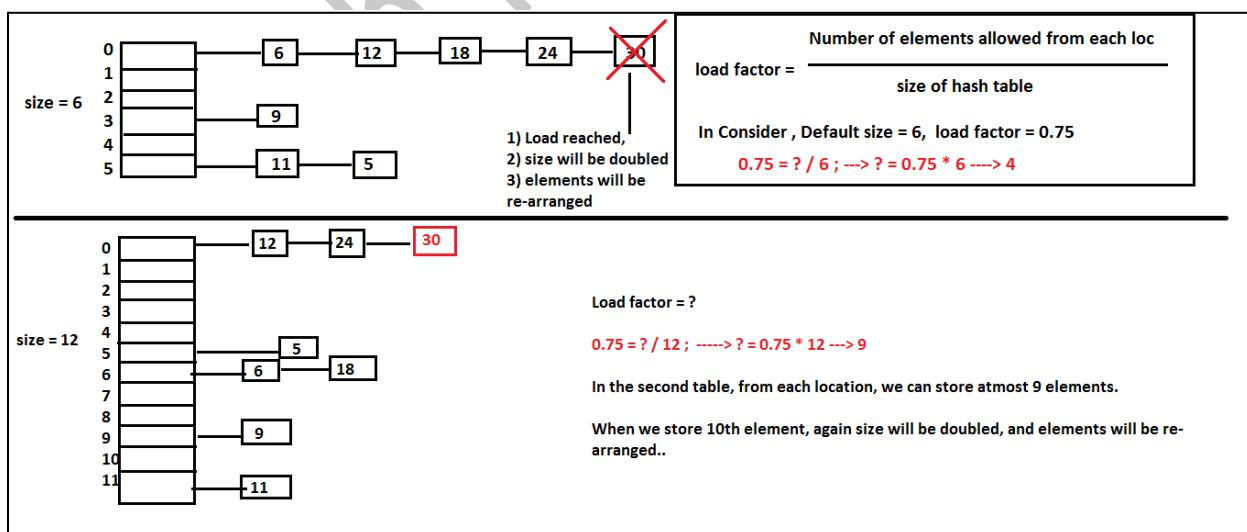
        if(set.add("amar"))
            System.out.println("duplicates are replaced");
        else
            System.out.println("duplicates are rejected");
    }
}

```

load factor:

- The amount of load(no of elements can add from each location of hash table).
- Once load is reached, the size of hash table will be increased implicitly(probably doubled) and all the elements are re-arranged automatically.

Load factor = number of elements / size of hash table



LinkedHashSet :

- The only difference between HashSet and LinkedHashSet is, HashSet does not maintain the insertion order of elements whereas LinkedHashSet does it, using hashCode.
- LinkedHashSet since from jdk 1.4

Iterator :

- It is an interface having pre-defined functions used to iterate Collection Object to process the elements.
- Much working like Enumeration.
- Iterator introduced in jdk 1.2 along with new implementations of Collection interface.
- To get Iterator object, we need to call iterator() method on any Collection instance.

Iterator <identifier> = Collection_Object.iterator();

```
import java.util.*;
class SetDemo
{
    public static void main(String[] args)
    {
        // HashSet set = new HashSet(); // No insertion order
        LinkedHashSet set = new LinkedHashSet(); // Maintain insertion order

        for(int i=10 ; i<=50 ; i+=10)
        {
            set.add(new Integer(i));
            set.add(new Integer(20));
        }

        System.out.println("Elements are : ");
        Iterator itr = set.iterator();
        while(itr.hasNext())
        {
            Integer i = (Integer)itr.next();
            System.out.println(i);
        }
    }
}
```

TreeSet:

- TreeSet maintains sorted order of inserted elements.
- It will arrange the elements in ascending order using balanced binary search tree algorithm.
- since from jdk 1.2

```
import java.util.*;
class SetDemo
{
    public static void main(String[] args)
    {
        Random r = new Random();
    }
}
```

```

// HashSet hs = new HashSet();
// LinkedHashSet hs = new LinkedHashSet();
TreeSet hs = new TreeSet();

for(int i=1; i<=5 ; i++)
{
    int ele = r.nextInt(100);
    hs.add(new Integer(ele));
    System.out.println("Element - "+i+" : "+ele);
}

// System.out.println("HashSet : "+hs); // Doesn't maintain insertion order
// System.out.println("LinkedHashSet : "+hs); // Maintain insertion order
System.out.println("TreeSet : "+hs); // Maintain sorted order
}
}

```

Map interface: The implementations are

1. HashMap
2. LinkedHashMap
3. TreeMap

Note : HashMap and LinkedHashMap stores the elements according to Hashtable algorithm only.

- In Map interface, elements will be stored using keys.
- Keys are always unique(duplicates not allowed).
- Keys are also objects.
- Duplicate elements are allowed with unique keys.
- "null" key is allowed in HashMap and LinkedHashMap.
- "null" object is also allowed.
- It is allowed to store the elements with heterogenous keys(but unique).
- Duplicates will be replaced but not rejected.

HashMap :

- available in util package
- since from jdk 1.2
- not ordered
- duplicates allowed with unique keys
- default capacity is 16
- default load factor is 0.75
- HashMap is not synchronized by default. but can be synchronized explicitly as follows.

Collections.synchronizedMap(new HashMap(...));

```

import java.util.*;
class SetDemo

```

By Srinivas (C/DS/Java trainer)

Page 280

```

{
    public static void main(String[] args)
    {
        HashSet hs = new HashSet();
        hs.add("one");
        if(hs.add("one"))
            System.out.println("Duplicate element is replaced in Set");
        else
            System.out.println("Duplicate element is rejected in Set");
    }
}

import java.util.HashMap ;
class MapDemo
{
    public static void main(String[] args)
    {
        HashMap map = new HashMap();

        map.put(10 , "One");
        System.out.println("Map : "+map);

        map.put(10 , "Two"); // replace existing element
        System.out.println("Map : "+map);
    }
}

import java.util.HashMap ;
class MapDemo
{
    public static void main(String[] args)
    {
        HashMap map = new HashMap();

        map.put(10 , "One");
        map.put(20 , "One");
        map.put(null , "Two");
        map.put("key" , "Three");
        map.put(34.56 , "Four");
        System.out.println("Map : "+map);
    }
}

```

Note : we can apply generics on map collection object. Once we set restrictions on data, all the methods logic will be converted according to generics type.

```

import java.util.HashMap ;
class MapDemo
By Srinivas (C/DS/Java trainer)

```

```

{
    public static void main(String[] args)
    {
        HashMap<Integer , String> map = new HashMap<Integer, String>();
        map.put(10 , "One");
        map.put(34.56 , "Four"); // Error : Only Integer keys allowed
        System.out.println("Map : "+map);
    }
}

```

Note : The only difference between LinkedHashMap and HashMap is, HashMap doesn't maintain insertion order of elements where as LinkedHashMap does it. LinkedHashMap since jdk 1.4

```

import java.util.*;
class MapDemo
{
    public static void main(String[] args)
    {
        HashMap hm = new HashMap();
        hm.put(26, "one");
        hm.put(26, "two"); //"one" is replaced with "two"
        hm.put(16, "two"); //duplicate elements allowed
        hm.put(null,"three"); //null key allowed
        hm.put(56,null); //null element allowed
        hm.put("56","four"); //heterogenous keys allowed
        System.out.println(hm);
    }
}

```

TreeMap :

- TreeMap maintains ascending order of keys.
- TreeMap does it through Balanced binary trees.

```

import java.util.*;
class MapDemo
{
    public static void main(String[] args)
    {
        TreeMap tm = new TreeMap();
        tm.put(26, "one");
        tm.put(16, "two");
        //tm.put(null,"three"); //not allowed
        //tm.put("str","four"); //not allowed
        System.out.println(tm);
    }
}

```

```

import java.util.*;
class MapDemo
{
    public static void main(String[] args)
    {
        String s[ ] = { "One", "Two", "Three", "Four" };
        Random r = new Random();

        //HashMap map = new HashMap();
        //LinkedHashMap map = new LinkedHashMap();
        TreeMap map = new TreeMap();

        for(int i=0 ; i<s.length ; i++)
        {
            int key = r.nextInt(100);
            String ele = s[i];
            map.put(key , ele);
            System.out.println(key + " : " + ele);
        }

        // System.out.println("HashMap : "+map); //Doesn't maintain insertion order

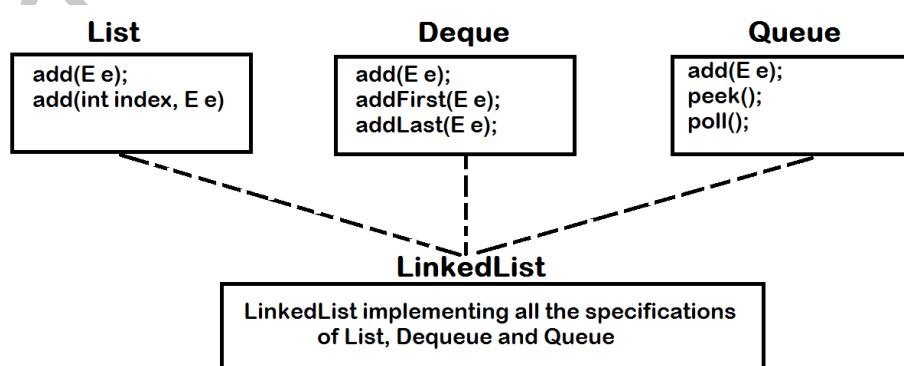
        // System.out.println("LinkedHashMap : "+map); //Maintain insertion order

        System.out.println("TreeMap : "+map); //Maintain ascending order of keys
    }
}

```

Queue interface : implementations are

1. LinkedList
 2. PriorityQueue
- Queue based on the rule First In First Out(FIFO)
 - LinkedList maintains insertion order of elements as it is implementing Queue interface.



If we create object for LinkedList and with LinkedList reference variable, we can access complete functionality of Queue, List and Deque.

```
import java.util.*;
class LinkedListDemo
{
    public static void main(String[] args)
    {
        LinkedList list = new LinkedList();

        list.add(100);
        list.add(200);
        System.out.println("List : "+list);

        list.addFirst(111);
        list.addLast(222);
        System.out.println("List : "+list);

        System.out.println("Peek : "+list.peek()); //Returns top element but not remove
        System.out.println("After peek List : "+list);

        System.out.println("Poll : "+list.poll()); //Returns top element & remove
        System.out.println("After poll List : "+list);
    }
}
```

Note : If we collect LinkedList object address into a particular implemented interface type, we cannot access all the interfaces functionality.

```
import java.util.*;
class LinkedListDemo
{
    public static void main(String[] args)
    {
        Queue q = new LinkedList();
        q.add(100);
        q.add(200);
        System.out.println("Queue : "+q);
        System.out.println("Peek : "+q.peek());

        q.addFirst(111); // Error : not specified in Queue
        System.out.println("Queue : "+q);
    }
}
```

PriorityQueue :

- Maintains insertion order but returns the elements in the ascending order(smallest element first).
- PriorityQueue top element is always the smallest element in the queue.

- available in util package.
- since jdk 1.5
- ordered
- allows duplicates.
- initial capacity is 11.

```

import java.util.*;
class PriorityQueueDemo
{
    public static void main(String[] args)
    {
        Random r = new Random();
        PriorityQueue q = new PriorityQueue();
        for(int i=1 ; i<=6 ; i++)
        {
            q.add(new Integer(r.nextInt(100)));
        }
        System.out.println("Elements : "+q);
    }
}

/*
public Object peek()
    Retrieves but does not remove
    on success it returns head element
    on failure it returns "null" element

public Object poll()
    Retrieves and removes .
    on success it returns next ascending element.
    on failure it returns "null" element.
*/
import java.util.*;
class PriorityQueueDemo
{
    public static void main(String[] args)
    {
        Random r = new Random();
        PriorityQueue q = new PriorityQueue();
        for(int i=1 ; i<=6 ; i++)
        {
            q.add(new Integer(r.nextInt(100)));
        }
        System.out.println("Queue : "+q);
        for(int i=1 ; i<=6 ; i++)
        {
    }

```

```

        System.out.println("Poll : "+q.poll());
        System.out.println("Queue : "+q);
    }
}

import java.util.*;
class QueueDemo{
    public static void main(String[] args){
        LinkedList q = new LinkedList();
        Random r = new Random();
        for(int i=1 ; i<=5 ; i++){
            q.add(new Integer(r.nextInt(50)));
        }
        q.addFirst(100);
        q.addLast(200);
        System.out.println(q);

        Iterator itr = q.descendingIterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}

import java.util.*;
class LinkedListDemo{
    public static void main(String[] args){
        LinkedList l = new LinkedList();
        for(int i=10 ; i<=100 ; i+=10){
            l.add(new Integer(i));
        }
        System.out.println("List : "+l);
        for(int i=0 ; i<l.size() ; i++){
            System.out.println("Poll : "+l.poll());
        }
    }
}

```

Applets and AWT

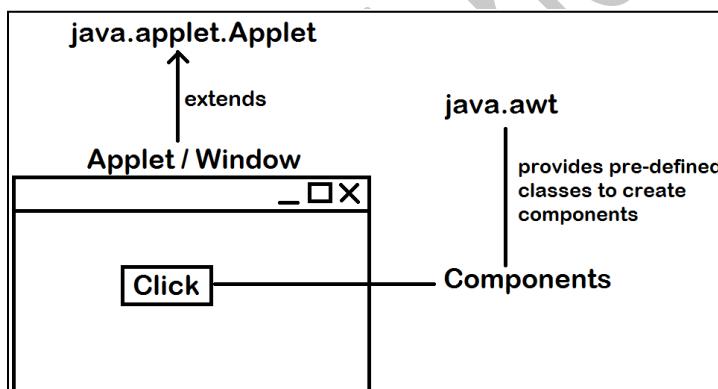
- Applet allows java programmer to implement GUI programming.
- "applet" is a package in java API.
- "Applet" is a pre-defined class provides the functionality to user-defined Applet classes.
- "Applet" is a window where we can arrange all the components to implement GUI.
- Every User-defined Applet class must extends pre-defined Applet.

```
class MyApplet extends java.applet.Applet
{
    //user-defined applet class.
}
```

AWT (Abstract Window Toolkit) :

Pre-defined(Abstract) classes and interfaces to implement Applets(Window programming) available as a Toolkit. for example...

Button, CheckBox, Paint, Graphics.....



First Applet Program :

- Applets are defined to be embedded into HTML pages.
- Applets can be run by the browser interpreter along with HTML tags.
- Every browser having an interpreter to execute HTML code.
- HTML code will not be compiled, it will be executed directly by the interpreter.
- Hence it is hard to find if error has encountered.
- Interpreter produces un-expected output page if any error in the HTML code.
- JVM cannot execute applet program, hence no need to define main() method inside the applet class.
- public void paint() method is the pre-defined method in Applet class.
- Graphics class which is the pre-defined class available in java.awt package.

```
import java.applet.Applet;
import java.awt.*;
public class FirstApplet extends Applet
{
```

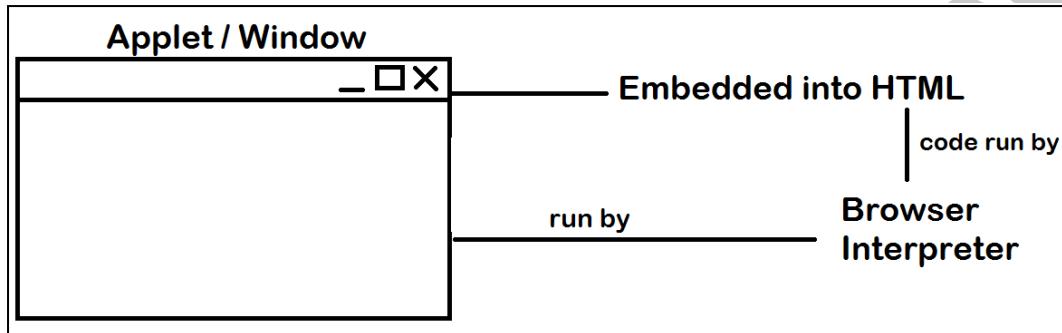
By Srinivas (C/DS/Java trainer)

Page 287

```

public void paint(Graphics g)
{
    g.drawString("Hello world", 200,200);
}
/*
<applet code = "FirstApplet.class" width="600" height = "600">
</applet>
*/

```



Running Applets from HTML pages :

```

<HTML>
    <HEAD>
        <title>My Applet Program</title>
    </HEAD>

    <BODY>
        <applet code = "FirstApplet.class" width="600" height = "600">
        </applet>
    </BODY>
</HTML>

```

Applet Life cycle :

- Some of the pre-defined functions of Applet executes implicitly which can describe the flow of Applet Life cycle.
- CUI based java application execution starts from main() function which is implicitly invokes my JVM.
- GUI based java application will be invoked either by "appletviewer" or "browser interpreter".
- When applet execution starts, it will instantiated implicitly.
- At the time of instantiation, constructor must be invoked.
- Hence the only way to know whether the Applet has been instantiated or not by defining a constructor in the Applet class.
- Members participate in Applet Life cycle :
 - 1) Constructor()
 - 2) init()
 - 3) start()
 - 4) paint()

```

5) stop()
6) destroy()

import java.applet.Applet;
import java.awt.*;
public class FirstApplet extends Applet
{
    public FirstApplet()
    {
        System.out.println("Applet is instantiated....");
    }
    public void paint(Graphics g)
    {
        g.drawString("Hello world", 200,200);
    }
    public void init()
    {
        System.out.println("Applet is initialized....");
    }
    public void start()
    {
        System.out.println("Control enter into Applet....");
    }
    public void stop()
    {
        System.out.println("Control goes off the Applet...");
    }
    public void destroy()
    {
        System.out.println("Resource released.....");
    }
}
/*
<applet code = "FirstApplet.class" width="600" height = "600">
</applet>
*/

```

Component:

- The Object which is displayed on the Applet window is called component.examples....
Buttons, Textbox, CheckBox, RadioButton....

Note : All the component classes available in java.awt package only.

Layout :

- Arrangement of Components by setting Bounds on the window.
- Java API has many pre-defined layout classes to arrange the components.
 - 1) Flow Layout

- 2) Grid Layout
- 3) Border Layout
- 4) Box layout....

Note : By default Applet arranges the components using FlowLayout structure.

```

import java.applet.Applet;
import java.awt.*;
public class Flow extends Applet
{
    public void init()
    {
        //this.setLayout(new FlowLayout(FlowLayout.LEFT));
        for(int i=1; i<=5 ; i++)
        {
            Button b = new Button("BUTTON#"+i);
            this.add(b);
        }
    }
}
/*
<applet code = "Flow.class" width="600" height = "600">
</applet>
*/
import java.applet.Applet;
import java.awt.*;
public class Buttons extends Applet
{
    Button b1,b2;
    public void init(){
        this.setLayout(null);
        this.b1 = new Button("BUTTON#1");
        this.b2 = new Button("BUTTON#2");
        b1.setBounds(150,200,150,50);
        b2.setBounds(300,200,150,50);
        this.add(b1);
        this.add(b2);
    }
}
/*
<applet code = "Buttons.class" width="600" height = "600">
</applet>
*/

```

Event Listeners:

- An Event is an operation(action) performed on a component of Applet.
- When action performed, Component fires one event.

- Listener class can collect event information and reacts to the event according to defined logic.
- "Listener" is a pre-defined interface of all the sub Listeners which are....
 - 1) ActionListener
 - 2) TextListener
 - 3) MouseListener
 - 4) MouseMotionListener
 - 5) FocusListener
- When we create Buttons, these cannot be reacted to Events which are performed.
- ActionListener interface must be implemented to write EventListener logic for Buttons.

```

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class ButtonListener extends Applet implements ActionListener
{
    Button b1,b2;
    public void init()
    {
        this.setLayout(null);
        this.b1 = new Button("BUTTON#1");
        this.b2 = new Button("BUTTON#2");
        b1.setBounds(150,200,150,50);
        b2.setBounds(300,200,150,50);
        this.add(b1);
        this.add(b2);
        b1.addActionListener(this);
        b2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        if(ae.getSource() == b1)
            System.out.println("you clicked button1");
        else
            System.out.println("you clicked button2");
    }
}
/*
<applet code = "ButtonListener.class" width="600" height = "600">
</applet>
*/
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class RadioButtons extends Applet implements ActionListener

```

```

{
    Button b;
    TextField t;
    CheckboxGroup rg;
    Checkbox r1,r2,r3;
    public void init()
    {
        setLayout(new FlowLayout());
        b = new Button("CLICK");
        t = new TextField("Type here Something",35);
        rg = new CheckboxGroup();
        r1 = new Checkbox("Red", rg , false);
        r2 = new Checkbox("Blue", rg , true);
        r3 = new Checkbox("Green", rg , false);
        add(r1);
        add(r2);
        add(r3);
        add(t);
        add(b);
        b.addActionListener(this);
    }
    public void paint(Graphics g) {
        if (r1.getState())
            g.setColor(Color.red);
        else if (r2.getState())
            g.setColor(Color.blue);
        else
            g.setColor(Color.green);
        g.drawString(t.getText(),50,200);
    }
    public void actionPerformed(ActionEvent evt) {
        repaint();
    }
}
/*
<applet code = "RadioButtons.class" width="400" height="400">
</applet>
*/

```

Frames :

- Applets are designed to embed with HTML pages and these can be run by Browser Interpreter.
- Frames are designed to implements Standalone GUI java applications run by JVM directly.
- main() is allowed while implementing Frames.

import java.applet.Applet;

```

import java.awt.*;
import java.awt.event.*;
public class RadioButton extends Frame
{
    Checkbox r1,r2;
    CheckboxGroup rg;
    TextField t;
    public RadioButton(String name)
    {
        this.setTitle(name);
        this.setSize(600,600);
        this.setLayout(new FlowLayout());
        rg = new CheckboxGroup();
        r1 = new Checkbox("RED",rg,false);
        r2 = new Checkbox("BLUE",rg,true);
        t = new TextField("A TextField",20);
        this.add(r1);
        this.add(r2);
        this.add(t);
    }
    public static void main(String args[])
    {
        RadioButton r = new RadioButton("MY Frame");
        r.show();
    }
}

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Calculator extends Frame implements ActionListener
{
    Label l1,l2,l3,l4;
    Button b1,b2,b3,b4,b5;
    TextField t1,t2,t3;
    public Calculator(){
        setLayout(null);
        setSize(600,600);
        l1 = new Label("Enter no1 : ");
        l2 = new Label("Enter no2 : ");
        l3 = new Label("Result : ");
        l4 = new Label("");
        t1 = new TextField(" ", 10);
        t2 = new TextField(" ", 10);
        t3 = new TextField(" ", 10);
        b1 = new Button("ADD");
    }
}

```

```
b2 = new Button("SUB");
b3 = new Button("MUL");
b4 = new Button("DIV");
b5 = new Button("EXIT");

l1.setBounds(100,100,150,50);
t1.setBounds(350,100,150,50);

l2.setBounds(100,200,150,50);
t2.setBounds(350,200,150,50);

l3.setBounds(100,300,150,50);
t3.setBounds(350,300,150,50);

l4.setBounds(150,50,200,50);

b1.setBounds(100,400,100,50);
b2.setBounds(220,400,100,50);
b3.setBounds(340,400,100,50);
b4.setBounds(460,400,100,50);
b5.setBounds(250,475,150,50);

add(l1);
add(t1);

add(l2);
add(t2);

add(l3);
add(t3);

add(l4);

add(b1);
add(b2);
add(b3);
add(b4);
add(b5);

b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);
b4.addActionListener(this);
b5.addActionListener(this);
}

public void actionPerformed(ActionEvent ae){
    if(ae.getSource() == b5){
        System.exit(1);
}
```

```

    }
try{
    int a,b,c;
    String s1 = t1.getText();
    String s2 = t2.getText();

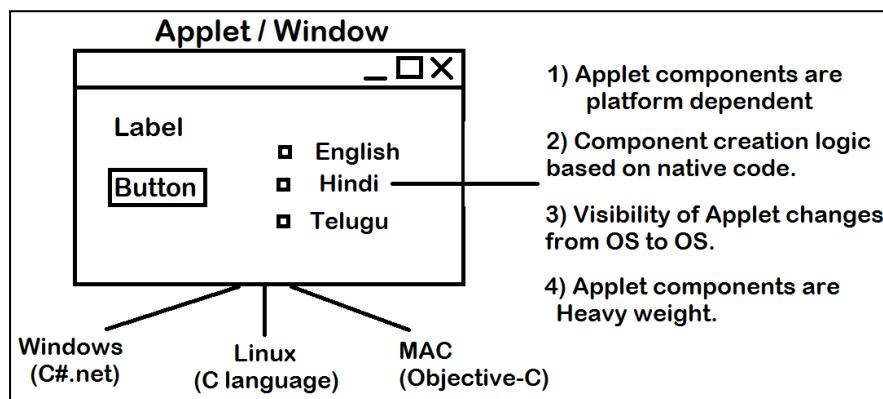
    a= Integer.parseInt(s1.trim());
    b= Integer.parseInt(s2.trim());
    if(ae.getSource() == b1){
        c = a+b ;
        String res = Integer.toString(c);
        t3.setText(res);
    }
    else if(ae.getSource() == b2){
        c = a-b ;
        String res = Integer.toString(c);
        t3.setText(res);
    }
    else if(ae.getSource() == b3){
        c = a*b ;
        String res = Integer.toString(c);
        t3.setText(res);
    }
    else if(ae.getSource() == b4){
        c = a/b ;
        String res = Integer.toString(c);
        t3.setText(res);
    }
}
catch (Exception e){
    t4.setText("Error Info : "+e.getMessage());
}
}

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

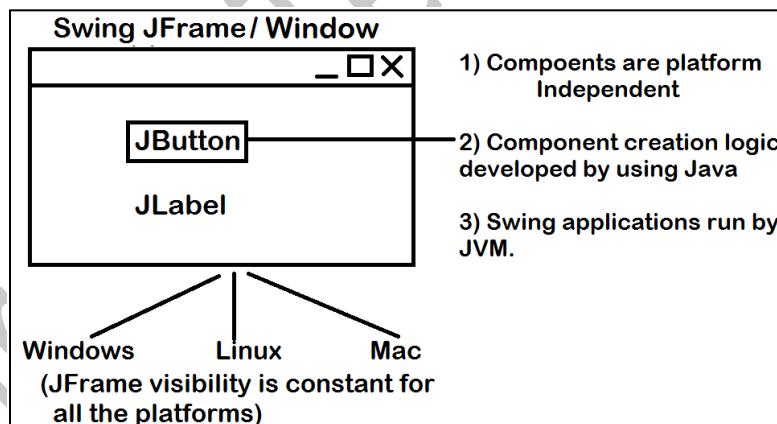
```

Swings API

- Java API provides extended functionality(updated) to develop standalone GUI applications called Swings API.
- The classes which are available in applet package are platform dependant.
- Components are created by using Applets are dependant to native code(OS code).
- Applet components are called Heavy weight(native) components.



- Swing components are platform independant, because the complete logic developed using java language only.
- Swing components are called Light weight(pure java) components.



```
package online;
import javax.swing.JFrame;
public class FirstFrame extends JFrame
{
    public static void main(String[] args)
    {
        new FirstFrame();
    }
}
```

By Srinivas (C/DS/Java trainer)

Page 296

```
package online;
import javax.swing.JFrame;
public class FirstFrame extends JFrame
{
    FirstFrame()
    {
        this.setSize(600, 400);
        this.setTitle("First Frame");
        this.setVisible(true);
    }
    public static void main(String[] args)
    {
        new FirstFrame();
    }
}
```

```
package online;
import javax.swing.JButton;
import javax.swing.JFrame;
public class FirstFrame extends JFrame
{
    JButton b ; //instance variable
    FirstFrame()
    {
        this.setSize(600, 400);
        this.setTitle("First Frame");

        this.b = new JButton("Click");
        this.add(b);
        this.setVisible(true);
    }
    public static void main(String[] args)
    {
        new FirstFrame();
    }
}
```

Layout : It is the proper arrangement of components on the window.
Java api providing many pre-defined layout classes.

FlowLayout
GridLayout
BoxLayout
.....

```

import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class FirstFrame extends JFrame
{
    JButton b ; //instance variable
    FirstFrame()
    {
        this.setSize(600, 400);
        this.setTitle("First Frame");
        this.setLayout(new FlowLayout());

        this.b = new JButton("Click");
        this.add(b);
        this.setVisible(true);
    }
    public static void main(String[] args)
    {
        new FirstFrame();
    }
}

```

We can change the properties of FlowLayout arrangement of components.

```

package online;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class FirstFrame extends JFrame
{
    JButton b[]; //instance variable
    FirstFrame()
    {
        this.setSize(600, 400);
        this.setTitle("First Frame");
        //this.setLayout(new FlowLayout()); //centered
        this.setLayout(new FlowLayout(FlowLayout.RIGHT));

        this.b = new JButton[5];
        for(int i=0 ; i<b.length ; i++)
        {
            this.b[i] = new JButton("Button-"+i);
            this.add(b[i]);
        }
        this.setVisible(true);
    }
    public static void main(String[] args)
    {
}

```

```

        new FirstFrame();
    }
}

package online;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class FirstFrame extends JFrame
{
    JButton b1, b2;
    FirstFrame()
    {
        this.setSize(600, 400);
        this.setTitle("First Frame");
        this.setLayout(null);

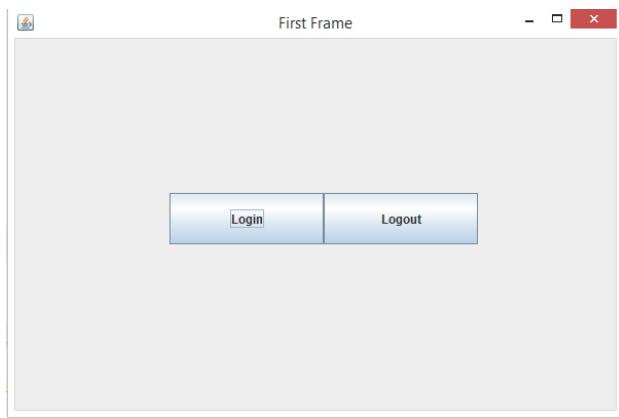
        this.b1 = new JButton("Login");
        this.b2 = new JButton("Logout");

        b1.setBounds(150,150,150,50);
        b2.setBounds(300,150,150,50);

        this.add(b1);
        this.add(b2);

        this.setVisible(true);
    }
    public static void main(String[] args)
    {
        new FirstFrame();
    }
}

```



Implementing ActionListener interface using same class :

```
package online;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
public class FirstFrame extends JFrame implements ActionListener
{
    JButton b1, b2;
    FirstFrame()
    {
        this.setSize(600, 400);
        this.setTitle("First Frame");
        this.setLayout(null);

        this.b1 = new JButton("Login");
        this.b2 = new JButton("Logout");

        b1.setBounds(150,150,150,50);
        b2.setBounds(300,150,150,50);

        b1.addActionListener(this);
        b2.addActionListener(this);

        this.add(b1);
        this.add(b2);

        this.setVisible(true);
    }
    public static void main(String[] args)
    {
        new FirstFrame();
    }

    public void actionPerformed(ActionEvent ae)
    {
        if(ae.getSource() == b1)
        {
            System.out.println("Login successful....");
        }
        else
        {
            System.exit(1);
        }
    }
}
```

Implementing ActionListener interface using Inner classes concept :

```
package online;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
public class FirstFrame extends JFrame
{
    JButton b1, b2 ;
    FirstFrame()
    {
        this.setSize(600, 400);
        this.setTitle("First Frame");
        this.setLayout(null);

        this.b1 = new JButton("Login");
        this.b2 = new JButton("Logout");

        b1.setBounds(150,150,150,50);
        b2.setBounds(300,150,150,50);

        b1.addActionListener(new Button1Inner());
        b2.addActionListener(new Button2Inner());

        this.add(b1);
        this.add(b2);

        this.setVisible(true);
    }
    public static void main(String[] args)
    {
        new FirstFrame();
    }
}

class Button1Inner implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
        System.out.println("Login successful....");
    }
}
class Button2Inner implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
        System.exit(1);
    }
}
```

```

        }
    }

Implementing ActionListener interface using Anonymous inner classes :
package online;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
public class FirstFrame extends JFrame
{
    JButton b1, b2;
    FirstFrame()
    {
        this.setSize(600, 400);
        this.setTitle("First Frame");
        this.setLayout(null);

        this.b1 = new JButton("Login");
        b1.setBounds(150,150,150,50);
        this.add(b1);
        b1.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent ae)
            {
                System.out.println("Login sucess...");
            }
        });
    }

    this.b2 = new JButton("Logout");
    b2.setBounds(300,150,150,50);
    this.add(b2);
    b2.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent ae)
        {
            System.out.println("Logout... ");
            System.exit(1);
        }
    });
}

this.setVisible(true);
}
public static void main(String[] args)
{
}

```

```

        new FirstFrame());
    }
}

DropDown box in Swings :
package online;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JComboBox;
import javax.swing.JFrame;
public class FirstFrame extends JFrame
{
    JComboBox jc;
    FirstFrame()
    {
        this.setSize(600, 400);
        this.setTitle("First Frame");
        this.setLayout(null);

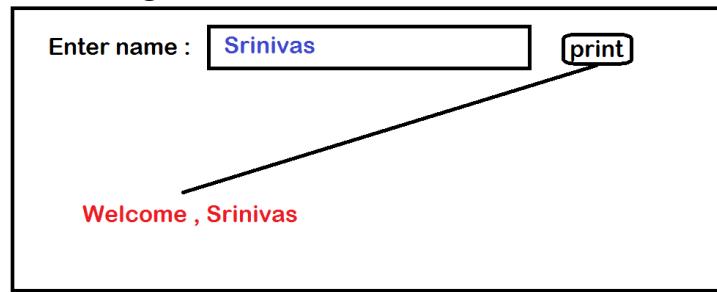
        jc = new JComboBox();
        jc.setFont(new Font("Arial", Font.BOLD, 30));
        String s[] = {"One", "Two", "Three", "Four"};
        for(int i=0 ; i<s.length ; i++)
        {
            jc.addItem(s[i]);
        }
        jc.setBounds(200,150,200,50);
        this.add(jc);
        jc.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent ae)
            {
                String item = (String) jc.getSelectedItem();
                System.out.println("Selected item : "+item);
            }
        });
    }

    this.setVisible(true);
}
public static void main(String[] args)
{
    new FirstFrame();
}

```

}

Program to implement following window :



```
package online;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
public class FirstFrame extends JFrame
{
    JLabel l1 , l2;
    JTextField t1 ;
    JButton b1 ;
    FirstFrame()
    {
        this.setSize(1000, 500);
        this.setTitle("First Frame");
        this.setLayout(null);

        l1 = new JLabel("Enter name :");
        l1.setFont(new Font("Arial", Font.BOLD , 35));
        l1.setBounds(100,70,250,50);
        this.add(l1);

        t1 = new JTextField();
        t1.setFont(new Font("Arial", Font.BOLD , 35));
        t1.setForeground(Color.BLUE);
        t1.setBounds(350,70,400,50);
        this.add(t1);

        b1 = new JButton("Print");
        b1.setFont(new Font("Arial", Font.BOLD , 25));
        b1.setBounds(800,75,100,40);
        this.add(b1);
    }
}
```

```

        b1.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent ae)
        {
            String name = t1.getText();
            l2.setText("Hello "+name+", Welcome...");
        }
    });
}

l2 = new JLabel();
l2.setFont(new Font("Arial", Font.BOLD, 40));
l2.setBounds(100,250,700,50);
l2.setForeground(Color.RED);
this.add(l2);

this.setVisible(true);
}
public static void main(String[] args)
{
    new FirstFrame();
}
}

```

Program to close the window when user click of close button.

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Second extends JFrame implements ActionListener
{
    static JButton b1;
    public Second(String name)
    {
        this.setTitle(name);
        this.setSize(600,600);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setVisible(true);

        setLayout(null);
        b1=new JButton("PREVIOUS");
        this.add(b1);
        b1.setBounds(220,250,150,50);
        b1.addActionListener(this);

    }
    public void actionPerformed(ActionEvent ae)
    {

```

```

        this.dispose();
        new First("FIRST FRAME");
    }

}

Create Menu and Sub menu :
import java.awt.event.*;
import javax.swing.*;
public class SubmenuExample extends JFrame
{
    public SubmenuExample()
    {
        initUI();
    }
    private void initUI()
    {
        JMenuBar menubar = new JMenuBar();
        ImageIcon iconNew = new ImageIcon("new.png");
        ImageIcon iconOpen = new ImageIcon("open.png");
        ImageIcon iconSave = new ImageIcon("save.png");
        ImageIcon iconExit = new ImageIcon("exit.png");

        JMenu file = new JMenu("File");
        file.setMnemonic(KeyEvent.VK_F);

        JMenu imp = new JMenu("Import");
        imp.setMnemonic(KeyEvent.VK_M);

        JMenuItem newsf = new JMenuItem("Import newsfeed list...");
        JMenuItem bookm = new JMenuItem("Import bookmarks...");
        JMenuItem mail = new JMenuItem("Import mail...");

        imp.add(newsf);
        imp.add(bookm);
        imp.add(mail);

        JMenuItem fileNew = new JMenuItem("New", iconNew);
        fileNew.setMnemonic(KeyEvent.VK_N);

        JMenuItem fileOpen = new JMenuItem("Open", iconOpen);
        fileOpen.setMnemonic(KeyEvent.VK_O);

        JMenuItem fileSave = new JMenuItem("Save", iconSave);
        fileSave.setMnemonic(KeyEvent.VK_S);

        JMenuItem fileExit = new JMenuItem("Exit", iconExit);
        fileExit.setMnemonic(KeyEvent.VK_C);
    }
}

```

```
fileExit.setToolTipText("Exit application");
fileExit.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_W,
    ActionEvent.CTRL_MASK));

fileExit.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent event) {
    System.exit(0);
}

file.add(fileNew);
file.add(fileOpen);
file.add(fileSave);
file.addSeparator();
file.add(imp);
file.addSeparator();
file.add(fileExit);
menubar.add(file);
setJMenuBar(menubar);
setTitle("Submenu");
setSize(600,600);
 setLocationRelativeTo(null);
setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public static void main(String[] args) {
SwingUtilities.invokeLater(new Runnable() {
@Override
public void run() {
    SubmenuExample ex = new SubmenuExample();
    ex.setVisible(true);
}
});
}
}
```

Network programming

- Network programming refers, writing programs that execute across multiple devices (computers), in a network.
- java.net package contains a collection of classes and interfaces to implement network programming.
- Computers running on the network use 2 protocols to communicate
 1. TCP (Transmission control Protocol)
 2. UDP (User Datagram Protocol)

TCP (Transmission Control Protocol):

- It is a connection-based protocol that provides a reliable flow of data between two computers.
- Making a telephone call in which the data will be transferred from source to destination in the sending order.

UDP (User Datagram Protocol):

- It is a protocol that sends independent packets of data called datagrams.
- UDP protocol sends the data with no guarantees about arrival.
- UDP is not connection-based like TCP.
- For example, a clock server sends the current time to client on request.

Port :

- Every computer which is connected in the network has a single physical connection.
- All data destined for a particular computer will be collected through that connection called Socket.

Question: How does the computer know, to which application it has to forward the data that is collected through Socket?

- In one computer, we can run number of applications simultaneously.
- Each application should run at a specific port number.
- Hence OS can identify the application by its port number to send the data collected through single physical connection (Socket).
- A 16-bit computer is having 2^{16} (65536) ports to run applications.
- Port numbers ranging from 0-1023 are restricted, these ports always used by well known system applications.

IP address

- An Internet Protocol address (IP address) is a numerical label assigned to each device which is connected in the network.
- Using IP address only, it is possible to send the information to a specific system connected in Network.
- IP address having 2 versions

Internet Protocol Version 4 (IPv4)

Internet Protocol Version6 (IPv6)

Networking Classes in the Java API :

- The following classes used to communicate in the network using TCP and UDP
- The **URL**, **URLConnection**, **Socket**, and **ServerSocket** classes all use TCP to communicate over the network.
- The **DatagramPacket**, **DatagramSocket**, and **MulticastSocket** classes are for use with UDP.

URL:

- Stands for Uniform Resource Locator.
- It is a reference (an address) to a resource(application) on the Internet.
- A URL has two main components:
- For example :

Protocol identifier: `http://`
Resource name: `www.google.com`

```
import java.net.*;
import java.io.*;
public class ParseURL
{
    public static void main(String[] args) throws Exception
    {
        URL aURL = new
URL("http://localhost:8086/RequestApp/go?userName=naresh&userPass=srinivas");
        System.out.println("protocol = " + aURL.getProtocol());
        System.out.println("host = " + aURL.getHost());
        System.out.println("port = " + aURL.getPort());
        System.out.println("path = " + aURL.getPath());
        System.out.println("query = " + aURL.getQuery());
        System.out.println("filename = " + aURL.getFile());
    }
}
```

Reading data from a URL

URL class is having a pre-defined method that creates a stream(pointer) to read the information of that page.

```
import java.net.*;
import java.io.*;
public class URLReader
{
    public static void main(String[] args) throws Exception
    {
        URL read = new URL("http://localhost:8086/RequestApp/login.html");
        InputStreamReader isr = new InputStreamReader(read.openStream());
        BufferedReader in = new BufferedReader(isr);
        String line;
        while ((line = in.readLine()) != null)
```

```

        {
            System.out.println(line);
        }
        in.close();
    }
}

```

Find IP address: Every system it is connected in the network can be recognized by a unique address and can send the information using Socket connection.

```

import java.net.*;
import java.io.*;
public class IP
{
    public static void main (String[] args) throws IOException
    {
        if(args.length == 0)
        {
            System.out.println("Input host name");
        }
        else
        {
            String host = args[0];
            try
            {
                InetAddress ip = InetAddress.getByName(host);
                System.out.println("IP address: " + ip.getHostAddress());
            }
            catch ( UnknownHostException e )
            {
                System.out.println("Could not find IP address for: " + host);
            }
        }
    }
}

```

Socket Overview

- A socket forms the interface between the protocol and client for communication.
- A Java socket forms the base for data transmission between two computers using TCP/IP.
- The socket specifies the site address and the connection port.
- When packets reach a client or server, they are routed to the destination port specified in packet header.

java.net Package

- The java.net package contains the classes and interfaces required for networking.
- Some important classes are
 1. MulticastSocket,

2. ContentHandler,
3. URLServerSocket,
4. Socket,
5. InetAddress,
6. URLConnection,
7. DatagramSocket, and DatagramPacket.

The TCP/IP Client Socket

- A TCP/IP client socket is used to create a reliable, bi-directional, stream-based connection between two computers on a network.
- The client socket is implemented by creating an instance of the Socket class.
- It is designed to connect to the server and initialize protocol exchanges.

Socket (String hostname, int port)

```
import java.io.*;
import java.net.*;
public class ExSocket
{
    public static void main(String args[]) throws UnknownHostException
    {
        try
        {
            Socket mySocket = new Socket("www.google.com",80);
            System.out.println("Connection to: " + mySocket.getInetAddress());
            System.out.println("Port Number: " + mySocket.getPort());
            System.out.println("Local Address: " + mySocket.getLocalAddress());
            System.out.println("Local Port: " + mySocket.getLocalPort());
        }
        catch (UnknownHostException e)
        {
            System.out.println("Site not found!");
        }
        catch (SocketException e)
        {
            System.out.println("Socket error");
        }
        catch ( IOException e)
        {
            System.out.println("An I/O Exception Occurred!");
        }
    }
}
```

The TCP/IP Server Socket :

- The TCP/IP server socket creates a socket that listens for incoming connections.
- The server socket is implemented by creating an instance of the ServerSocket class.

- The server socket creates a server on the system to detect client connections.
ServerSocket(int port1)

```

import java.io.*;
import java.net.*;
public class SerSocket
{
    public static void main(String args[])
    {
        int port=1080;
        try
        {
            ServerSocket mySocket = new ServerSocket(port);
            System.out.println("Server initialized on port " + port);
            mySocket.getLocalPort();
            {
                while(true)
                {
                    mySocket.accept();
                }
            }
        }
        catch (Exception e)
        {
            System.out.println("Exception : "+e.getMessage());
        }
    }
}

```

A CHAT PROGRAM

CLIENT SIDE CODE:

```

import java.net.*;
import java.io.*;
class ClientChild implements Runnable
{
    Thread t;
    Socket client;
    ClientChild(Socket client)
    {
        this.client=client;
        t=new Thread(this);
        t.start();
    }
    public void run()
    {
        try

```

```

        {
            BufferedReader br=new BufferedReader(new
InputStreamReader(client.getInputStream()));
            while(true){
                String msg = br.readLine();
                System.out.println("Server:"+msg);
            }
        }
    catch(IOException e){
        System.out.println(e);
    }
}
}

class ClientSide
{
    public static void main(String args[]) throws IOException
    {
        try
        {
            System.out.println("sending request to peer....");
            Socket client = new Socket("127.0.0.1",1300);
            System.out.println("successfully connected");
            ClientChild c = new ClientChild(client);
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            PrintStream ps=new PrintStream(client.getOutputStream());
            while(true){
                String s = br.readLine();
                ps.println(s);
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}

```

SERVER SIDE CODE:

```

import java.net.*;
import java.io.*;
class ServerChild implements Runnable
{
    Thread t;
    Socket client;
    ServerChild(Socket client)
    {
        this.client = client;
    }
}

```

```

        t = new Thread(this);
        t.start();
    }
    public void run()
    {
        try
        {
            BufferedReader br=new BufferedReader(new
InputStreamReader(client.getInputStream()));
            while(true)
            {
                String msg = br.readLine();
                System.out.println("Client: "+msg);
            }
        }
        catch(IOException e)
        {
            System.out.println(e);
        }
    }
}
class ServerSide
{
    public static void main(String args[])
    throws IOException
    {
        ServerSocket server=new ServerSocket(1300);
        System.out.println("waiting for request from peer.....");
        Socket client = server.accept();

        ServerChild s = new ServerChild(client);
        System.out.println("request accepted");

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintStream ps2=new PrintStream(client.getOutputStream());
        while(true)
        {
            String st = br.readLine();
            ps2.println(st);
        }
    }
}

```