# Core JAVA Left over Topics-Part 2(Main Part - 8)

**BufferedWriter:**

By using BufferedWriter object we can write character data to the file.

**Constructor**

BufferedWriter bw=new BufferedWriter(writer w);

BufferedWriter bw=new BufferedWriter(writer w,int buffersize);

**Note:BufferedWriter never communicates directly with the file it should communicates via some writer object.**

**Which of the following declarations are valid?**

1. BufferedWriter bw =new BufferedWriter("cricket.txt"); (invalid)
2. BufferedWriter bw =new BufferedWriter (new File("cricket.txt")); (invalid)
3. BufferedWriter bw =new BufferedWriter (new FileWriter("cricket.txt")); (valid)
4. BufferedWriter bw =new BufferedWriter(new BufferedWriter(new FileWriter("crickter.txt")));

**Methods**

1. write(int ch);
2. write(char[] ch);
3. write(String s);
4. flush();
5. close();
6. newline();

**Inserting a new line character to the file.**

eg#1.

```
import java.io.*;
import java.util.*;
public class Test
{
    public static void main(String[] args) throws IOException
    {
        //BufferedReader,BufferedWriter
        BufferedWriter bw = new BufferedWriter(new FileWriter("cricket.txt"));
        bw.write(97);
        bw.newLine();
        char[] arr = {'p','w','s','k','i','l','l','s'};
        bw.write(arr);
        bw.newLine();
```

```
            bw.write("PWIOI");
            bw.newLine();
            bw.flush();
            bw.close();
        }
}
```

**Note**

1.bw.close()// recomended to use

**//When ever we are closing BufferedWriter automatically underlying writer will be closed and we are not close explicitly.**

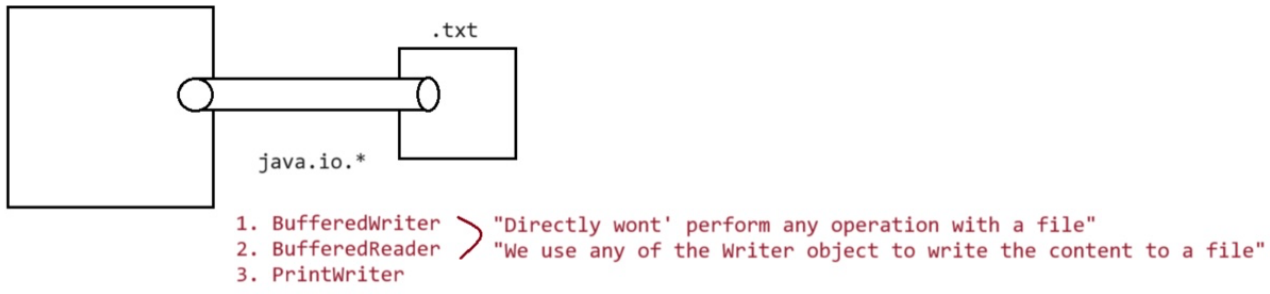
2.fw.close()**// not recomended to use**

**=> When ever we are closing BufferedWriter automatically underlying writer will be closed and we are not close explicitly.**

**Note:**

When compared with FileWriter which of the following capability(facility) is available as method in BufferedWriter.

1. Writing data to the file.
2. Closing the writer.
3. Flush the writer.
4. **Inserting newline character.**




```
BufferedWriter bw = new BufferedWriter("cricket.txt"); //invalid
BufferedWriter bw = new BufferedWriter(new File("Cricket.txt"));//invalid
BufferedWriter bw = new BufferedWriter(new FileWriter("cricket.txt"));//valid
```

**BufferedReader:**

This is the most enhanced(better) Reader to read character data from the file.

**Constructors:**

BufferedReader br=new BufferedReader(Reader r);

BufferedReader br=new BufferedReader(Reader r,int buffersize);

**Note**

**=> BufferedReader can not communicate directly with the File it should communicate via some Reader object.**

**=> The main advantage of BufferedReader over FileReader is we can read data line by line instead of character by character.**

**Methods:**

1. int read();
2. int read(char[] ch);
3. String readLine();

   **It attempts to read next line and return it , from the File. if the next line is not available then this method returns null.**

4. void close();

**eg#1.**

```
import java.io.*;
import java.util.*;
//Client Code
public class Test
{
    public static void main(String[] args) throws IOException
    {
        //BufferedReader,BufferedWriter
        BufferedReader br = new BufferedReader(new FileReader("Sample.txt"));
        String line = br.readLine();
        while (line!=null)
        {
            System.out.println(line);
            line =br.readLine();
        }
        br.close();
    }
}
```

Output

**Sachin**

**Saurav**

**Dhoni**

**kohli**

**raina**

**dravid**

**yuvi**

**zaheer**

**munaf**

**Note:**

1.br.close() // recomended to use

2.fw.close() // not recomended to use

**=> Whenever we are closing BufferedReader automatically underlying FileReader will be closed it is not required to close explicitly.**

**=> Even this rule is applicable for BufferedWriter also.**

**PrintWriter:**

=> This is the most enhanced Writer to write text data to the file.

**=> By using FileWriter and BufferedWriter we can write only character data to the File but by using PrintWriter we can write any type of data to the File.**

**Constructors:**

PrintWriter pw=new PrintWriter(String name);

PrintWriter pw=new PrintWriter(File f);

PrintWriter pw=new PrintWriter(Writer w);

**PrintWriter can communicate either directly to the File or via some Writer object also.**

**Methods:**

1. write(int ch);
2. write (char[] ch);
3. write(String s);
4. flush();
5. close();
6. print(char ch);
7. print (int i);
8. print (double d);
9. print (boolean b);
10. print (String s);
11. println(char ch);
12. println (int i);
13. println(double d);
14. println(boolean b);

15. println(String s);

**eg#1.**

```
import java.io.*;
import java.util.*;
//Client Code
public class Test
{
    public static void main(String[] args) throws IOException
    {
        //Handled only character type of data
        //FileWriter ---> BufferedWriter---> PrintWriter(println())
        //FileReader ---> BufferdReader(readLine())

        //Handle images,video file,audio files,... -> binary data
        //InputStream :: BufferedInputStream
        //OutputStream :: BufferedOutputStream

        //PrintWriter
        PrintWriter out = new PrintWriter(new FileWriter("Cricket.txt"));
        out.write(97);//97 -> unicode value
        out.println(100);//100 -> int value
        out.println(true);
        out.println('c');
        out.println(10.5f);
        out.println("sachintendulkar");
        out.flush();
        out.close();
    }
}
```

**cricket.txt**

**++++++++++**

**a100**

**true**

**c**

**10.5**

**sachintendulkar**

**What is the difference between write(100) and print(100)?**

=> In the case of write(100) the corresponding character "d" will be added to the File but

=> In the case of print(100) "100" value will be added directly to the File.

**Note 1:**

1. The most **enhanced Reader** to read character data from the File is **BufferedReader**.
2. The most **enhanced Writer** to write character data to the File is **PrintWriter**.

**Note 2:**

1. In general we can use Readers and Writers to handle character data. Where as we can use **InputStreams and OutputStreams to handle binary data(like images, audio files, video files etc).**
2. We can use OutputStream to write binary data to the File and we can use InputStream to read binary data from the File

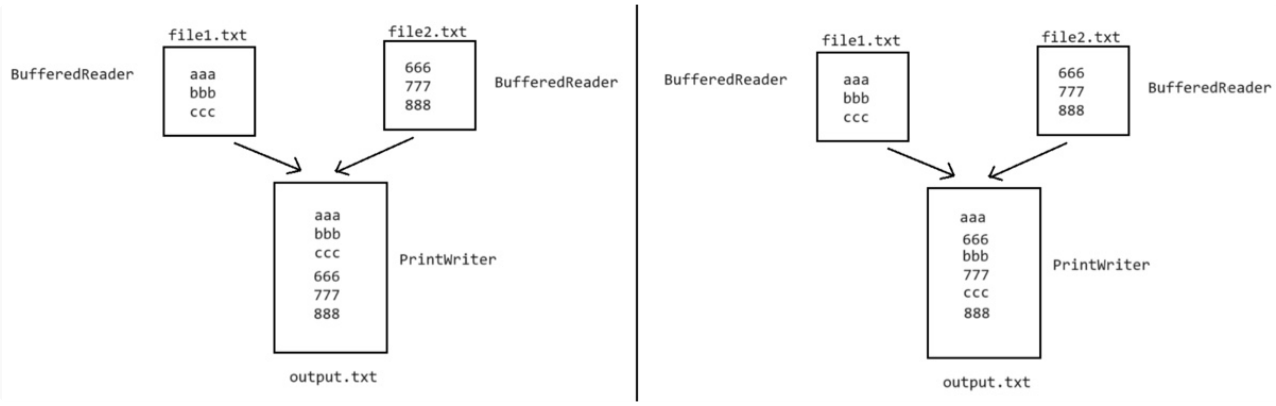
   **Character Data =>** Reader and Writer

   **Binary Data =>** InputStream and OutputStream

**eg#1.Requirement => file1.txt ,file2.txt copy all the contents to output.txt**

```
import java.io.*;
import java.util.*;
public class Test
{
    public static void main(String[] args) throws IOException
    {
        //Setting the writer object to output.txt
        PrintWriter out = new PrintWriter("output.txt");
        //Reader the data from file1.txt
        BufferedReader br1 = new BufferedReader(new FileReader("file1.txt"));
        String line = br1.readLine();
        while (line!=null)
        {
            //write the content to output.txt
            out.println(line);
            line = br1.readLine();
        }
        //Reader the data from file2.txt
        br1 = new BufferedReader(new FileReader("file2.txt"));
        line = br1.readLine();
        while (line!=null)
```

```
        {
            //write the content to output.txt
            out.println(line);
            line = br1.readLine();
        }
        out.flush();
        out.close();
        br1.close();
    }
}
```




**eg#2.(second part of above image)**

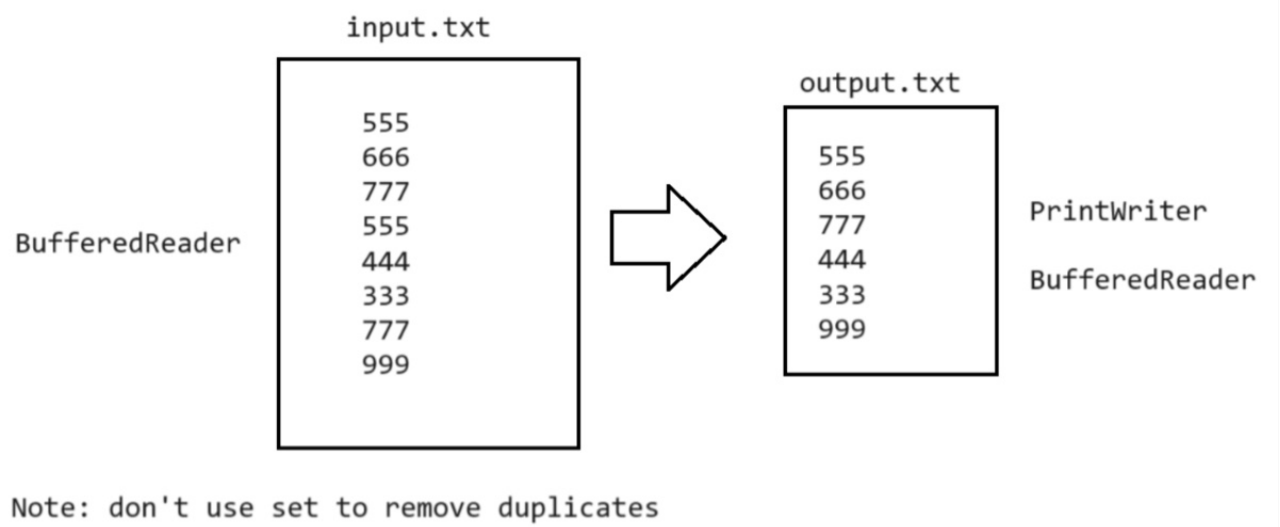
**Requirement => file1.txt file2.txt copy one line from file1.txt and from file2.txt to file3.txt.**

```
import java.io.*;
import java.util.*;
public class Test
{
    public static void main(String[] args) throws IOException
    {
        //Requirement => file1.txt ,file2.txt copy all the contents to output.txt
        //Setting the writer object to output.txt
        PrintWriter out = new PrintWriter("output.txt");
        BufferedReader br1 = new BufferedReader(new FileReader("file1.txt"));
        BufferedReader br2 = new BufferedReader(new FileReader("file2.txt"));
        //Reading the data from file1.txt and file2.txt
        String line1 = br1.readLine();
        String line2 = br2.readLine();
        while (line1!=null || line2!=null)
```

```
        {
            //write the content to output.txt
            if (line1!=null)
            {
                out.println(line1);
                line1 = br1.readLine();
            }
            if (line2!=null)
            {
                out.println(line2);
                line2 = br2.readLine();
            }
        }
        out.flush();
        out.close();
        br1.close();
        br2.close();
    }
}
```

**eg#3.**

**Requirement => Write a program to perform extraction of mobile no only if there is no duplicates**




```
import java.io.*;
import java.util.*;
//Client Code
```

```
public class Test
{
    public static void main(String[] args) throws IOException
    {
        //Requirement => file1.txt ,file2.txt copy all the contents to output.txt
        //Setting the writer object to output.txt
        PrintWriter out = new PrintWriter("output.txt");
        //Setting the reader object to input.txt
        BufferedReader br = new BufferedReader(new FileReader("input.txt"));
        String target =br.readLine();
        while (target!=null)
        {
            boolean isAvailable =false;
            BufferedReader br1 = new BufferedReader(new FileReader("output.txt"));
            String line =br1.readLine();
            //loop and check whether target exists or not in the output.txt file
            while (line!=null)
            {
                if (target.equals(line))
                {
                    isAvailable = true;
                    break;
                }
                //read nextLine from output.txt
                line = br1.readLine();
            } //end of while loop
            if (isAvailable==false)
            {
                //write to output.txt
                out.println(target);
                out.flush();
            }
            //read the next line from input.txt
            target =br.readLine();
        }//end of while loop
        out.close();
        br.close();
    }
}
```

**eg#4.**

**Requirement => Write a program to remove duplicates from the file**

```
package in.pwskills;
public class Sample
{
    static final String name = new String("sachin");
}
```

**Write a code to find the length of the String present in name variable**

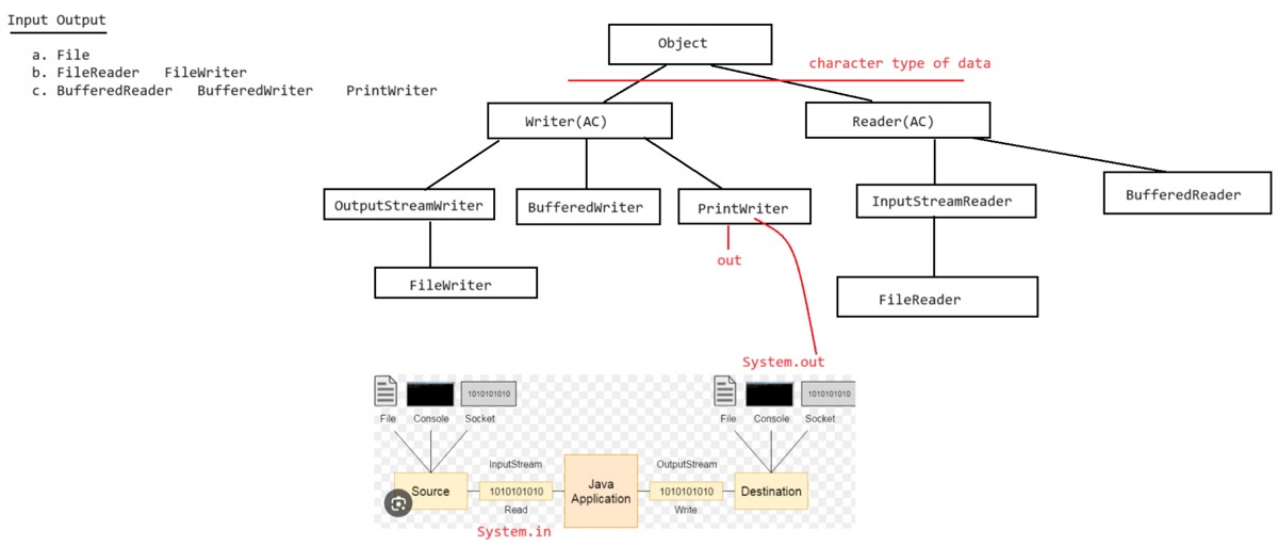
**=> Sample.name.length()**

```
package java.lang;
import java.io.PrintWriter;
class System
{
    public static final Writer out = new PrintWriter();
}
package java.io;
class PrintWriter
{
    println(){
    }
    println(int){
    }
    println(String){
    }
    print(int){
    }
    print(){
    }
}
```

**UserCode**

```
import java.lang.*; //default package avaialable to Compiler and JVM
public class Test{
    public static void main(String[] args){
            System.out.println("Welcome to java coding....");
```

```
    }
}
```




**Serialization and DeSerialization**

**Agenda :**

1. Serialization
2. Deserialization
3. transient keyword
4. static Vs transient
5. transient Vs final
6. Object graph in serialization.
7. customized serialization.
8. Serialization with respect inheritance.
9. Externalization
10. Difference between Serialization & Externalization
11. SerialVersionUID

**Serialization: (1.1 v)**

=> The process of saving (or) writing state of an object to a file is called serialization but strictly speaking it is the process of converting an object from java supported form to either network supported form (or) file supported form.

=> By using FileOutputStream and ObjectOutputStream classes we can achieve serialization process.

|=> writeObject(Object obj)

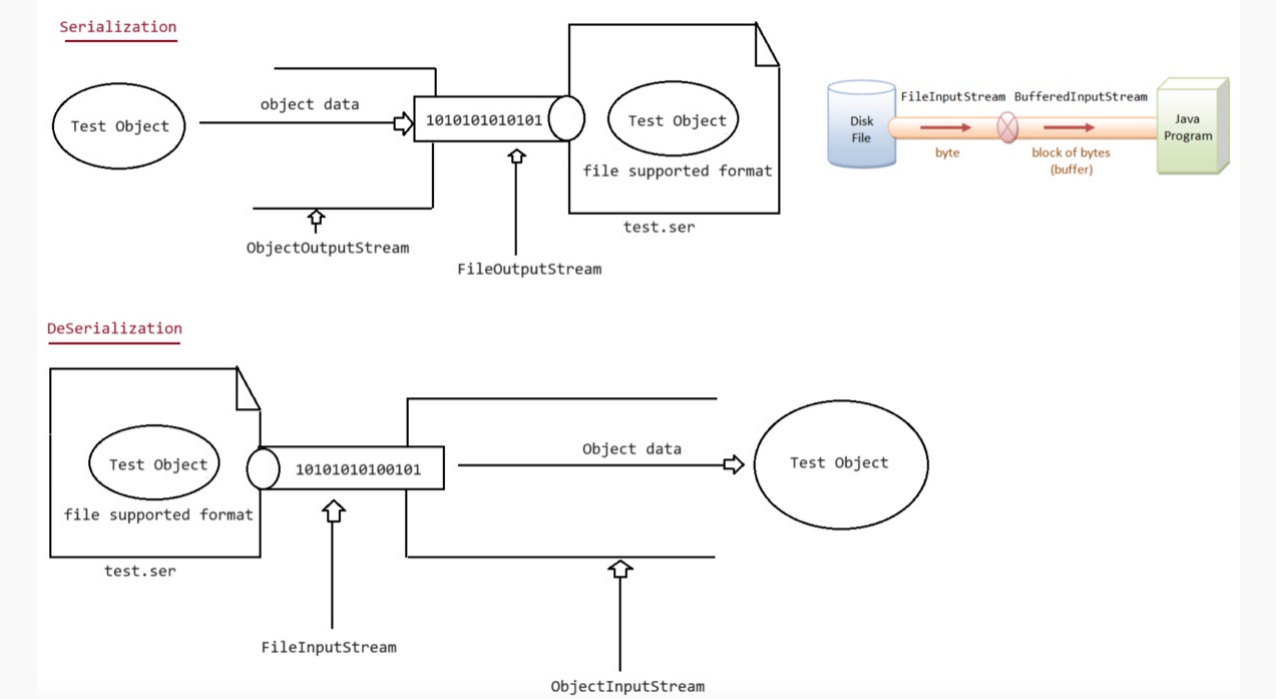
**Ex:** Myntra product

**De-Serialization:**

=> The process of reading state of an object from a file is called DeSerialization but strictly speaking it is the process of converting an object from file supported form (or) network supported form to java supported form.

=> By using FileInputStream and ObjectInputStream classes we can achieve DeSerialization.

|=> readObject()




**eg#1.**

```
import java.io.*;
//NotSerializableException
class Dog implements Serializable
{
    int i =10;
    int j =20;
}
public class Test
{
    public static void main(String[] args) throws Exception
    {
        Dog d1 = new Dog();

        System.out.println("Serialization Started");

        FileOutputStream fos = new FileOutputStream("Dog.ser");
```

```
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        System.out.println("Dog Object data :: "+d1.i+"-----"+d1.j);
        oos.writeObject(d1);
        System.out.println("Serialziation Completed");
        System.out.println();

        System.out.println("De-Serialization Started");
        FileInputStream fis = new FileInputStream("Dog.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();
        System.out.println("Dog Object data :: "+d2.i+"-----"+d2.j);
        System.out.println("De-Serialization Completed");
    }
}
```

**Output**

**Serialization Started**

**Dog Object data :: 10-----20**

**Serialziation Completed**

**De-Serialization Started**

**Dog Object data :: 10-----20**

**De-Serialization Completed**

**eg#2.**

```
import java.io.*;
//NotSerializableException
class Dog implements Serializable
{
    int i =10;
    int j =20;
}
class Cat implements Serializable{
    int i = 100;
    int j = 200;
}
public class Test
{
```

```
    public static void main(String[] args) throws Exception
    {
        Dog d1 = new Dog();
        Cat c1 = new Cat();

        System.out.println("Serialization Started");
        FileOutputStream fos = new FileOutputStream("obj.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        System.out.println("Dog Object data :: "+d1.i+"-----"+d1.j);
        System.out.println("Cat Object data :: "+c1.i+"-----"+c1.j);
        oos.writeObject(d1);
        oos.writeObject(c1);
        System.out.println("Serialziation Completed");
        System.out.println();

        System.out.println("De-Serialization Started");
        FileInputStream fis = new FileInputStream("obj.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        //order is important first we should de serialize dog as we serialize dog object first
        Cat c2 = (Cat)ois.readObject();
        Dog d2 = (Dog)ois.readObject();
        System.out.println("Dog Object data :: "+d2.i+"-----"+d2.j);
        System.out.println("Cat Object data :: "+c2.i+"-----"+c2.j);
        System.out.println("De-Serialization Completed");
    }
}
```

**Output**

**Serialization Started**

**Dog Object data :: 10-----20**

**Cat Object data :: 100-----200**

**Serialziation Completed**

**De-Serialization Started**

**Exception in thread "main" java.lang.ClassCastException: Dog cannot be cast to Cat**

**Note:**

1. We can perform Serialization only for Serilizable objects.

2. An object is said to be Serilizable if and only if the corresponding class implements Serializable interface.
3. Serializable interface present in java.io package and does not contain any methods. It is marker interface. The required ability will be provided automatically by JVM.
4. We can add any no. Of objects to the file and we can read all those objects from the file but in which order we wrote objects in the same order only the objects will come back. **That is order is important.**
5. **If we are trying to serialize a non-serializable object then we will get RuntimeException saying "NotSerializableException".**

**Transient keyword:**

1. transient is the modifier applicable **only for variables**,but not for classes and methods.
2. While performing serialization **if we don't want to save the value of a particular variable to meet security constant such type of variable** ,then we should declare that variable with "transient" keyword.
3. At the time of serialization **JVM ignores the original value of transient variable and save default value to the file .**
4. **That is transient means "not to serialize"**

**static Vs transient :**

1. **static variable is not part of object state hence they won't participate in serialization because of this declaring a static variable as transient there is no use.**

**Transient Vs Final:**

1. **final variables will be participated into serialization directly by their values. Hence declaring a final variable as transient there is no use.**

**=>final variable will not participate in serialization instead only its value will participate hence making it transient is of no use.**

```
//the compiler assign the value to final variable
import java.io.*;
//NotSerializableException
class Dog implements Serializable
{
    transient final int i =10;   //class variables
    transient static int j =20;   //value will particpate but not the variable
}
public class Test
{
    public static void main(String[] args) throws Exception
    {
```

```
        Dog d1 = new Dog();

        System.out.println("Serialization Started");
        FileOutputStream fos = new FileOutputStream("obj.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        System.out.println("Dog Object data :: "+d1.i+"-----"+d1.j);
        oos.writeObject(d1);
        System.out.println("Serialziation Completed");

        System.out.println();

        System.out.println("De-Serialization Started");
        FileInputStream fis = new FileInputStream("obj.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();
        System.out.println("Dog Object data :: "+d2.i+"-----"+d2.j);
        System.out.println("De-Serialization Completed");
    }
}
```

**Output**

**declaration output**

**int i=10;**

**int j=20;**

**10.....20**

**transient int i=10;**

**int j=20;**

**0.....20**

**transient int i=10;**

**transient static int j=20;**

**0.....20**

**transient final int i=10;**

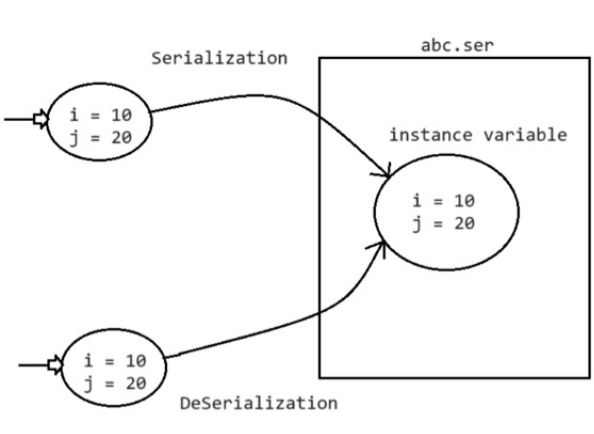
**transient int j=20;**

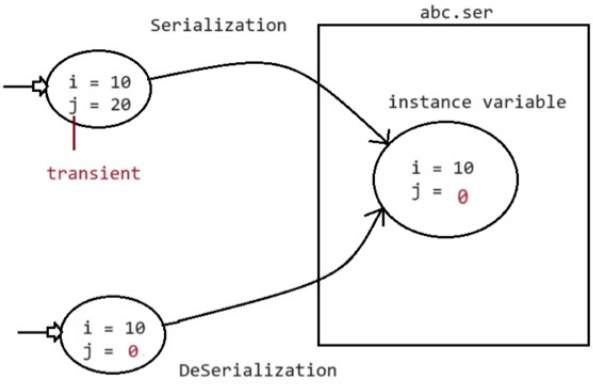
**10.....0**

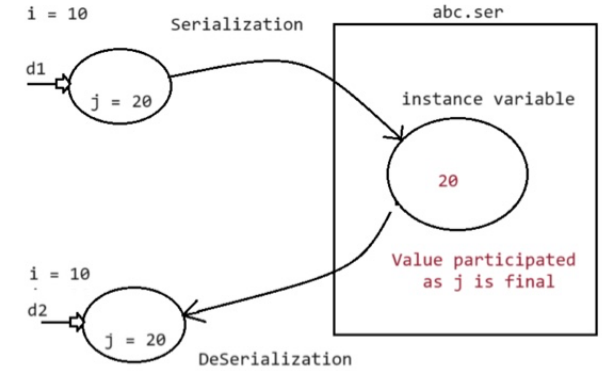
**transient final int i=10;**

**transient static int j=20;**

**10.....20**




```
class Dog  implements Serializable
{
    static transient int i =10;
    final  transient int j =20;
}
```

variable :Compile Time Constant
Value can't be changed
Variable will be replace with value by Compiler

```
class Test
{
    final int i = 10;
    int j =20;
    public void disp()
    {
        System.out.println(i); //System.out.println(10);
        System.out.println(j);
    }

}
```

Compilation code

**Object class methods**

public class Object {

public Object();
public final native Class<?> getClass();
//Commonly used methods on every object
public java.lang.String toString();
public native int hashCode();
public boolean equals(Object object);

```
    //cloning :: shallowcopy, deep copy
    protected native Object clone() throws CloneNotSupportedException;

    //multithreading environment
    public final native void notify();
    public final native void notifyAll();
    public final void wait() throws InterruptedException;
    public final native void wait(long) throws InterruptedException;
    public final void wait(long, int) throws InterruptedException;

    //Garbage Collector
    protected void finalize() throws java.lang.Throwable;
}
```

**Garbage Collector**

1. Introduction:
2. The way to make an object eligible for GC
   **i. Nullifying the reference variable**
   **ii. Reassign the reference variable**
   **iii. Objects created inside a method**
   **iv. Island of Isolation**
3. The methods for requesting JVM to run GC
   i. By System class
   ii. By Runtime class
4. Finalization

**=> Case 1 :** Just before destroying any object **GC calls finalize() met**hod on the object.

**=> Case 2 :** We can call finalize() method explicitly.

**=> Case 3 :** finalize() method can be call either by the programmer or by the GC.

**=> Case 4 :** On any object GC calls finalize() method only once Memory leak.

**Introduction:**

=> In old languages like C++ programmer is responsible for both creation and destruction of objects.

Usually programmer is taking very much care while creating object and neglect destruction of useless objects.

**Due to his negligence at certain point of time for creation of new object sufficient memory may not be available and entire application may be crashed due to memory problems.**

=> But in java programmer is responsible only for creation of new object and his not responsible for destruction of objects.

=> Sun people provided one assistant which is always running in the background for destruction at useless objects.

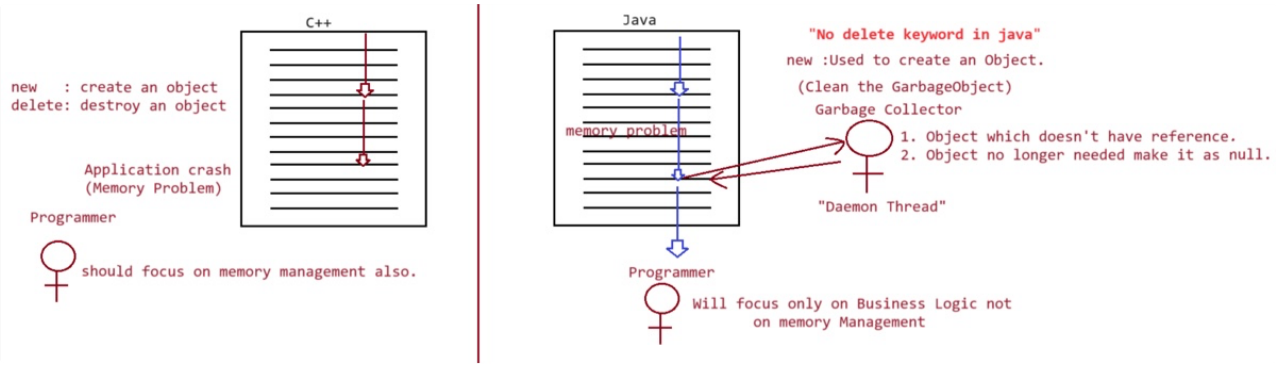
**Due to this assistant the chance of failing java program is very rare because of memory problems.**

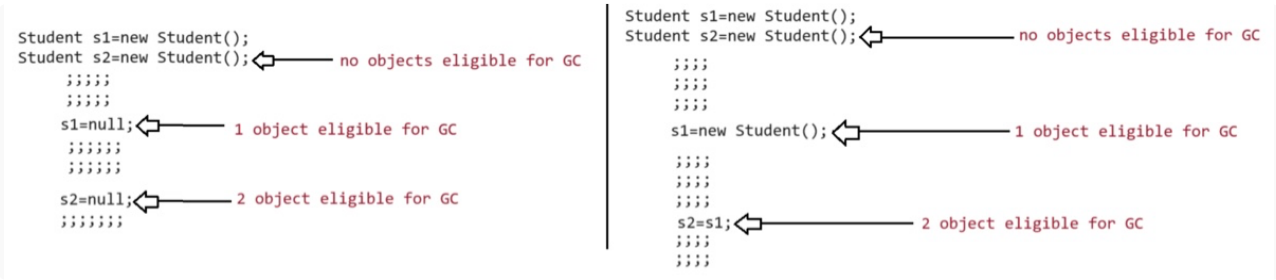
=> **This assistant is nothing but garbage collector.** Hence the main objective of GC is to destroy useless objects.

**The ways to make an object eligible for GC:**

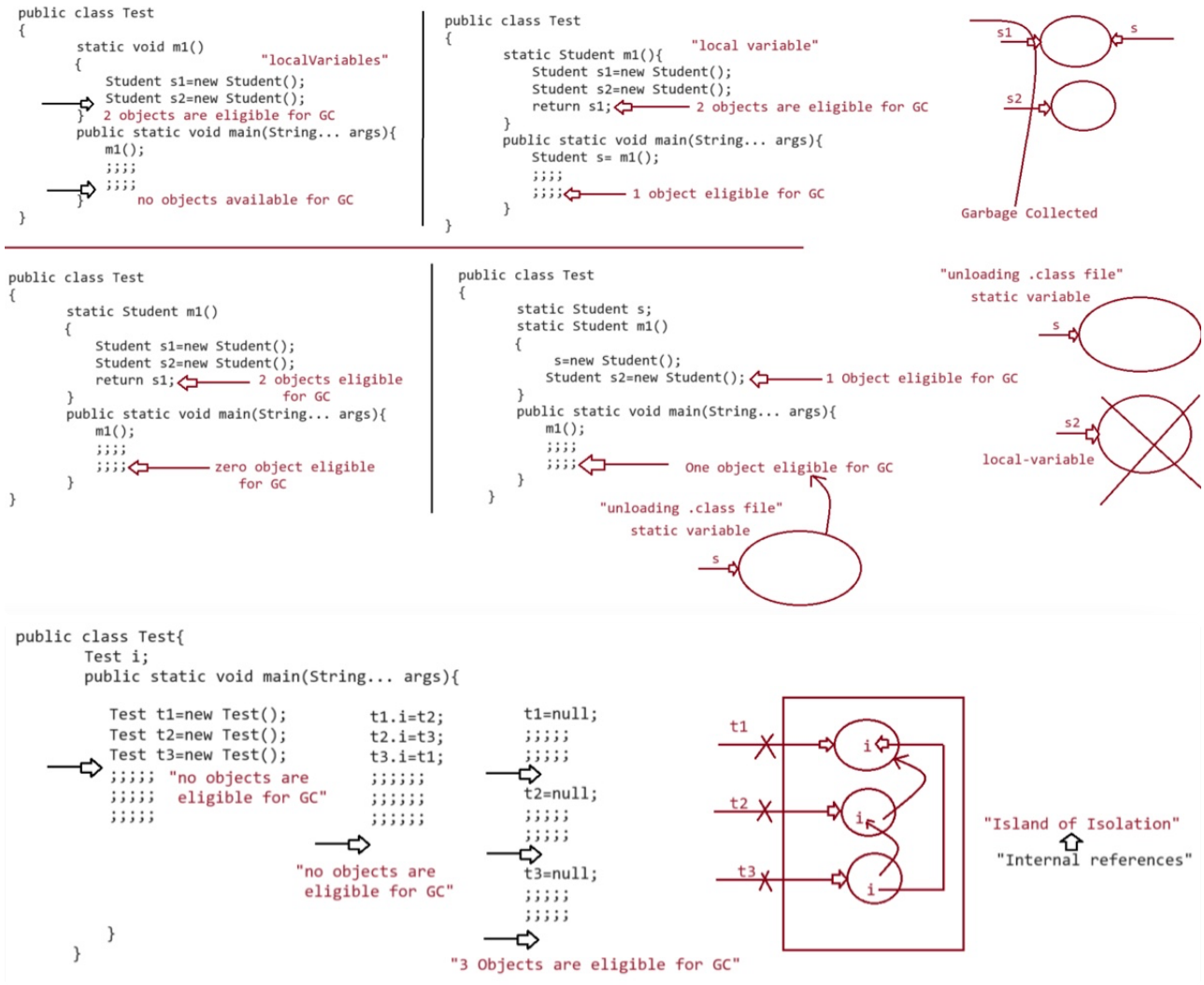
=> Even through programmer is not responsible for destruction of objects but it is always a good programming practice to make an object eligible for GC if it is no longer required.

**=> An object is eligible for GC if and only if it does not have any references.**




**In the second part(above image), the object that s1 points to in first line will be eligible for gc when we assign a new object to s1. Similarly for s2.**




**Only at the end of main all objects will be eligible for gc because of internal references which is also know as Island of Isolation**

**Note:** gc is not directly called once the object is eligible for gc . If this is called continously then performance will be low.

**eg#1**

```
import java.util.Date;

public class Test

{

    public static void main(String[] args)throws Exception

    {

        /*

            public native long freeMemory();

            public native long totalMemory();

            public native long maxMemory();

        */
```

```
        Runtime r = Runtime.getRuntime(); //getRuntime is a static method
        System.out.println("Total memory on heap is :: "+r.totalMemory());
        System.out.println("Free memory on heap is :: "+r.freeMemory());
        for (int i =0;i<10000 ; i++)
        {
            Date d = new Date();
            d = null;
        }
        System.out.println("Free memory on heap is :: "+r.freeMemory());
        r.gc();
        System.out.println("Free memory on heap is :: "+r.freeMemory());
    }
}
```

**Output**

**Total memory on heap is :: 126877696**

**Free memory on heap is :: 125535480**

**Free memory on heap is :: 124859112**

**Free memory on heap is :: 125911304**

**Note : Runtime class is a singleton class(try to compare the hashcode of two objects of Runtime they will have same value as it is a singleton class)** so not create the object to use constructor.

**Singleton Class:**

- A singleton class ensures that only one instance of that class exists throughout the entire Java application.
- This is typically achieved by making the constructor private and providing a static method to access the single instance.

**`Runtime` Class as a Singleton:**

- The `Runtime` class in Java follows the singleton pattern.
- Its constructor is private, preventing you from directly creating objects using `new Runtime()`.
- Instead, it provides the static `getRuntime` method to retrieve the single instance.

**Benefits of Singleton Pattern for `Runtime`:**

- There's only one `Runtime` object needed to manage the system's resources.
- Using `getRuntime` ensures you're always interacting with the same instance, promoting consistency.

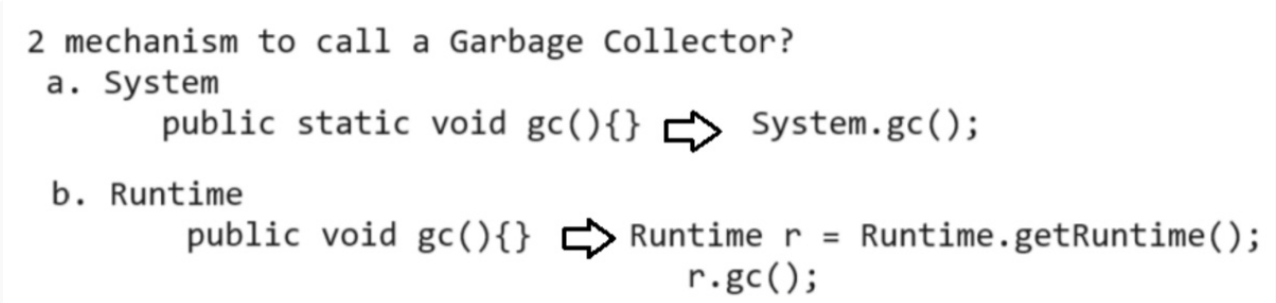
**Which of the following are valid ways for requesting jvm to run GC ?**

System.gc(); (valid)

Runtime.gc(); (invalid)

(new Runtime).gc(); (invalid)

Runtime.getRuntime().gc(); (valid)




**Note:** gc() method present in System class is static, where as it is instance method in Runtime class

**Note:** Over Runtime class gc() method , System class gc() method is recommended to use because of the following reason.

```
public class System{
    public static void gc(){
        Runtime.getRuntime().gc()
    }
}
```

This is because internally System class also uses Runtime class gc.

**Hence Runtime.getRuntime().gc() is best suited as inside of System.gc() same method is called.**

**Note:** In java it is not possible to find size of an object and address of an object.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Student std = new Student();

**Q> what std holds?**

**ans. Address(wrong)(a variable which holds address is know as pointer and in java we dont have a concept of pointers)**

**ans. object reference(internally hashCode value)**

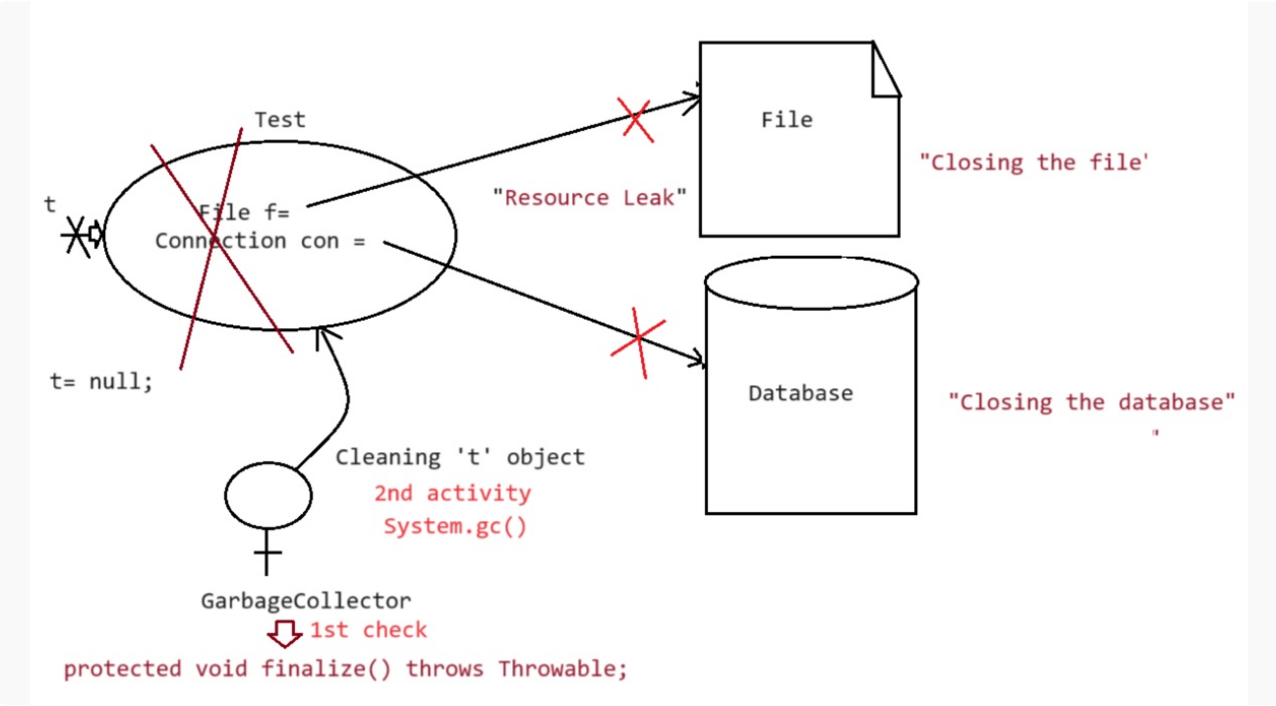
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Finalization:**

=> Just before destroying any object gc always calls finalize() method to perform cleanup activities.

**=> If the corresponding class contains finalize() method then it will be executed otherwise Object class finalize() method will be executed,which is declared as follows.**

protected void finalize() **throws Throwable**




**Memory Leak → when 't' is destroyed but the resources(file,db) it was using are not closed.This is resolved by "finalize".**

**Case 1:**

Just before destroying any object GC calls finalize() method on the object which is eligible for GC then the corresponding class finalize() method will be executed.

**For Example** if String object is eligible for GC then String class finalize()method is executed but not Test class finalize()method.

**eg#1.**

```
public class TestApp{
    public static void main(String... args){
        String s=new String("sachin");
        s=null;
        Runtime.getRuntime().gc();
        System.out.println("end of main()");
    }
    public void finalize() {
        System.out.println("finalized method called..");
    }
}
```

**In the above program String class finalize()method got executed. Which has empty implementation.**

**If we replace String object with Test object then Test class finalize() method will be executed.**

**eg#2.**

```
public class TestApp{
    public static void main(String... args){
        String s=new String("sachin");
        s=null;
        TestApp t =new TestApp();
        t=null;
        Runtime.getRuntime().gc();
        System.out.println("end of main()");
    }
    public void finalize() {
        System.out.println("finalized method called..");
    }
}
```

**case2:**

**We can call finalize() method explicitly then it will be executed just like a normal method call** and **object won't be destroyed.** But before destroying any object GC always calls finalize() method.

**eg#1.**

```
public class TestApp{
    public static void main(String... args){
        TestApp t =new TestApp();
        t.finalize();    //Explicitly we are calling so object wont be cleaned
        t.finalize();
        t=null;
        System.gc();
        System.out.println("End of main()");
    }
    public void finalize() {
        System.out.println("finalized method called..");
    }
}
```

**Output:**

**finalize() method called.**

**finalize() method called.**

**finalize() method called.**

**End of main.**

**In the above program finalize() method got executed 3 times in that 2 times explicitly(object will not be destroyed) by the programmer and one time(object will be destroyed here) by the gc.**

**Note: In servlet coding**

1. init() => At the time of Servlet Initialisation.
2. service() => For every request it will be called.
3. destroy() => At the time of servlet destroy object by container.

**Life cycle methods of Servlet**

```
init(){
    destroy();  //just like normal methods call will be made no servlet deinstantion.
}
service(){
    destroy();
}
destroy(){
    //logic of deinstantiation
}
```

**Case 3:**

finalize() method can be call either by the programmer or by the GC .

**=> If the programmer calls explicitly finalize() method and while executing the finalize() method if an exception raised and uncaught then the program will be terminated abnormally.**

**=> If GC calls finalize() method and while executing the finalize()method if an exception raised and uncaught then JVM simply ignores that exception and the program will be terminated normally**

**eg#1.**

```
public class TestApp{
    public static void main(String... args){
        TestApp t =new TestApp();
        t.finalize();  //line1
        t=null;
        System.gc();
        System.out.println("End of main()");
    }
    public void finalize() {
        System.out.println("finalized method called..");
        System.out.println(10/0);
    }
}
```

=> If we un-comment line1 then programmer calling finalize() method explicitly and while executing the finalize()method ArithmeticException raised which is uncaught hence the program terminated abnormally.

=> If we un-comment line1 then GC calls finalize() method and JVM ignores ArithmeticException and program will be terminated normally.

**Garbage Collector**

**Different ways to make Garbage Object eligible for GC**

a. Nullifying the Object

b. ReUsing the same reference

c. Objects created inside the method

d. Island of Isolation

**Different ways of Calling Garbage Object**

a. System.gc()

b. Runtime.getInstance().gc() [best suited to call GC]

**Finalization**

=> Garbage Collector before cleaning the object,it would internally call finalize() to clean the reference associated with the object to avoid "Memory Leaks".

=> protected void finalize() throws Throwable

**Case4: On a particular Object, JVM will call finalize() only once.**

```
public class Test
{
    static Test t;
    public static void main(String[] args)throws Exception
    {
        Test t1 =new Test();
        System.out.println("T1 HASHCODE :: "+t1.hashCode());
        t1 =null;
        System.gc();     //Called GC -> finalize()

        Thread.sleep(2000);

        System.out.println("T HASHCODE :: "+t.hashCode());
        t = null;
        System.gc();    //Called GC
        Thread.sleep(2000);
        System.out.println("End of main method...");
    }
    @Override
```

```
    public void finalize()
    {
        System.out.println("finalized method called...");
        t = this;
    }
}
```

**Output**

**T1 HASHCODE :: 366712642**

**finalized method called...**

**T HASHCODE :: 366712642**

**End of main method...**

**Note:**

The behavior of the GC is vendor dependent and varied from JVM to JVM hence we can't expect exact answer for the following

1. What is the algorithm followed by GC.(mark and sweep is the common algorithm)
2. **Exactly at what time JVM runs GC.**
3. **In which order GC identifies the eligible objects.**
4. **In which order GC destroys the object etc.**
5. **Whether GC destroys all eligible objects or not.**

=> When ever the program runs with low memory then the JVM runs GC, but we can't except exactly at what time.

=> Most of the GC's followed mark & sweap algorithm , but it doesn't mean every GC follows the same algorithm.

**eg#1.**

```
public class Test
{
    static int counter =0;
    public static void main(String... args){
        for (int i=1;i<=10000000; i++){
            Test t =new Test();
            t=null;
        }
    }
    public void finalize() {
        System.out.println("finalized method called:: "+ (++counter));
    }
```

}

**Output varied based on i value if we keep increasing to 100,1000,10000,100000,1000000,....**

**Memory leaks:**

**=> An object which is not using in our application and it is not eligible for GC such type of objects are called "memory leaks".**

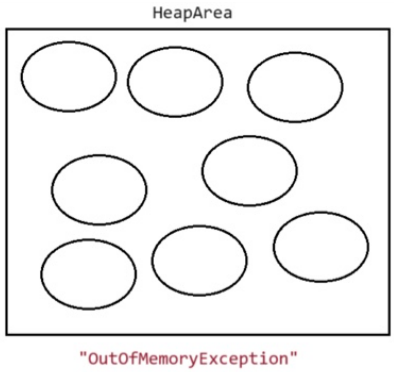
=> In the case of memory leaks GC also can't do anything the application will be crashed due to memory problems.

=> In our program if memory leaks present then certain point we will get OutOfMemoryException. Hence if an object is no longer required then it's highly recommended to make that object eligible for GC.

```
Student s1 =new Student();
Student s2 =new Student();
Student s3 =new Student();
Student s4 =new Student();
Student s5 =new Student();
Student s6 =new Student();

   ;;;;;;;
   ;;;;;;;
Student s100 =new Student();
   ;;;;;;;
   ;;;;;;;
Student s1000 = new Student();
   ;;;;;;;
   ;;;;;;;
Student s10000 = new Student();
   ;;;;;;;
   ;;;;;;;

"Application Crash"
```




```
"Memory Leak"


GC can it come and clean the object
Ans. No

    Tools to monitor Memory Leaks
     a. HPJ Meter
     b. HP OVO
     c. J Probe
     d. Patrol ....
```

**eg#1.**

Student s1=new Student();

Student s2=new Student();

Student s3=new Student();

;;;;

;;;;

;;;;

Student s10000000 =new Student();

;;;;

;;;;

;;;;

**//program crash =>memory leak(In this case JVM and Garbage Collector can't do anything)**

**OutOfMemoryError**

**=> By using monitoring tools we can identify memory leaks.**

HPJ meter

HP ovo

IBM Tivoli These are monitoring tools.

J Probe (or memory management tools)

Patrol and etc

**Note:**

**What is the difference b/w final,finally and finalize?**

**Ans.**

**final =>** It is an access modifer applicable on class,method and varaible

**class ->** inheritance is not possible.

**method ->** we can't override.

**variable ->** we can't change the value for the variable during execution[Compile Time constant].

**finally ->** It is a block of code which gets executed irrespective of whether exception occurs or not in java application

```
try{
    //risky code
}catch(XXXX e){
    // handling exception
}finally{
    //resource releasing logic
}
```

**finalize() ->** It is a method which gets called automatically by the GC, to perform clean up activities associated with the Objects to avoid memory leaks.

protected void finalize() throws Throwable

**Serialization Vs DeSerialization**

1. **Serialization =>** Converting java object to n/w supported of file supported format.
   FileOutpuStream,ObjectOutputStream(writeObject(obj))
2. **DeSerialization =>** Converting n/w or file supported format object to java object.
   FileInputStream,ObjectInputStream(readObject())

**Note:** When we try to Serialize the object, if the object is not in Serializable format then it would result in "NotSerializableException".

To make the object Serializable, we need to implement an interface called "Serializable"[Marker Interface].

**Object Graph in Serialization**

1. **Whenever we are serializing an object the set of all objects which are reachable from that object will be serialized automatically.**
   **This group of objects is nothing but object graph in serialization.**

2. In object graph every object should be Serializable otherwise we will get runtime exception saying "**NotSerializableException**".

**eg#1.**

```
import java.io.*;
class Dog implements Serializable
{
    Cat c = new Cat();
}
class Cat implements Serializable
{
    Rat r = new Rat();
}
class Rat implements Serializable
{
    int j = 20; //serialization is olny for refrence type not for premitive types
}
public class Test
{
    public static void main(String... args) throws Exception{
    Dog d1 = new Dog();
    System.out.println("Serialization Started...");
    //Performing Serialization
    new ObjectOutputStream(
        new FileOutputStream("Dog.ser")).writeObject(d1);
    System.out.println("End of Serialization...");

    System.out.println();

    System.out.println("De-Serialization Started...");
    //Performing Serialization
    Dog d2= (Dog)new ObjectInputStream(
        new FileInputStream("Dog.ser")).readObject();
    System.out.println("Value of j is :: "+d2.c.r.j);
    System.out.println("End of Serialization...");
    }
}
```

**Output**

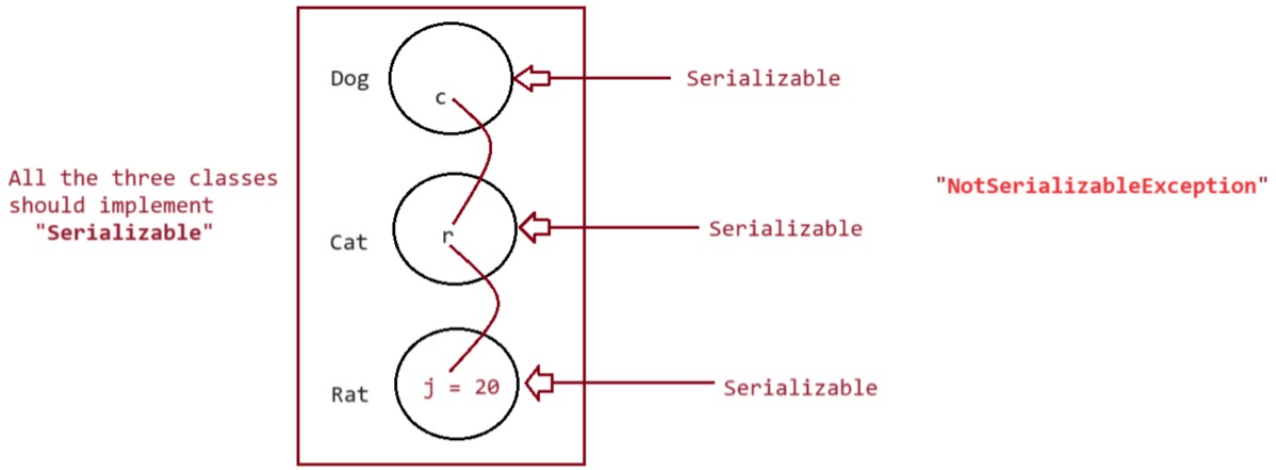
**Serialization Started...**

**End of Serialization...**

**De-Serialization Started...**

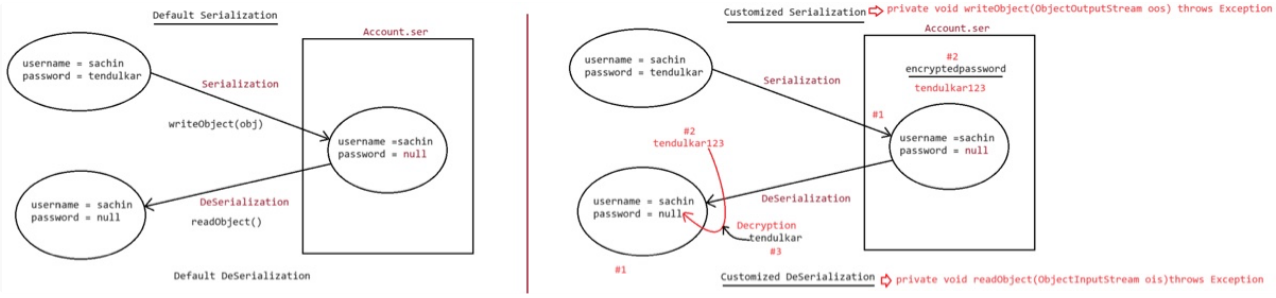
**Value of j is :: 20**

**End of Serialization...**




**CustomizedSerialization**

During default Serialization there may be a chance of lose of information due to transient keyword.




**eg#1.**

```
import java.io.Serializable;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
class Account implements Serializable{
    String name="sachin";
```

```
    transient String password="tendulkar";
}
public class Test {
    public static void main(String[] args)throws IOException,ClassNotFoundException{
        Account acc=new Account();
        System.out.println(acc.name +" => "+ acc.password);
        System.out.println("Serialization Started");
        FileOutputStream fos= new FileOutputStream("abc.ser");
        ObjectOutputStream oos=new ObjectOutputStream(fos);
        oos.writeObject(acc);
        System.out.println("Serialization ended");
        System.out.println("***********************************");
        System.out.println("DeSerialization Started");
        FileInputStream fis= new FileInputStream("abc.ser");
        ObjectInputStream ois=new ObjectInputStream(fis);
        acc=(Account)ois.readObject();
        System.out.println(acc.name +" => "+ acc.password);
        System.out.println("DeSerialization ended");
    }
}
```

=> In the above example before serialization Account object can provide proper username and password.

But after Deserialization Account object can provide only username but not password. This is due to declaring password as transient.

Hence doing default serialization there may be a chance of loss of information due to transient keyword.

**=> We can recover this loss of information by using customized serialization.**

**We can implements customized serialization by using the following two methods.**

1. private void writeObject(ObjectOutputStream os) throws Exception.

   => **This method will be executed automatically by jvm at the time of serialization.**

   **=> It is a callback method(no @override is written over writeObject and readObject). Hence at the time of serialization if we want to perform any extra work we have to define that in this method only. (prepare encrypted password and write encrypted password seperate to the file )**

2. private void readObject(ObjectInputStream is) throws Exception.

   => This method will be executed automatically by JVM at the time of Deserialization.

   =>Hence at the time of Deserialization if we want to perform any extra activity we have to define that in this method only. (read encrypted password , perform decryption and assign decrypted password to the current object password variable )

**eg#1.**

```
import java.io.Serializable;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
class Account implements Serializable{
    String name="sachin";
    transient String password="tendulkar";
    private void writeObject(ObjectOutputStream oos)throws Exception{
        oos.defaultWriteObject();     //performing default Serialization
        String epwd="123"+password;    //performing encryption
        oos.writeObject(epwd);    //write the encrypted data to file(abc.ser)
    }
    private void readObject(ObjectInputStream ois)throws Exception{
        ois.defaultReadObject();  //performing default Serialization
        String epwd=(String)ois.readObject();  //performing decryption
        password=epwd.substring(3);  //writing the extra data to Object
    }
}
public class Test {
public static void main(String[] args)throws IOException,ClassNotFoundException{
    Account acc=new Account();
    System.out.println(acc.name +" => "+ acc.password);
    System.out.println("Serialization Started");
    FileOutputStream fos= new FileOutputStream("abc.ser");
    ObjectOutputStream oos=new ObjectOutputStream(fos);
    oos.writeObject(acc);
    System.out.println("Serialization ended");
    System.out.println("**********************************");
    System.out.println("DeSerialization Started");
    FileInputStream fis= new FileInputStream("abc.ser");
    ObjectInputStream ois=new ObjectInputStream(fis);
    acc=(Account)ois.readObject();
    System.out.println(acc.name +" => "+ acc.password);
    System.out.println("DeSerialization ended");
    }
```

}

**=> At the time of Account object serialization JVM will check is there any writeObject() method in Account class or not.**

**=> If it is not available then JVM is responsible to perform serialization(default serialization).**

**=> If Account class contains writeObject() method then JVM feels very happy and executes that Account class writeObject() method. The same rule is applicable for readObject() method also.**

**Try for the following data**

```
class Account implements Serializable{
    String name="sachin";
    transient String password="tendulkar";
    transient int pin=4444;
    private void writeObject(ObjectOutputStream oos)throws Exception{
        oos.defaultWriteObject();  //performing default Serialization
        String epwd="123"+password;   //performing encryption
        int epin=1234+pin;   //performing encryption
        oos.writeObject(epwd);   //write the encrypted data to file(abc.ser)
        oos.writeInt(epin);   //write the encrypted data to file(abc.ser)
    }
    private void readObject(ObjectInputStream ois)throws Exception{
        ois.defaultReadObject();//performing default Serialization
        String epwd=(String)ois.readObject();//performing decryption
        int epin=ois.readInt();//performing decryption
        password=epwd.substring(3);//writing the extra data to Object
        pin=epin-1234;//writing the extra data to Object
    }
}
public class Test {
    public static void main(String[] args)throwsIOException,ClassNotFoundException{
        Account acc=new Account();
        System.out.println(acc.name +"=> "+ acc.password+"==>"+acc.pin);
        System.out.println("Serialization Started");
        FileOutputStream fos= new FileOutputStream("abc.ser");
        ObjectOutputStream oos=new ObjectOutputStream(fos);
        oos.writeObject(acc);
        System.out.println("Serialization ended");
        System.out.println("**********************************");
        System.out.println("DeSerialization Started");
        FileInputStream fis= new FileInputStream("abc.ser");
```

```
        ObjectInputStream ois=new ObjectInputStream(fis);
        acc=(Account)ois.readObject();
        System.out.println(acc.name +"=> "+ acc.password+">"+acc.pin);
        System.out.println("DeSerialization ended");
    }
}
```

**Output**

**sachin => tendulkar ==>4444**

**Serialization Started**

**Serialization ended**

---

**DeSerialization Started**

**sachin => tendulkar >4444**

**DeSerialization ended**

**}**

**Serialization**

=> Converting the java object to file/network supported format

=> To implement Serialziation, we use "Serializable".

=> If the object doesnot support Serialization it would result in "NotSerializableException".

=> Streams used for Serialization

```
new ObjectOutputStream(new FileOutputStream(".ser")).writeObject(object)
```

**DeSerialization**

=> Converting the file/network supported object format to java supported Object format

=> To implement DeSerialziation also the class should implements "Serializable".

=> If the object doesnot support De-Serialization it would result in"NotSerializ ableException".

=> Streams used for De-Serialization

```
new ObjectInpuStream(new FileInputStream(".ser")).readObject()
```

**Usage of tranisent keyword in Serialziation and DeSerialization**

**a. transient =>** variable value won't participate in serialization rather default value will be stored in serialized file.

**Note:**

**Combination of final, static vs transient keyword**

**a. final transient =>** final means variable won't participate value will participate so transient on final variables has no impact.

**b. static transient =>** static means class variables, so during serialization only instance variable will participate so transient on static variables has no impact.

**We can serialize any no of objects,but during DeSerialization order of Serialized objects is important otherwise it would result in "ClassCastException".**

**=> Object graph in serialization**

a. Whenever we serialize any object, the set of all the objects which are reachable from that object will be serialized automatically. this group is nothing but object graph in serialization.

b. In object graph every objet must be serialized otherwise it would result in "NotSerializableException".

**=> Customized Serialization**

By default when we do serialization all the instance variable will be serialized, if the instance variable is marked with trasient keyword then variable value won't be serialized, As a result of which the there would be loss of data at the time of DeSerialization, to resolve this problem we need to go for "Customized Serialization".

**-> method used for Customized Serialization and DeSerialization**

```
private void writeObject(ObjectOutputStream oos) throw Exception{
    //Default serialization
    oos.defaultWriteObject();
    //customization
    oos.writeObject(),oosWriteInt(),oosWriteFloat(),.....
}
private void readObject(ObjectInputStream ois) throw Exception{
    //Default serialization
    ois.defaultReadObject();
    //customization
    ois.readObject(),ois.readInt(),ois.readFloat(),.....
}
```

**Serialization w.r.t inheritance**

**Case 1:**

**If parent class implements Serializable then automatically every child class by default implements Serializable.**

That is Serializable nature is inheriting from parent to child. Hence even though child class doesn't implements Serializable, we can serialize child class object if parent class implements serializable interface.

**eg#1.**

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

```
class Animal implements Serializable
{
    int i = 10;
}
class Dog extends Animal
{
    int j = 20;
}
public class Test
{
    public static void main(String... args) throws Exception{
        Dog d1 = new Dog();
        new ObjectOutputStream(
            new FileOutputStream("Dog.ser")).writeObject(d1);
        Dog d2 = (Dog)new ObjectInputStream(
                    new FileInputStream("Dog.ser")).readObject();
        System.out.println("Dog object :: "+d2.i+"....."+d2.j);
    }
}
```

**Output**

**Dog object** **Dog object** **10.....20**

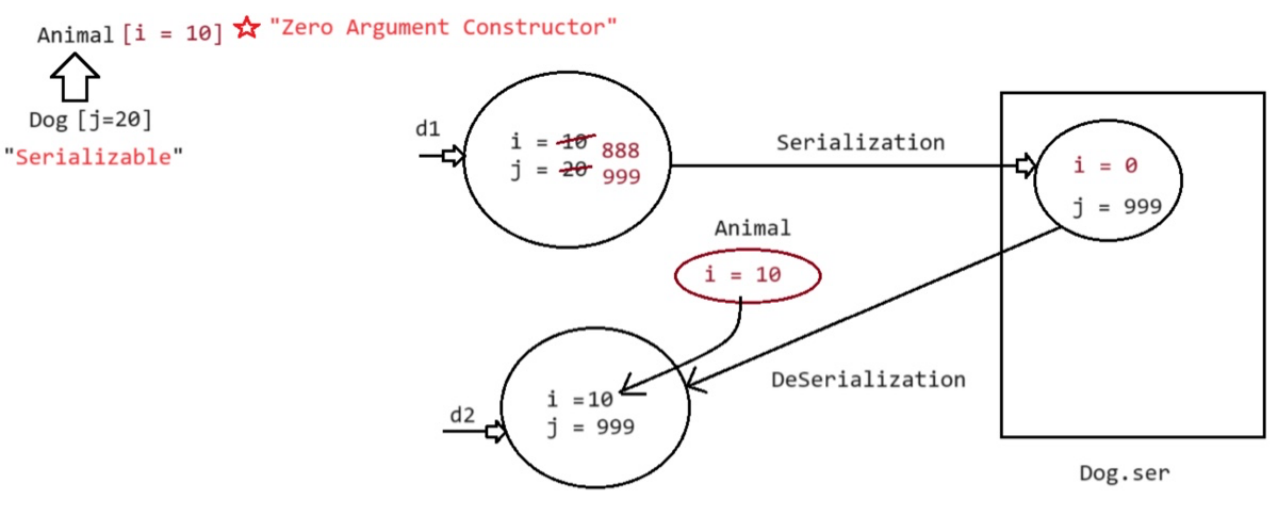
**Does Object class implements Serializable?**

**Ans. No, Because every Object we will not be sent to the network or to the file for storage purpose,so Object class doesn't implement Serializable interface.**

**Case 2:**

1. Even though parent class does not implements Serializable we can serialize child object if child class implements Serializable interface.
2. At the time of serialization JVM ignores the values of instance variables which are coming from non Serializable parent then instead of original value JVM saves default values for those variables to the file.
3. **At the time of Deserialization JVM checks whether any parent class is non Serializable or not. If any parent class is nonSerializable JVM creates a separate object for every non Serializable parent and shares its instance variables to the current object.**
4. **To create an object for non-serializable parent JVM always calls no arg constructor (default constructor) of that non-Serializable parent hence every non Serializable parent should compulsory contain no arg constructor otherwise we will get runtime exception "InvalidClassException".**

5. If non-serializable parent is abstract class **then just instance control flow will be performed** and share it's instance variable to the current object.




**eg#1.**

```
import java.io.*;
abstract class Animal
{
    int i = 10;
    Animal(){
        System.out.println("Animal constructor called");
    }
}
class Dog extends Animal implements Serializable
{
    int j = 20;
    Dog(){
        System.out.println("Dog constructor called");
    }
}
public class Test
{
    public static void main(String... args) throws Exception{
    Dog d1 = new Dog();
    d1.i = 888;
    d1.j = 999;
    System.out.println("Serialization started....");
    new ObjectOutputStream(
```

```
                new FileOutputStream("Dog.ser")).writeObject(d1);
        System.out.println("Serialization ended....");
        System.in.read();
        Dog d2 = (Dog)new ObjectInputStream(
                        new FileInputStream("Dog.ser")).readObject();
        System.out.println("Dog object :: "+d2.i + "........"+d2.j);
        }
}
```

**Output**

**Dog constructor called**

**Animal constructor called**

**Serialization started....**

**Serialization ended....**

**Animal constructor called**

**Dog object :: 10.....999**

**Externalization : ( 1.1 v )**

1. In default serialization every **thing takes care by JVM and programmer doesn't have any control.**
2. In serialization total object will be saved always and it is not possible to save part of the object, which creates performance problems at certain point.
3. To overcome these problems we should go for externalization where **every thing takes care by programmer and JVM doesn't have any control.**
4. The main advantage of externalization over serialization is we can save either total object or part of the object based on our requirement.
5. **To provide Externalizable ability for any object compulsory the corresponding class should implements externalizable interface.**
6. **Externalizable interface is child interface of serializable interface.**

**Externalizable interface defines 2 methods :**

1. writeExternal(ObjectOutput out ) throws IOException
2. readExternal(ObjectInput in) throws IOException,ClassNotFoundException

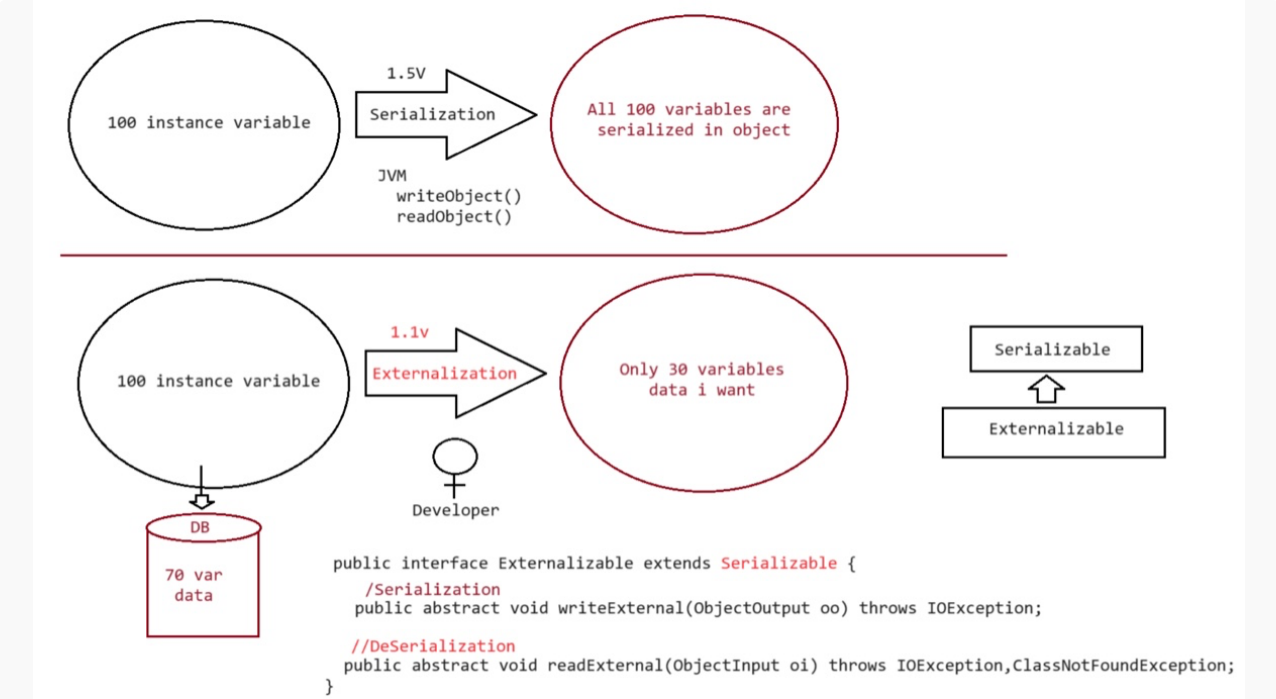
**public void writeExternal(ObjectOutput out) throws IOException**

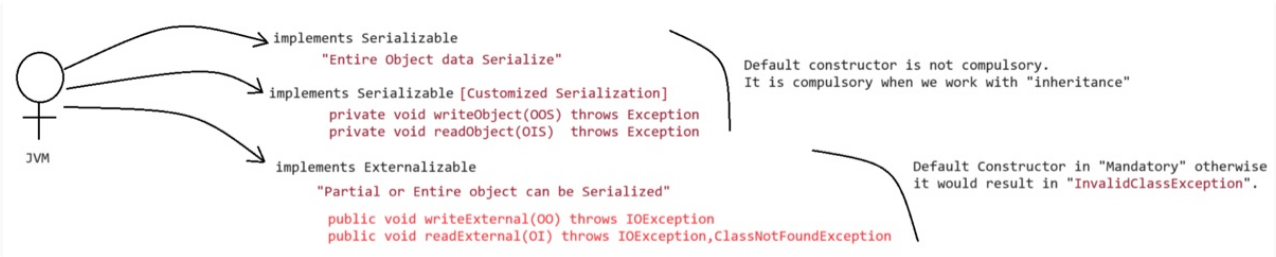
This method will be executed automaticcay at the time of Serialization with in this method , we have to write code to save required variables to the file .

**public void readExternal(ObjectInput in) throws IOException,ClassNotFoundException**

This method will be executed automatically at the time of deserialization with in this method , we have to write code to save read required variable from file and assign to the current object.

At the time of deserialization JVM will create a seperate new object by executing public no-arg constructor on that object JVM will call readExternal() method.




**eg#1.**

```
import java.io.*;
class Dog implements Externalizable
{
    //instance variable
    String s;
    int i;
    int j;
    //parameterized constructor
    Dog(String s,int i, int j){
        this.s = s;
        this.i = i;
        this.j = j;
    }
```

```
    public Dog()
    {
        //To avoid InvalidClassException during "DeSerialization"
        System.out.println("Dog constructor called...");
    }
    //Serialization
    public void writeExternal(ObjectOutput oo) throws IOException
    {
        System.out.println("Serializing the required fields of the Object");
        oo.writeObject(s);
        oo.writeInt(i);
    }
    //DeSerialization
    public void readExternal(ObjectInput in) throws IOException,ClassNotFoundException
    {
        System.out.println("DeSerializing the required fields of the Object");
        s = (String)in.readObject();
        i = in.readInt();
    }
}
public class Test
{
    public static void main(String... args) throws Exception{
    Dog d1 = new Dog("bruno",10,15);
    System.out.println("Dog Object :: "+d1.s+"...."+d1.i+"...."+d1.j);
    System.out.println("Serialization started....");
    new ObjectOutputStream(
        new FileOutputStream("Dog.ser")).writeObject(d1);
    System.out.println("Serialization ended....");
    System.in.read();
    System.out.println("DeSerialziation Started...");
    Dog d2 = (Dog)new ObjectInputStream(
                new FileInputStream("Dog.ser")).readObject();
    System.out.println("DeSerialziation ended...");
    System.out.println("Dog Object :: "+d2.s+"...."+d2.i+"...."+d2.j);
    }
}
```

**Output**

**Dog Object :: bruno....10....15**

**Serialization started....**

**Serializing the required fields of the Object**

**Serialization ended....**

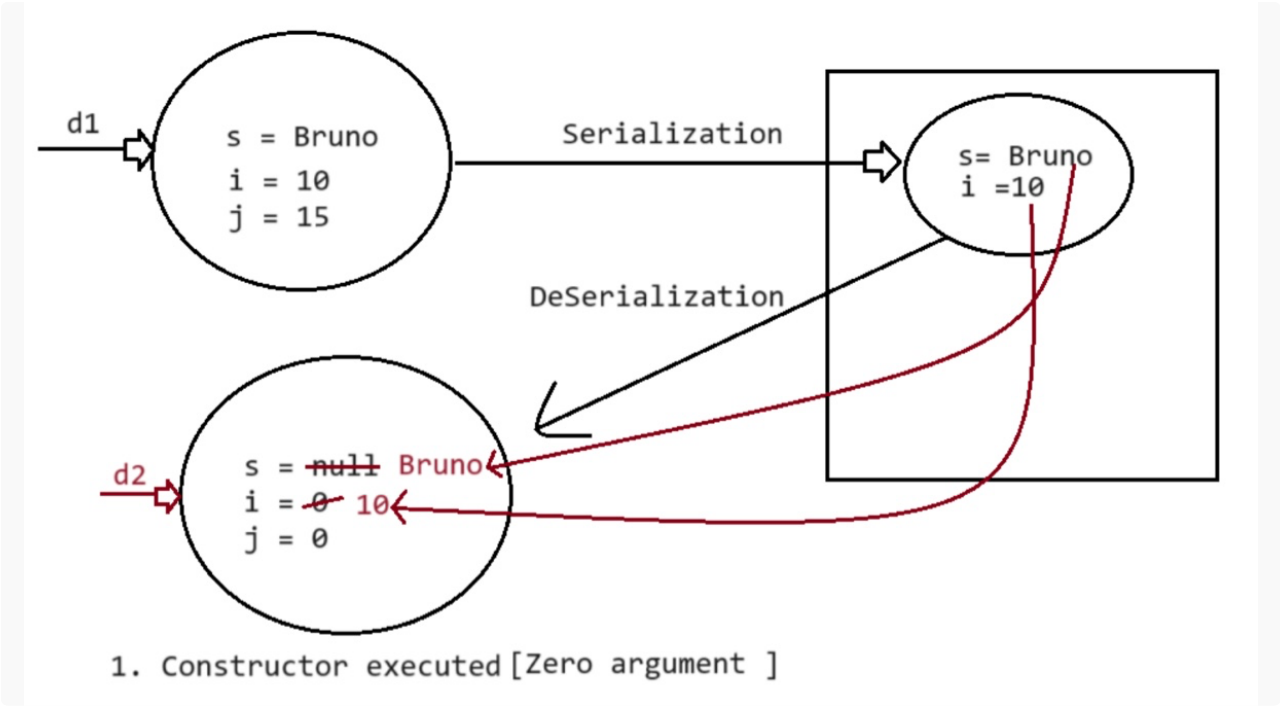
**DeSerialzation Started...**

**Dog constructor called...**

**DeSerializing the required fields of the Object**

**DeSerialzation ended...**

**Dog Object :: bruno....10....0**




**For this reason only zero argument constructor is mandatory in Externalization as we are not sending whole object so during de-serialization first object will be created and will be initialized with its default value and than the constructor will be called to assign the values**

**Externalization is not famous as developer has to do most of things while serialization is more famous in which compiler will do everything.**

**CaseStudy**

1. If the class implements Externalizable interface then only part of the object will be saved in the case output is public no-arg constructor

   **bruno---- 10 ----- 0**

2. If the class implements Serializable interface then the output is

   **bruno --- 10--- 20**

3. **In externalization transient keyword won't play any role , hence transient keyword not required.**

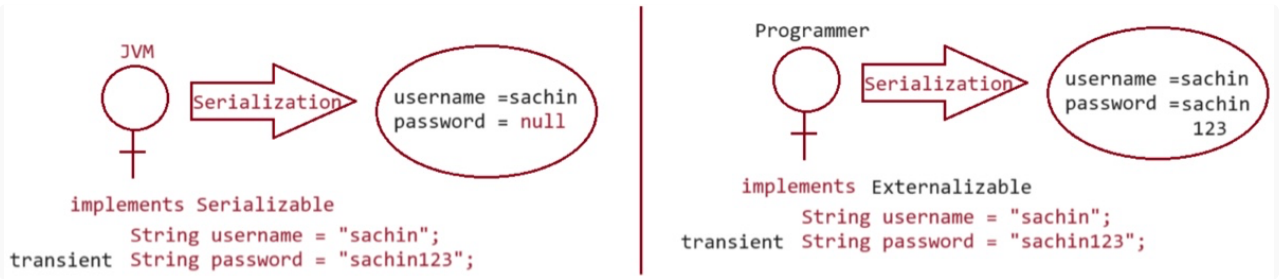
**Difference b/w Serialization and Externalization**

**Serialization**

1. It is meant for default Serialization
2. **Here every thing takes care by JVM and programmer doesn't have any control.**
3. Here total object will be saved always and it is not possible to save part of the object.
4. Serialization is the best choice if we want to save total object to the file.
5. **relatively performence is low.**
6. Serializable interface doesn't contain any method
7. **It is a marker interface.**
8. Serializable class not required to contains public no-arg constructor.
9. transient keyword play role in serialization

**Externalization**

1. It is meant for Customized Serialization
2. Here every thing takes care by programmer and JVM does not have any control.
3. Here based on our requirement we can save either total object or part of the object.
4. Externalization is the best choice if we want to save part of the object.
5. **relatively performence is high**
6. Externalizable interface contains 2 methods :
    1. writeExternal()
    2. readExternal()
7. **It is not a marker interface.**
8. **Externalizable class should compulsory contains public no-arg constructor otherwise we will get RuntimeException saying "InvalidClassException"**
9. **transient keyword don't play any role in Externalization.**




**serialVersionUID**

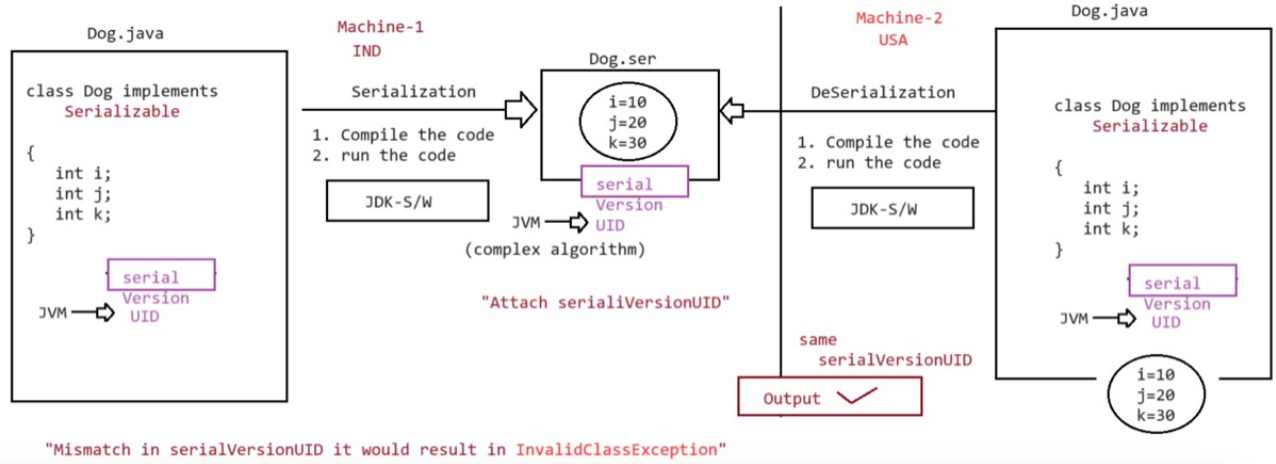
=> To perform Serialization & Deserialization internally JVM will use a unique identifier,which is nothing but serialVersionUID .

=> At the time of serialization JVM will save serialVersionUID with object.

**=> At the time of Deserialization JVM will compare serialVersionUID and if it is matched then only object will be Deserialized otherwise we will get RuntimeException saying "InvalidClassException".**

**The process in depending on default serialVersionUID are :**

1. After Serializing object if we change the .class file then we can't perform deserialization because of mismatch in serialVersionUID of local class and serialized object in this case at the time of Deserialization we will get RuntimeException saying in "InvalidClassException".
2. **Both sender and receiver should use the same version of JVM if there any incompatability in JVM versions then receive anable to deserializable because of different serialVersionUID , in this case receiver will get RuntimeException saying "InvalidClassException".(Sol → dont allow jvm to create it instead we should write code to create it.)**
3. **To generate serialVersionUID internally JVM will use complexAlgorithm which may create performence problems.**




**Make two different files SenderApp.java and ReceiverApp.java and also a separate file for Dog. First only add i and j in dog and run SenderApp and then add a variable k to Dog and now run ReceiverApp, since the source has been changed(so new id will be generated for ReceiverApp and it will not match with SenderApp, hence the error) we will get an error related to serialVersionUID. After deleting .class files for both the apps, now add serialVersionUID to Dog and check the same procedure as done before, now we will not get the error.**

```
import java.io.*;

class Dog implements Serializable

{
    static final long serialVersionUID = 1L;
    int i =10;
    int j =20;
    int k =30;

}

import java.io.*;

class SenderApp

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

```
        System.out.println("Serialization Started****");
        new ObjectOutputStream(
            new FileOutputStream("Dog.ser"))
            .writeObject(new Dog());
        System.out.println("Serialization Ended****");
    }
}
```

**Output**

***De-Serialization Started*****

**Dog Object :: 10....20**

***De-Serialization Ended*****

```
import java.io.*;
class Dog implements Serializable
{
    static final long serialVersionUID = 1L;
    int i =10;
    int j =20;
    int k =30;
}
import java.io.*;
class ReceiverApp
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("De-Serialization Started****");
        Dog d = (Dog)new ObjectInputStream(
            new FileInputStream("Dog.ser"))
            .readObject();
        System.out.println("Dog Object :: "+d.i+"...."+d.j);
        System.out.println("De-Serialization Ended****");
    }
}
```

**Output**

***De-Serialization Started*****

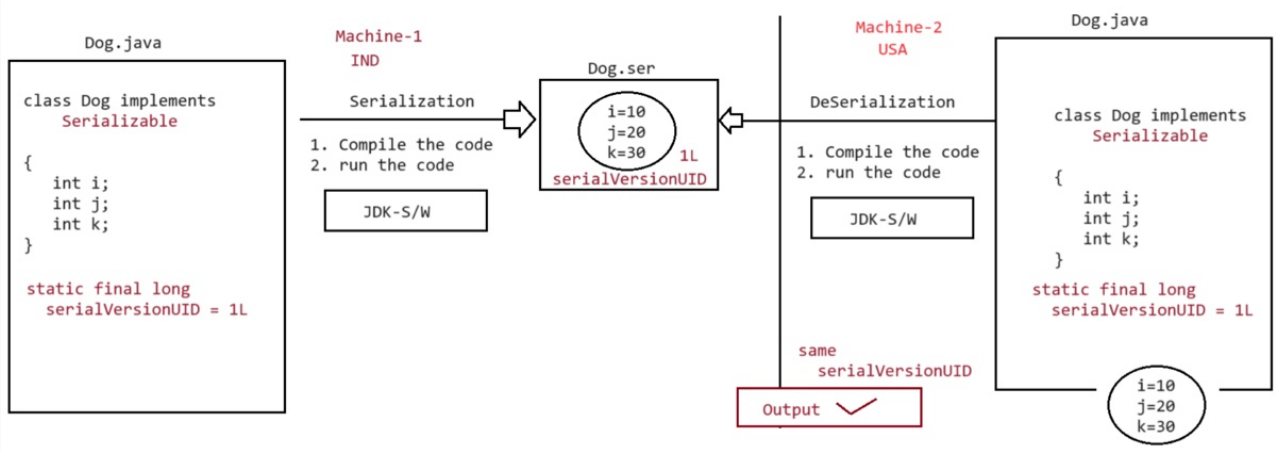
**Dog Object :: 10....20**

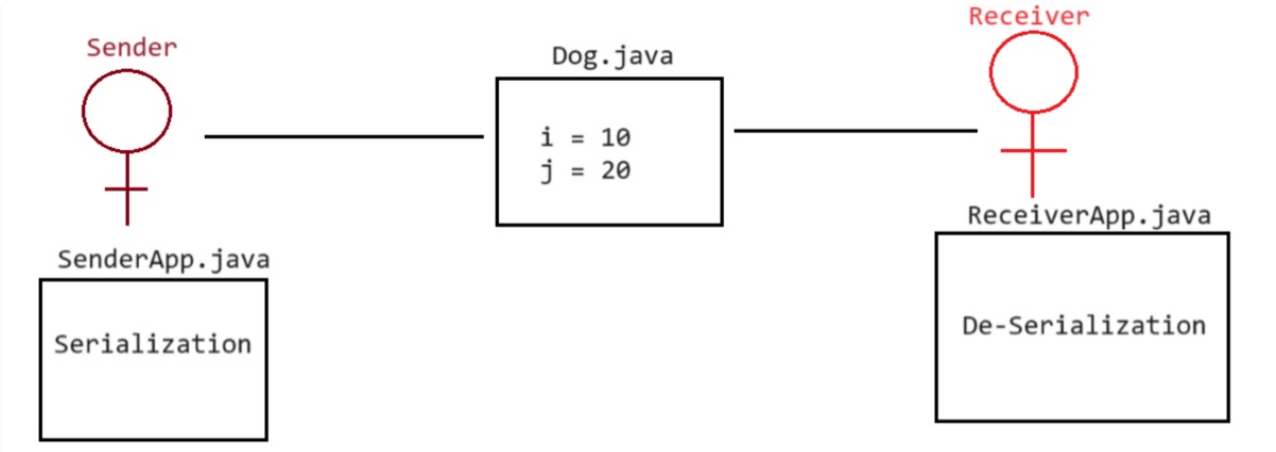
*De-Serialization Ended*****

**KeyPoints**

**=> In the above program after serialization even though if we perform any change to Dog.class file we can deserialize object.**

**=> We can configure our own serialVersionUID both sender and receiver not required to maintain the same JVM versions.**




**Note :** some IDE's generate explicit serialVersionUID

**What are the different ways to create an Object in java?**

a. using new keyword

Test t =new Test();

b. using clonning

class Test implements Cloneable{}

t.clone();

c. using Serialization and DeSerialization

Dog d1 = (Dog)new ObjectInputStream(new FileInputStream("Dog.ser")).readObject();

d. using FactoryDesign pattern

Runtime r = Runtime.getInstance();

e. using class.forName() approach

Test t1 = (Test)Class.forName(Test).newInstance();

**Question**

1. Difference b/w ClassNotFoundException vs NoClassDefFoundError?
2. Difference b/w instanceof vs isInstance()?

**Generics**

The purpose of Generics is

1. To provide TypeSafety.
2. To resolve TypeCasting problems.

**Case1:**

**TypeSafety**

A guranatee can be provided based on the type of elements.

If our programming requriement is to hold only String type of Objects,we can choose String Array.

By mistake if we are trying to add any another type of Objects, we will get "CompileTimeError".

**eg#1.**

String[] s=new String[10000];

s[0] = "dhoni";

s[1] = "sachin";

s[2] = new Integer(10); **//CE: incompatible types found : java.lang.Integer required: java.lang.String**

W.r.t Arrays we can give guratante that what type of elements is present inside Array, hence Arrays are safe to used w.r.t type**,so Arrays as TypeSafety.**

**eg#2.**

ArrayList l=new ArrayList();

l.add("dhoni");

l.add("sachin");

l.add(new Integer(10));

...

....

String s1=(String)l.get(0);

String s2=(String)l.get(1);

String s3=(String)l.get(2)**; //RE: ClassCastException**

For Collections, we can't give gurantee for the type of elements present inside Collection.

If our pgm requirement is to hold only String type of Objects then if we choose ArrayList by mistake if we are trying to add any other type of Object,we won't get any CompileTimeError,**but the program may fail at runtime(business loss).**

**Note: Arrays are TypeSafe,Where as Collections are not TypeSafe.**

Arrays provide guarantee for the type of elements we hold, whereas Collections won't provide gurantee for the type of elements.

**Need of Generics**

1. Arrays to use we need to know the size from the begining,but if we don't the size and still if we want to provide **type casting we need to use "Collections along with Generics".**

**Case2:**

**Type casting**

**eg#1.**

String s[]=new String[3];

s[0] = "sachin";

String name=s[0];**//Typecasting not required as we it holds only String elements only.**

**eg#2.**

ArrayList l=new ArrayList();

l.add("sachin");

String name=l.get(0); **//CE: found : java.lang.Object required: java.lang.String**

String name=**(String)**l.get(0);

**|=>**TypeCasting is compulsory when we work with Collections.

To Overcome the above mentioned problems of Collections we need to go for Generics in 1.5V, which provides TypeSafety and to Resolve TypeCasting problems.

**How TypeSafety is provided in Generics and how it resolves the problems of TypeCasting?**

**eg#1.**

ArrayList al=new ArrayList();**//Non-Generic ArrayList which holds any type of elements.**

al.add("sachin");

al.add("dhoni");

al.add(new Integer(10));

al.add("yuvi");

**eg#2.**

**ArrayList<String>** al=new **ArrayList<String>**(); **//Generic ArrayList which holds only String.**

al.add("sachin");

al.add("dhoni");

al.add(new Integer(10));**//CE**

al.add("yuvi");

**Note: Through Generics TypeSafety is provided.**

ArrayList al**<String>**=new ArrayList**<String>**(); **//Generic ArrayList which holds only String.**

al.add("sachin");

al.add("dhoni");

al.add("yuvi");

String name=al.get(0); **//TypeCasting is not required**

**At the time of retrieval, we are not requried to perform TypeCasting.**

**Note: Through Generics TypeCasting problem is solved.**

Difference b/w

ArrayList l=new ArrayList();

1. It is a nongeneric version of ArrayList Object.
2. **It wont provide TypeSafety as we can add any elements into the ArrayList.**
3. **TypeCasting is required when we retrieve elements.**

ArrayList**<String>** l=new ArrayList**<String>**();

1. It is a generic version of ArrayList Object.
2. **It provides TypeSafety as when we can add only String type of Objects.**
3. **TypeCasting is not required when we retrieve elements.**

**Conclusion1**

**|=>** parameter type

1)- ArrayList<String> al =new ArrayList<String>();

**BaseType** List al =new ArrayList();

2)- Collection<String> al =new ArrayList<String>();

ArrayList<Object> al=new ArrayList<String>()**;//CE:incompatible type: found ArrayList required ArrayList**

**Polymorphism is applicable only for the BaseType(for 1 and 2 Collection and ArrayList),but not for Parameter type.**

**Polymorphism=>** usage of parent reference to hold Child object is the concept of "Polymporphism".

**Conclusion2**

**Collection concept is applicable only for Object,it is not applicable for primitive types. so parameter type should be always be class/interface/enum,if we take primitive it would raise in "CompileTimeError".**

**eg#1.**

ArrayList<int> al=new ArrayList<int>(); **//CE: unexpected type: found int required reference**


**Generic classes**

untill 1.4 version, nongeneric version of ArrayList class is declared as follows

class ArrayList{

boolean add(Object o);**//Argument is Object so no typesafety.**

Object get(int index)**;//return type is Object so type casting is requried.**

}

In 1.5 Version Generic version class is defined as follows

|=> TypeParameter

class ArrayList<T t>{

boolean add(T t);

T get(int index);

}

T => Based on our runtime requirement, T will be replaced with our provided type.

|

|

class ArrayList{

boolean add(String t);**//We can add only String type of Object it provides TypeSafety**

String get(int index)**;//Retrieval Object is always of type String, so TypeCasting not required.**

}

To hold only String type of Object

ArrayList al =new ArrayList

al.add("sachin");

al.add(new Integer(10))**;//CE: can't find symbol method: add(java.lang.Integer) location: class ArrayList**

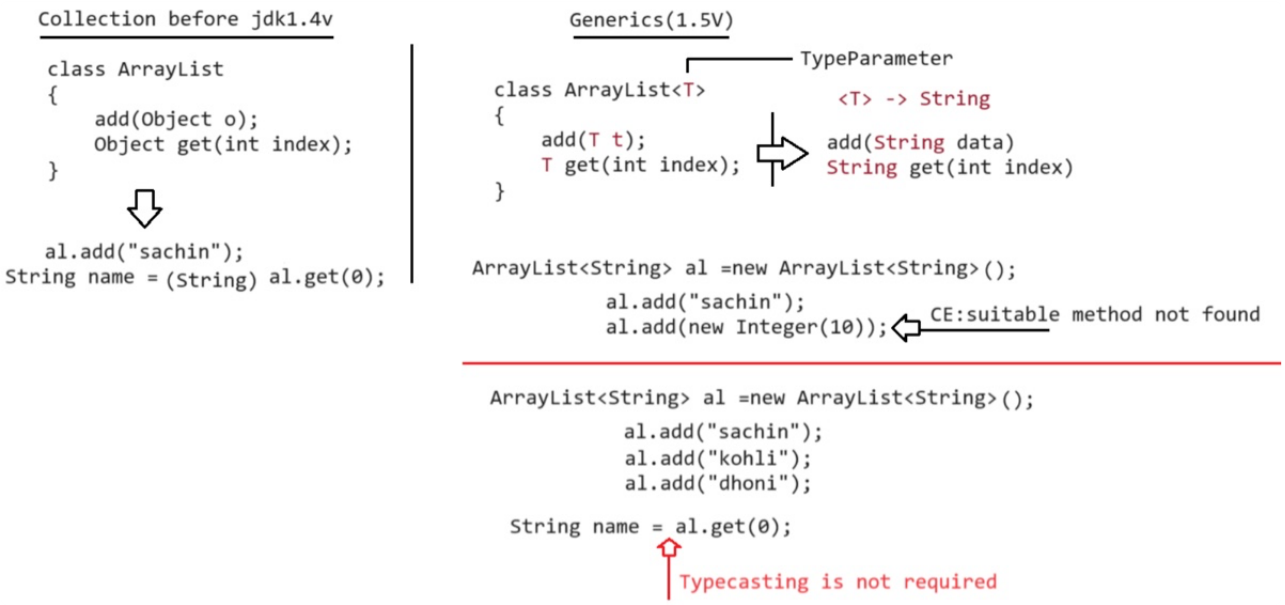
String name=al.get(0);

System.out.println(name);

**Note:**

In Generics we are associating a type-parameter to the class, such type of parameterised classes are nothing but Generic classes.

**Generic class :** class with type-parameter.




**Normal Classes also we can apply Generics**

Based on our requrimenet, we can define our own generic classes also

**eg#1.**

```
class Account{

}

Account a1=new Account();

Account a2=new Account();
```

**eg:**

```
class Gen{
    T obj;
    Gen(T obj){
        this.obj =obj;
    }
    public void show(){
        System.out.println("The type of class is : "+obj.getClass().getName());
    }
    public T getObj(){
        return obj;
    }

}
public class Test {
    public static void main(String[] args) {
        Gen g1=new Gen("sachin");
        g1.show();
```

```
        System.out.println(g1.getObj());
        System.out.println();
        Gen g2=new Gen(10);
        g2.show();
        System.out.println(g2.getObj());
        System.out.println();
        Gen g3=new Gen(10.5);
        g3.show();
        System.out.println(g3.getObj());
    }
}
```

**output**

**The type of class is : java.lang.String**

**sachin**

**The type of class is : java.lang.Integer**

**10**

**The type of class is : java.lang.Double**

**10.5**

**Bounded Types**

We can bound the type parameter for a particular range by using extends keyword such types are called bounded types.

**Example 1:**

class Test<T>

{}

Test <Integer> t1=new Test< Integer>();

Test <Integer> t2=new Test < String>();

Here as the type parameter we can pass any type and there are no restrictions hence it is unbounded type.

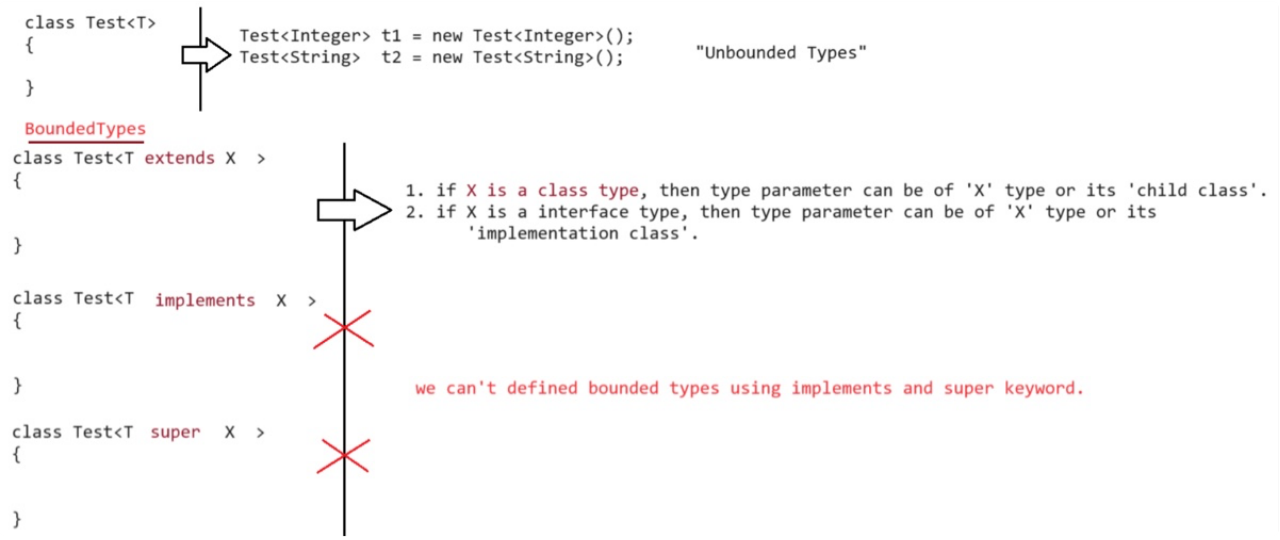
**To specify the bounded types we need to use only extends keyword,we can't use implements and super keyword,but we can replace implements keyword with extends keyword.**

eg: class Test<T extends Number>{}

eg: class Test<T implements Runnable>{}**//invalid**

eg: class Test<T super String>{}**//invalid**

eg: class Test<T extends Runnable>{} **//valid**




**Example 2:**

class Test<T extends X>

{}

If x is a class then as the type parameter we can pass either x or its child classes.

If x is an interface then as the type parameter we can pass either x or its implementation classes

class Test<T extends Number>{}

Test<Integer> t=new Test<Integer>();**//valid**

Test<String> t=new Test<String>();**//CE: Type parameter java.lang.String is not in the boundary.**

class Test<T extends Runnable>{}

Test<Runnable> t=new Test<Runnable>()**;//valid**

Test<Thread> t=new Test<Thread>();**//valid**

Test<String> t=new Test<String>(); **//CE::Type parameter java.lang.String is not within its bound.**

**Note:**

class Test<T extends Number>{} class Test<T extends Runnable>{}

|

**both conditions should be satisified class Test<T extends Number & Runanble>{}**

As the type parameter we can pass any type which extends Number class and implements Runnable interface.

class Test<T extends Number & Runnable>{}

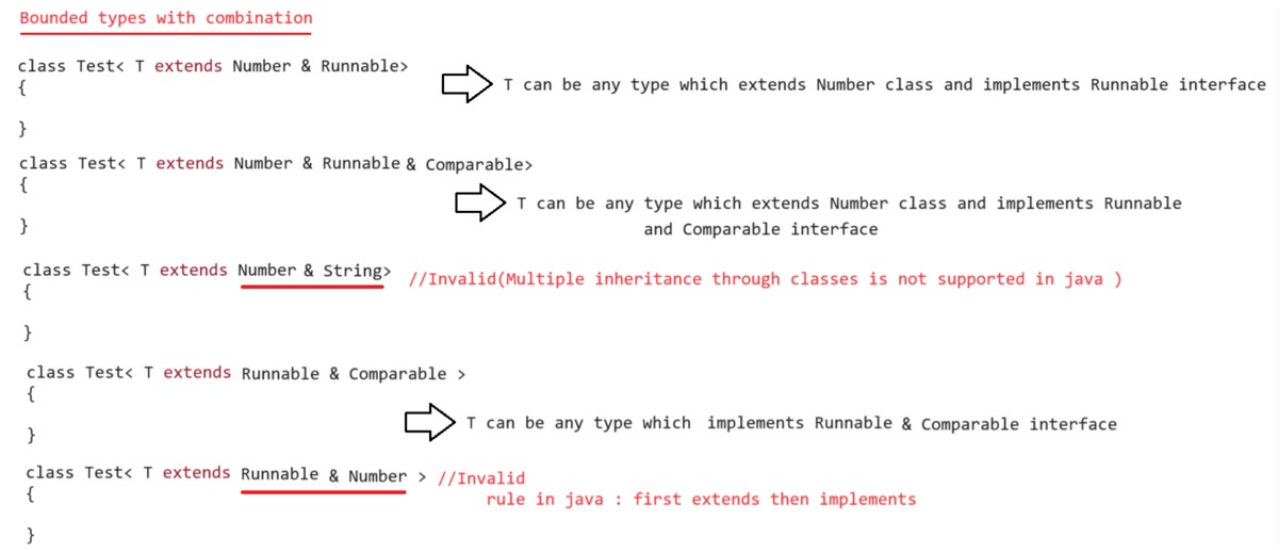
class Test<T extends Runnable & Comparable>{}**//valid**

class Test<T extends Number & Runnable & Comparable>{**}//valid**

class Test<T extends Runnable & Number>{**}//invalid(first class then interface)**

class Test<T extends Number & String>{} **//invalid(mulitple inheritance is not supported)**

class Test<T extends Number & Thread>{} **//invalid(mulitple inheritance is not supported)**




**Conclusions**

class Test<T extends Number>{}**//valid**

class Test<T implements Runnable>{}**//invalid**

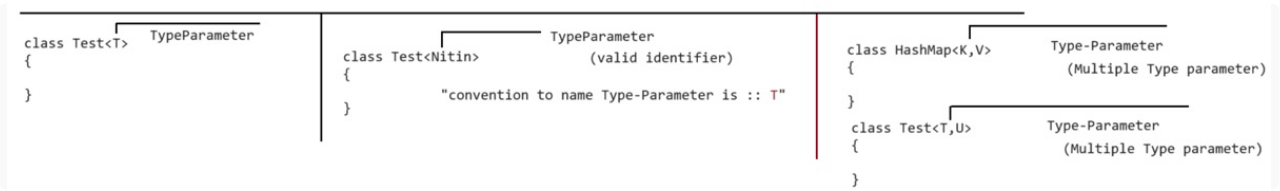
class Test<T extends Runnable>{}**//valid**

class Test<T super String>{}**//invalid**

**To specify the bounded types we need to use only extends keyword,we can't use implements and super keyword,but we can replace implements keyword with extends keyword.**

As the type parameter we can use any valid java identifier but it convention to use T always.

**T=> Type parameter.**




**Multiple parameters example →** HashMap(key,value)

**eg#1.**

```
class Sample<T extends Number>
{
}
class Test
{
    public static void main(String[] args)
    {
        Sample<Number> s1 =new Sample<Number>();
```

```
        Sample<Integer> s2 =new Sample<Integer>();
        Sample<String> s3 =new Sample<String>();//CE: String is not bounded within T type
    }
}
```

**eg#2.**

```
class Sample<T extends Runnable>
{
}
class Test
{
    public static void main(String[] args)
    {
        Sample<Runnable> s1 =new Sample<Runnable>();
        Sample<Thread> s2 =new Sample<Thread>();
        Sample<String> s3 =new Sample<String>();  //String is not within bounds of type-variable T
    }
}
```

**Generic methods and wild-card character (?)**

```
ArrayList<String> l=new ArrayList<String>();
m1(l);
;;;;;
;;;;;
ArrayList<Integer> l1=new ArrayList<Integer>();
m1(l1);
;;;;;;
;;;;;;
ArrayList<Double> l2=new ArrayList<Double>();
m1(l2);
;;;;;;
;;;;;;
ArrayList<Student> l3=new ArrayList<Student>();
m1(l3);
```

```
public static void m1(ArrayList al){
    ;;;;
    ;;;;
}
public static void m1(ArrayList al){
    ;;;;
    ;;;;
}
public static void m1(ArrayList al){
    ;;;;
    ;;;;
}
public static void m1(ArrayList al){
    ;;;;
    ;;;;
}
```

**As noticed the length of the code is increased and it increases the burden. To reduce the burden we can write the code as shown below**

```
public static void m1(ArrayList<?>  al){
    ;;;;;
    ;;;;;
}
```

**methodOne(ArrayList<String> l):**

This method is applicable for ArrayList of only String type.

**Example:**

l.add("A");

l.add(null);

l.add(10)**;//(invalid)**

Within the method we can add only String type of objects and null to the List.

**methodOne(ArrayList<?> l):**

**We can use this method for ArrayList of any type but within the method we can't add anything to the List except null.**

Example:

l.add(null)**;//(valid)**

l.add("A");**//(invalid)**

l.add(10);**//(invalid)**

**This method is useful whenever we are performing only read operation.**

m1(ArrayList<?> l){ System.out.println(l);}

**methodOne(ArrayList<? extends x> l):**

**=>** If x is a class then this method is applicable for ArrayList of either x type or its child classes.

**=>** If x is an interface then this method is applicable for ArrayList of either x type or its implementation classes.

**=> In this case also within the method we can't add anything to the List except null.**

**=> This method is useful whenever we are performing only read operation.**

**Case 4: At method level super keyword is valid while at class level its not valid.**

**methodOne(ArrayList<? super X> al)**

=> We can call this method by passing ArrayList<? super X> where

X -> X can be class or interface.

X -> If X refers to class, then it can be X type **or its Parent type**

X -> If X refers to interface, then it can be interface type or **its super classes of implementation of X.**

**eg#1.**

Runnable

|

|

Thread extends Object

methodOne(ArrayList<? super Runnable> al)

**|=> Object**

ArrayList<String> al = new ArrayList<String>();**//valid**

ArrayList<?> al = new ArrayList<String>()**;//valid**

ArrayList<?> al = new ArrayList<Integer>()**;//valid**

ArrayList<? extends Number>al = new ArrayList<Integer>();**//valid**

ArrayList<? extends Number>al = new ArrayList<String>();**//invalid**

ArrayList<?> al = new ArrayList<? extends Number>(); **//invalid→ on RHS always specific type should be there so that compiler should not get confused**

ArrayList<?> al = new ArrayList<?>()**;//invalid**

**Declaring TypeParameter at the classLevel**

class Test<T>

{

}

**Declaring TypeParameter at the MethodLevel**

Just declare before the returntype

1. public <T> void methodOne(T t){}**//valid**
2. public <T extends Number> void methodOne(T t){}**//valid**
3. public <T extends Number&Comparable> void methodOne(T t){}**//valid**
4. public <T extends Number&Comparable&Runnable> void methodOne(T t){}**//valid**
5. public <T extends Number&Thread> void methodOne(T t){**}//invalid**
6. public <T extends Runnable&Comparable> void methodOne(T t){}**//valid**

**Communication of GenericCode with Non-Generic Code(when in differnt scope)**

To provide **compatibility with old version** sun people compramized the concept of generics in very few area's the following is one such area.

**Example:**

```
import java.util.ArrayList;
class Test
{
    //JDK18
    public static void main(String[] args)
    {
        //JDK1.5V
        //Generic version of ArrayList
        ArrayList<String> al =new ArrayList<String>();
        al.add("sachin");
        methodOne(al);
        System.out.println(al);/[sachin, 10, 54.5, true]
    }
    //JDK1.4V
    public static void methodOne(ArrayList al)
    {
        //ArrayList al :: NonGeneric Version
        al.add(10);
        al.add(54.5f); //(when in differnt scope)
```

```
        al.add(true);
    }
}
```

**Output**

**[sachin, 10, 54.5, true]**

**Conclusions :**

**Generics concept is applicable only at compile time(because typesafety is done at compile time and typecasting at runtime(by jvm)[LHS type parameter is important not RHS to decide the generic of the expression]), at runtime there is no such type of concept. At the time of compilation, as the last step generics concept is removed,hence for jvm generics syntax won't be available.**

Hence the following declarations are equal.

ArrayList l=new ArrayList<String>();

ArrayList l=new ArrayList<Integer>();

ArrayList l =new Arraylist<Double>();

All are equal at runtime,becoz compiler will remove these generics syntax

ArrayList l=new ArrayList();

**Example 1:**

```
import java.util.*;
class Test {
    public static void main(String[] args) {
        ArrayList l=new ArrayList<String>();
        l.add(10);
        l.add(10.5);
        l.add(true);
        System.out.println(l);// [10, 10.5, true]
    }
}
```

**Example 2:**

```
import java.util.*;
class Test {
    public void methodOne(ArrayList<String> l){}
    public void methodOne(ArrayList<Integer> l){}
}
```

**CE: duplicate methods found error: name clash: methodOne(ArrayList) and methodOne(ArrayList) have the same erasure(LHS matters not RHS)**

**Example3:**

The following 3 declarations are equal.

```
ArrayList al4 = new ArrayList();
ArrayList al5 = new ArrayList();
ArrayList al6 = new ArrayList<>();
```

For these ArrayList objects we can add only String type of objects.

l1.add("A");**//valid**

l1.add(10); **//invalid**

**Example4: all are valid**

```
List<Map<Integer,String>> al1 = new ArrayList<Map<Integer,String>>();
List<Map<Integer,String>> al2 = new ArrayList<>();
List<Map<Integer,String>> al3 = new ArrayList();
```

**Example5: in higher versions(jdk9 and above) diamond(<>) operation works for inner ananomous classes as well.**

```
ArrayList jdk7 =new ArrayList();
ArrayList jdk8=new ArrayList<>();//Extension of diamond operator in jdk8 version
//Extension of diamod operator in jdk9 and above version
ArrayList jdk9 =new ArrayList<>()
{
};
```

**Example6:**

```
import java.util.*;
class MyGenClass<T>
{
    T obj;
    public MyGenClass(T obj){
        this.obj =obj;
    }
    public T getObj(){
        return obj;
    }
    public void process(){
        System.out.println("Processing obj....");
    }
```

```
}
public class Test
{
    public static void main(String[] args)
    {
        MyGenClass<String> c1 =new MyGenClass<>("sachin"){
            @Override
            public void process(){
                System.out.println("Processing :: "+getObj());
            }
        };
        c1.process();
        MyGenClass<String> c2 =new MyGenClass<>("dhoni"){
            @Override
            public void process(){
                System.out.println("Processing :: "+getObj());
            }
        };
        c2.process();
    }
}
```

**Output**

**JDKVersion :: 1.8**

```
D:\OctBatchMicroservices\Test.java:20: error: cannot infer type arguments for
MyGenClass
MyGenClass c1 =new MyGenClass<>("sachin"){
^
reason: cannot use '<>' with anonymous inner classes
where T is a type-variable:
T extends Object declared in class MyGenClass
D:\OctBatchMicroservices\Test.java:29: error: cannot infer type arguments for
MyGenClass
MyGenClass c2 =new MyGenClass<>("dhoni"){
^
reason: cannot use '<>' with anonymous inner classes
where T is a type-variable:
```

T extends Object declared in class MyGenClass

**Output**

**JDKVersion :: 9 and above**

D:\OctBatchMicroservices>java -version

java version "18.0.1.1" 2022-04-22

Java(TM) SE Runtime Environment (build 18.0.1.1+2-6)

Java HotSpot(TM) 64-Bit Server VM (build 18.0.1.1+2-6, mixed mode, sharing)

D:\OctBatchMicroservices>javac Test.java

D:\OctBatchMicroservices>java Test

Processing :: sachin

Processing :: dhoni

**final**

- **a. method =>** it can't be overriden in child class.
- **b. class =>** class won't participate in inheritance.
- **c. variable =>** It would be treated like CompileTimeConstant,whose value should not be changed in the application.(compiler replace the variable with value)

**In java application to declare constant variables, java has provided a standard convention "public static final"**

**eg#1.**

**System class**

public static final InputStream in;

public static final PrintStream out;

public static final PrintStream err;

**eg#2.**

**Thread class**

public static final int MIN_PRIORITY;

public static final int NORM_PRIORITY;

public static final int MAX_PRIORITY;

**eg#3.**

```
class UserStatus
{
    public static final String AVAILABLE="Available";
    public static final String BUSY="Busy";
    public static final String IDLE="Idle";
```

```
}
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(UserStatus.AVAILABLE);
        System.out.println(UserStatus.BUSY);
        System.out.println(UserStatus.IDLE);
    }
}
```

**Output**

**Available**

**Busy**

**Idle**

To declare the constant variables in java application, if we use above convention then we are able to get the following problems.

a. we must declare "public static final" for each constant variable explicitly.

b. It is possible to allow multiple datatype to represent one type,it will reduces typednesss in java application.

c. If we access constant variables then these variables will display their values, here constant variable values may or may not reflect the actual meaning of constant variables.

**To resolve the above mention limitations, we need to go for "enum".**

**enum(1.5V)**

In case of enum,

1. All the constants are by default "public static final".
2. All the constant variables are by default **the same type called "enum",**it will improve the typedness of java applications.
3. All the constant variables **are by default "Named Constants"** that is these constant variables will display their names instead of their values.

**General Syntax**

```
[Access_modifier] enum EnumName{
    ---List of Constants----
}
```

**eg#1.**

**//Group of "NamedConstants"**

```
enum User_Status{
    AVAILABLE,BUSY,IDLE; //semicolon is optional
```

```
}
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(User_Status.AVAILABLE);
        System.out.println(User_Status.BUSY);
        System.out.println(User_Status.IDLE);
    }
}
```

**Output**

**AVAILABLE**

**BUSY**

**IDLE**

**Upon Compilation of enum code**

```
final class User_Status extends java.lang.Enum
{
    public static final User_Status AVAILABLE = new User_status();
    public static final User_Status BUSY = new User_status();
    public static final User_Status IDLE = new User_status();
}
```

**eg#2.**

```
enum Result_Status{
    PASS,FAIL,ABSENT;
}
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(Result_Status.PASS);
        System.out.println(Result_Status.FAIL);
        System.out.println(Result_Status.ABSENT);
    }
}
```

**Upon Compilation of enum code**

```
final class Result_Status extends java.lang.Enum
{
    public static final Result_Status PASS = new Result_status();
    public static final Result_Status FAIL = new Result_status();
    public static final Result_Status ABSENT = new Result_status();
}
```

**Note:**

1. **Internally enum concept is implemented using "class" concept.**
2. **Every enum constant is a reference variable to that enum type object.**
3. Every enum constant is by default public static final.
4. **Internaly inside every enum toString() is implemented to return the name of the constant.**
5. **By using enum we can define our own datatypes which comes under "UserDefinedDatatype"**
   **eg: class,enum**

**enum in switch**

**// JDK1.4V -> allowed argument types are :: byte,short,char,int**

**// JDK1.5V -> byte,short,char,int + Wrapperclasses + enum types**

**// JDK1.7V -> byte,short,char,int + Wrapperclasses + enum types + String type**

```
enum Result_Type{
    PASS,FAIL,ABSENT;
}
class ResultType{
    public static final String PASS = "pass";
    public static final String FAIL = "fail";
    public static final String ABSENT = "absent";
}
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Using String");
        String result = ResultType.PASS;
        switch (result)
        {
            case "pass":System.out.println("FCD");
            break;
```

```
            case "fail":System.out.println("Ooops....");
            break;
            case "absent":System.out.println("Absent...");
            break;
            case "Rajinikanth": System.out.println("Invalid input");
            break;
            default : System.out.println("invalid option...");
        }
        System.out.println();
        System.out.println("Using ENUM");
        Result_Type resultType=Result_Type.PASS;
        switch (resultType)
        {
            case PASS:System.out.println("FCD");
            break;
            case FAIL:System.out.println("Ooops....");
            break;
            case ABSENT:System.out.println("Absent...");
            break;
            default : System.out.println("invalid option...");
        }
    }
}
```

**Output**

**Using String**

**FCD**

**Using ENUM**

**FCD**

**Power of Enum → For the first switch we can add invalid cases as well(which is not a constant and part of ResultType) and if we do the same thing with second case then it will immediately throw a CE.Why this is important?? Consider if some exception happens with "Rajinikanth" case and it was not tested than at prod level it will be a big problem.**

**Note: If we are passing enum type as the argument to switch, then those case labels should be always valid enum constants only, otherwise it would result in "CompilertimeError".**

**What are the places where enum can be defined in java program?**

**a. Inside the class. //valid**

```
class X{
    enum Y{
```

```
    }
}
```

**b. Inside the method. //invalid**

```
class X{
    public void m1(){
        enum Y{
        }
    }
}
```

**Enum vs inheritance**

1. Every enum is by default final, so enum won't participate in inheritance.
2. **Every enum by default will inherit from "java.lang.Enum", so explicitly enum can't inherit from another class because java won't support mulitple inheritance through classes.**
3. **Enum can implement any no of interfaces silmultaneously.**

a. enum X{}

enum Y extends X{} **//invalid**

enum X extends Enum**{} //invalid**

b. class X{}

enum Y extends X{} **//invalid(2nd point)**

c. enum X{}

class Y extends X{} **//invalid**

d. interface X{}

enum Y implements X{} **//valid**

**java.lang.Enum**

1. For every enum the java.lang.Enum would act like a base class.
2. It implements serializable and cloneable interface.
3. It is a direct child class of "Object" class.
    a. values() **=> it lists all constants of enum**
    b. public final int ordinal() **=> it prints the order of constants.**

**Speciality of enum**

1. Inside enum we can write
    a. method : **static,instance :: only concrete, no abstract methods**
    b. constructor : yes
    c. normal variables : yes

d. constants : yes

**eg#1.**

```
enum ResultType{
    PASS,FAIL,ABSENT;
    public void methodOne(){
    }
    public static void main(String[] args){
        System.out.println("inside enum main()...");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        ResultType[] result = ResultType.values();
        for(ResultType resultType : result)
        System.out.println(resultType+"....."+resultType.ordinal());
    }
}
```

**Now we have two main methods so we need to execute both of them(java Test, java ResultType), then we can get output for both separately.**

**Output**

**D:\OctBatchMicroservices>javac Test.java**

**D:\OctBatchMicroservices>java Test**

**PASS.....0**

**FAIL.....1**

**ABSENT.....2**

**D:\OctBatchMicroservices>java ResultType**

**inside enum main()...**

**Note:**

**In addition to constants, we can write extra members like methods then the list of constants should be in 1st line and it should end with semicolon.**

If we are taking any extra member then enum should contain atleast one constant.

Empty enum is also valid.

**case1:**

**//Invalid**

```
enum ResultType{
    public void methodOne(){}
    PASS,FAIL,ABSENT;
}
```

**Case2:**

**//Invalid**

```
enum ResultType{
    public void methodOne(){}
}
```

**Case3:**

**//valid**

```
enum ResultType{
}
```

**Case4:**

**//valid**

```
enum ResultType{
    PASS,FAIL,ABSENT; //; is compulsory
    public void methodOne(){}
}
```

**Case5:**

**//valid**

```
enum ResultType{

    public void methodOne(){}
}
```

**Enum vs Constructor**

1.Enum can contain a constructor.

2. Every enum constant represent an object of that enum class which is static.
3. **Enum constants will be created at the time of loading the .class file which internally triggers constructor to create an object.(hence console of constructor is printed first)**

eg#1.

```
enum ResultType{
    PASS,FAIL,ABSENT;
```

```
    //public static final ResultType PASS = new ResultType();
    //public static final ResultType FAIL = new ResultType();
    //public static final ResultType ABSENT = new ResultType();
    static{
        System.out.println("ResultType.class file is loading...\n");
    }
    ResultType(){
        System.out.println("Constructor is called...");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        ResultType result = ResultType.PASS;
    }
}
```

**Output**

```
D:\OctBatchMicroservices>javac Test.java
D:\OctBatchMicroservices>java Test
Consturctor is called...
Consturctor is called...
Consturctor is called...
ResultType.class file is loading...
```

**eg#2.**

```
enum ResultType{
    PASS,FAIL,ABSENT;
    //public static final ResultType PASS = new ResultType();
    //public static final ResultType FAIL = new ResultType();
    //public static final ResultType ABSENT = new ResultType();
    static{
        System.out.println("ResultType.class file is loading...\n");
    }
    ResultType(){
        System.out.println("Constructor is called...");
    }
```

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

**Output**

**D:\OctBatchMicroservices>javac Test.java**

**Test.java:22: error: enum classes may not be instantiated**

```
enum ResultType{
    PASS(65),FAIL(28),ABSENT;
    int marks;
    //public static final ResultType PASS = new ResultType(65);
    //public static final ResultType FAIL = new ResultType(28);
    //public static final ResultType ABSENT = new ResultType();
    static{
        System.out.println("ResultType.class file is loading...\n");
    }

    //for ABSENT zero argument constructor will be called automatically by jvm
    ResultType(){
        marks = 0;
        System.out.println("Zero argument constructor is called...");
    }
    ResultType(int marks){
        this.marks = marks;
        System.out.println("Parameterized Constructor is called...");
    }
    public int getMarks(){
        return marks;
    }
}
public class Test
```

```
{
    public static void main(String[] args)
    {
        ResultType[] result = ResultType.values();
        for (ResultType resultType : result )
        {
            System.out.println(resultType + "..... " +resultType.getMarks());
        }
    }
}
```

**Output**

**Parameterized Constructor is called...**

**Parameterized Constructor is called...**

**Zero argument constructor is called...**

**ResultType.class file is loading...**

**PASS..... 65**

**FAIL..... 28**

**ABSENT..... 0**

**What is the difference b/w enum,Enum,Enumeration?**

**enum ->** it is a keyword which is used to define group of "NamedConstants".

**Enum ->** it is a inbuilt class which acts like a base class for every userdefined enum.

**Enumeration ->** it is an interface which is a part of java.util package, using this interface we can iterate an object from collections.

**enum will contain everything that a class contains except abstract classes, as we can't inherit it and object creation is also not possible, hence abstract classes in enum doesn't make any sense. Hence it will have only concreate classes.**