Development

Javac :-

We Can use this Command to Compile a Single or group of .java files.

Syn:- javac [options] A.java / A.java B.java / *.java

[options] →
-d
-source
-cp
-classpath
-version

java :-

We Can use java Command to run a .class file

Syn:- java [options] A ↵

[options] →
-ea | -esa | -da | -dsa
-version
-cp / -classpath
-D

Note:- We can compile a group of .java files at a time where as we can run only on .class file at a time.

Classpath :-

→ Classpath describes the location where required .class files are available.

→ JVM will always use Classpath to locate the required .class file.

→ The following are various possible ways to Set the Classpath.

① permenently by using environment variable classpath.

→ This Classpath will be preserved after system restart also

② At Command prompt level by using Set Command.

Set classpath = %classpath% ; D:\path >

→ This classpath will be applicable only for that particular Comand prompt window only. Once we close that Command prompt automatically classpath will be lost

③ At Command Level by using -cp option

java -cp D:\path > Test ↵

→ This class path is applicable only for this particular Command. Once Command execution Completes automatically Classpath will be lost.

* Among the above 3 ways the most Commonly used approach is Setting classpath at Command Level.

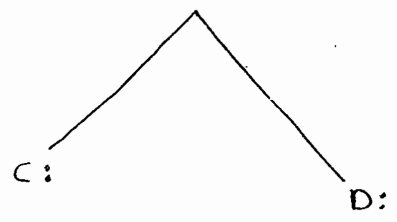
Ex:-
```
class Test
{
  p.s.v.m (——)
  {
    S.o.pln(" Classpath Demo");
  }
}
```

```
D:\ Durgaclasses\> javac Test.java ↵
                 >java Test ↵
      o/p:- Classpath Demo
✗D:\ java Test ↵     R.E:- NoClassDefFound Error
✓D:\> java -cp D:\Durgaclasses Test ↵ ✓
                  o/p:- classpathDemo
✓C:\> java -cp D:\Durgaclasses Test ↵
                o/p:- ClasspathDemo
```

Note:-

If we set classpath explicitly then we can run Java program from any Location but if we are not setting the classpath then we have to run java program only from current working directory.

Ex2:-



```
public class fresher              class Company
{                                 {
  public void m1()                  p.s.v.m (____)
  {                                 {
    S.o.pln ("I want JOB");           Fresher f =new Fresher();
  }                                   f.m1()
}                                     S.o.pln("Getting JOB is very
                                              easy..not required to
                                              crazy");
                                    }
                                  }
```

```
C:\> javac Fresher.java ✓
D:\> javac Company.java ✗
              C.E:- Cannot find symbol
                    Symbol:- class Fresher
                    location : class Company
D:\> javac -cp C: Company.java ✓
✗ D:\java Company ←
     R.E:- NoClassDefFoundError: Fresher
✗ D:\ java -cp C: Company ←
     R.E:- NoClassDefFoundError: Company
```

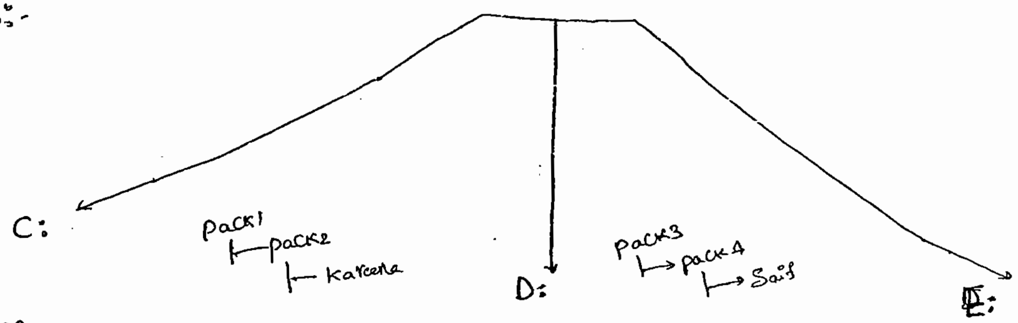
✓ D:\ java -cp D:;C: Company (or) D:\java -cp .;C: Company

o/p :- I won JOB

Getting JOB is very easy... not required to worry.

✓ E:\ java -cp D:;C: Company

Ex3:-



```
Package pack1.pack2;
Public class Kareena
{
 public void m1()
 {
S.o.pln(" Hello Saif Can U
        please set hello
          tune");
 }
}
```

```
package pack3.pack4;
import pack1.pack2.Kareena
public class Saif
{
 public void m1()
 {
  Kareena k = new Kareena();
  K.m1();
  S.o.pln("Not possible...As I am
            in S(J)P class");
 }
}
```

```
import pack3.pack4
        .Saif;
class Durga
{
 p.s.v.m(—)
 {
  Saif s = new Saif();
  S.m1();
  S.o.pln(" Can
    I help U");
 }
}
```

✓ C:\> java -d. Kareena

∅ D:\> java -d. Saif.java

C.E:- Cannot find Symbol
Symbol : Class Kareena
Location : Class Saif

✓ D:\ java -cp c: -d . Saif.java

✗ E:\ javac Durga.java

C.E:- Cannot find Symbol
Symbol: class Saif
location: class Durga

✓ E:\> javac -cp D: Durga.java

✗ E:\> java Durga ↵

R.E:- NoClassDefFoundError : Saif

✗ E:\> java -cp D: Durga ↵

R.E:- NoClassDefFoundError : Durga

✗ E:\> java -cp .;D: Durga

R.E:- NoClassDefFoundError : Kareena

✓ E:\> java -cp E:;D:;C: Durga

Note:-

① Compiler will check only one level dependency where as Jvm will check all levels of dependency

② If any folder structure created because of package statement it should be resolved through import statement only. & Base package location we have to update in classpath.

③ within the classpath the order of locations is very important for the required .class file, JVM will always search the locations from

Left → Right in classpath. Once JVM finds the required file then the rest of the classpath wont to be searched.

C:

```
Class Nagavali
{
 p.s.v.m(—)
 {
 S.o.pln("C: Nagavali");
 }
}
```

D:

```
Class Nagavali
{
 p.s.v.m(—)
 {
 S.o.pln("D: Nagavali");
 }
}
```

E:

```
Class Nagavali
{
 p.s.v.m()
 {
 S.o.pln("E:Nagavali
 }
}
```

```
C:\> javac Nagavali.java ✓
D:\> javac Nagavali.java ✓
E:\> javac Nagavali.java ✓
C:\> java Nagavali ✓
        o/p C: Nagavali
D:\> java -cp C:;D:;E: Nagavali ↵
        o/p C: Nagavali
D:\> java -cp E:;D:;C: Nagavali ↵
        o/p!- E:Nagavali
D:\ java -cp D:;E:;C: Nagavali ↵
        o/p' D: Nagavali
```

JAR file :-

→ If several dependent files are available then it is never recommended to set each class file individually in the classpath we have to group all those .class file into a single zip file & we have to make that zip file available in the class path. this zip file is nothing but JAR file.

Ex(1):-

To develope Servlet all required .class files are available in Servlet-api.jar. we have to make this Jar file available in the classpath then only Servlet will be Compiled.

jar vs war vs ear :-

⇒ ① jar: (java archieve file)

→ It Contains a group of .class files

② war :- (web archieve file)

→ It represents a web application which may contain Servlets, Jsps, HTMLs, css file, Javascripts, e.t.c..

③ ear :- (Enterprise archieve file)

→ It represents an enterprise application which may Contains Servlets, JSPs, EJBs, JMS Components e.t.c .

Various Commands :-

① ~~To~~ Create a jar file :

```
jar -cvf durga.jar A.class B.class C.class
                                *.class
```

② To extract a jar file :-

```
jar -xvf durga.jar
```

③ To Display table of Contents of a jar file :-

```
jar -tvf durga.jar
```

Ex:-

```
public class DurgaColorfullCalc
{
  public static int add(int x, int y)
  {
    return x*y;
  }
 public static int add(int x, int y)
  {
    return 2*x*y;
  }
}
```

```
C:\> javac DurgaColorfullCalc.java ✓
C:\> jar -cvf durgaCalc.jar DurgaColorfullCalc.class
```

```
class   Bakara
{
 p.s.v.m(——)
 {
 s.o.pln(DurgaColorfulCalc.add(10,20));
 s.o.pln(DurgaColorfulCalc.multply(10,20));
 }
}
```

```
✗ D:\> javac  Bakara.java
✗ D:\> javac  -cp  c:  Bakara.java
✓ D:\> javac  -cp  c:\durgacalc.jar  Bakara.java
✓ D:\> javac  -cp  .;c:\durgaCalc.jar  Bakara.
```

O/P:- 200
400

Note:-

→ when ever we are placing a jar file in the classpath compulsary name of the jar file we should include, Just Location is not enough.

Short cut way to place Jar file :-

→ If we are placing the jar file in the following Location then it is not required to set classpath explicitly bydefault it is available to Jvm & Java Compiler.

```
JDK
 |→JRE
    |→Lib
       |→ ext
          |→ *.jar
```

System properties :-

→ for every System persistance information will be maintain in the form of System properties. These may include o.s name, Varchual mechine version, User Country .e.t.c....

→ we can get System properties by using getProperties() method of System class

Ex:- Demo program to print all System properties.

```
import java.util.*;
class Test
{
 public static void main(String[] args)
  {
    Properties p = System.getProperties();
    p.list(System.out);
  }
}
```

→ we can set System property from the command prompt by using -D option

Space is not allowed

ex:- `java -Ddurga=SCJP Test`

name of the property

Value of the property

Q) JDK Vs JRE Vs JVM :-

JDK:(Java developement kit) :-

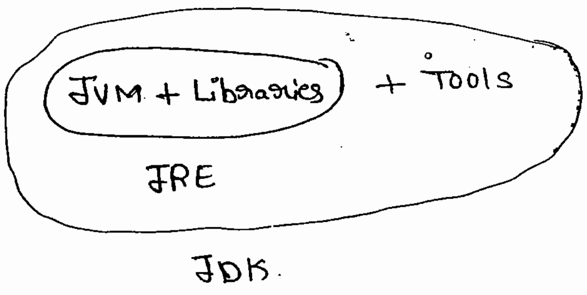
→ To develop & run Java application the required environment provided by JDK.

JRE :- (Java Runtime Environment) :-

→ To run Java application the required environment provided by JRE

JVM :-

→ this machine is responsible to execute Java program.



JDK = JRE + Tools

JRE = JVM + Libraries.

Note:-

→ On Client machine we have to install JRE, where as on the developers machine we have to install ~~JR~~DK.

diff. b/w path & classpath :-

→ we can use classpath to describe the location where required .class files are available.

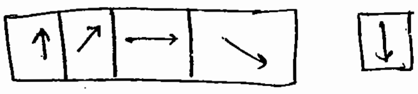
→ If we are not setting the classpath then our program won't be run.

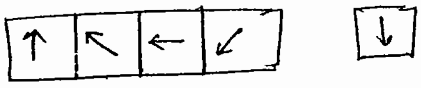
Path :-

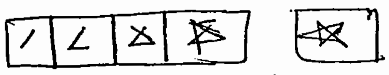
→ we can use path variable to describe the Location where required Binary executables are available.

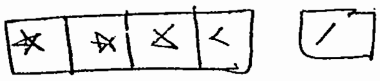
→ If we are not setting path variable then java & javac Commands won't work.

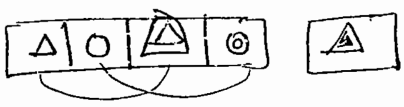
237

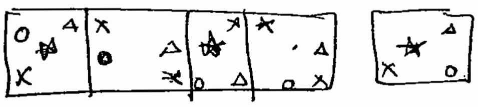
1) Clockwise 

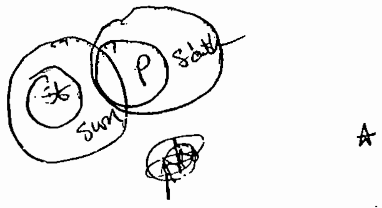
2) Anticlock 

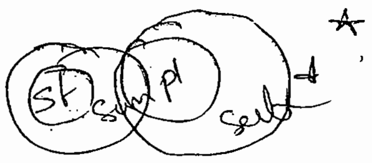
3) Increasing 

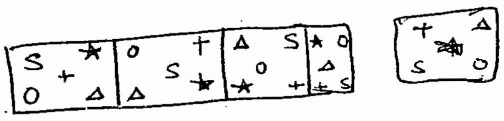
4) Decreasing 

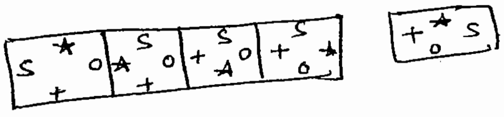
5) Alternate 

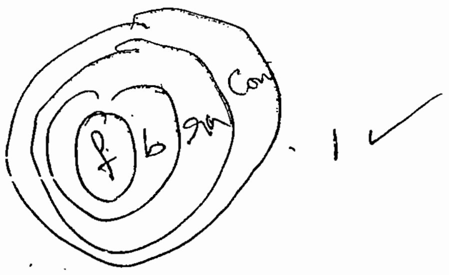
6) multiple movement 

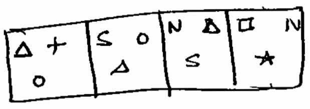


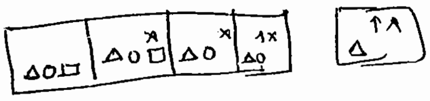


7) Rotation 

8) Interchange 



9) missing of fig 

10) Substitution 

miracle

1.5V
- → Autoboxing & Unboxing -
- → generics;
- → var-arg -
- → for-each
- → enum
- → Annotations ✓
- → Queue ✓
- → Static imports } not recommended ✓
- → Co-Varic of return types.

Walk
↓
Jogging
↓
Running
↓
Sprinting

Siddhartha (VNR VJIT)
9951884313
Siddharthahprs@yahoo.co.in

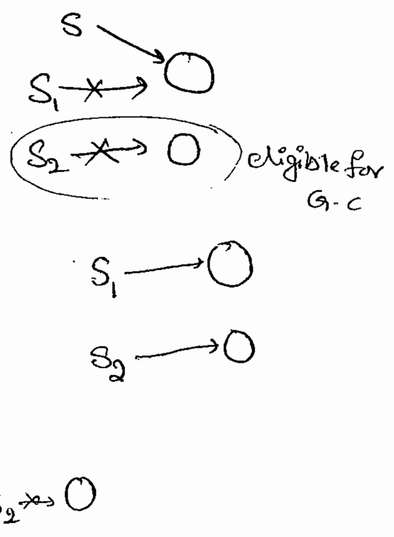
Vishnuteja.Y.S (
9703340473, 9495410648
Vishnuteja87@gmail.com

Vasu - 8179969444 (CEVM)
slvudsarma@gmail.com.

Ex 2:-

```
Class Test
{
 p.s.v.m(String[] args)
 {
   Student s = m1();
 }
 p.s.Student m1()
 {
   Student s1 = new Student();
   Student s2 = new Student();
   return s1;
 }
}
```

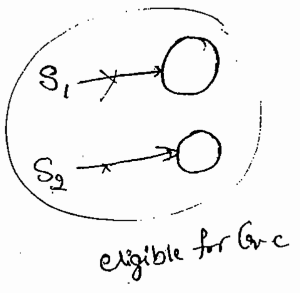
one object eligible for G.C →



Ex 3 :-

```
Class Test
{
 p.s.v.main (String[] args)
 {
    m1();
 }
 p.s.Student m1()
 {
   Student S1 = new Student();
   Student S2 = new Student();
   return S1;
 }
}
```

Two objects eligible for G.C →

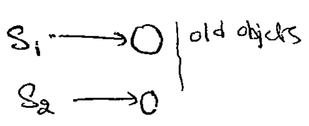


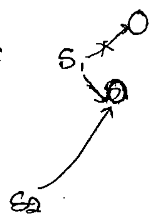
2) Reassigning the Reference variable :-

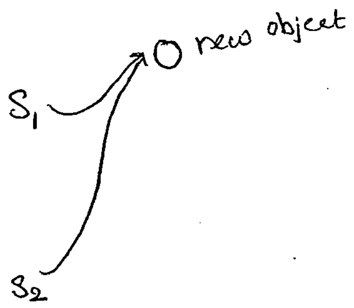
→ If an object is no longer required then reassign its reference variables to some other objects then that old object automatically eligible for G.C.

ex(i):-

```
                        Student S1 = new Student();
no object eligible for G.C  →  Student S2 = new Student();
                              :
One object eligible for G.C  →  S1 = new Student();
                              :
Two objects eligible  →  S2 = S1;
```







(3) Objects Created Inside a method :-

→ The Objects which are Created inside a method are by default eligible for G.C after Completing that method.

ex!-

```
class Test
{
    p.s.v. main(String [] args)
    {
2 objects eligible for G.C  →  m1();
    }
    p.s.v. m1()
    {
        Student S1 = new Student();
        Student S2 = new Student();
```

26/02/11

Garbage Collection

1) Introduction.
2) Various ways to make an object eligible for G.C.
3) The methods for requesting JVM to run garbage collector.
4) Finalization..

Garbage Collector :→

→ In old languages like C++, creation & destruction of object is responsibility of programmer only.

→ Usually programmer taking very much care while creating objects & his neglecting destruction of useless objects. due to this neglectance at certain point of time for the creation of new object sufficient memory may not be available & entire program will be collaps due to memory problems.

→ But in Java, programmer is responsible only for creation of objects and He is not responsible for destruction of useless objects.

→ Sun people provided one assistent which is always running in the background for destruction of useless objects. due to this assistant the chance of faillure java program with memory problem is very rare. this assistant is nothing "Garbage Collector".

→ Hence, the main objective of Garbage Collector is to "destroy useless objects".

The Various ways to make an object elligible for G.c :-

→ Eventhough programmer is not responsible to destroy useless objects, it is always a good programing practice to make an object eligible for G.c if it is no longer required.

→ An object is said to be eligible for G.c, if it doesn't contain any references.

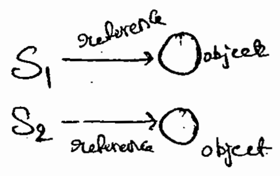
→ The following are various possible ways to make an object eligible for G.c.

(i) Nullyfying the reference variable :-

→ If an object is nolonger required then assign null to all its references, then automatically that object eligible for G.c.

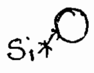
Ex: (1)

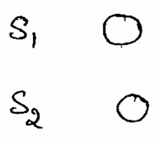
```
Student S1 = new Student();
Student S2 = new Student();
```



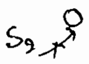
No objects eligible for G.c →

one object eligible for G.c → S1 = null;





two objects eligible for G.c → S2 = null;



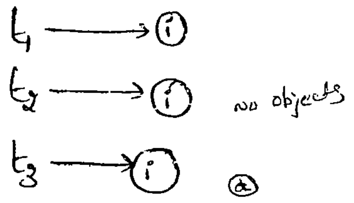
4) Island of isolation :-

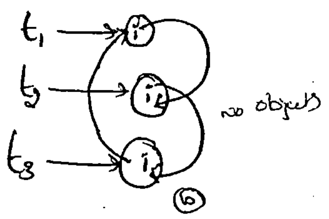
Ex:-

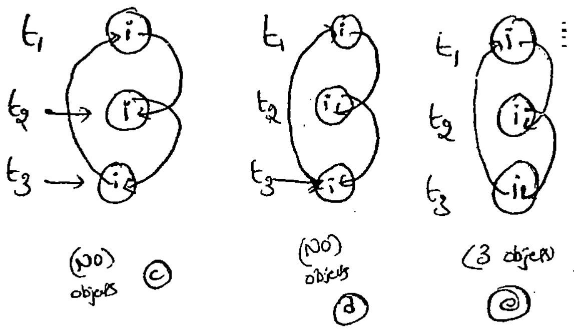
```
Class Test
{
    Test i;
   p.s.v. main(String[] args)
  {
   Test t1 = new Test();
   Test t2 = new Test();
   Test t3 = new Test();
   :
    t1.i = t2;
    t2.i = t3;
    t3.i = t1;
    :
     t1 = null;
    :
     t2 = null;
    :
     t3 = null;
  }
 }
```

No objects eligible for G.C

3 objects eligible for G.C







Note:-

→ If an Object doesn't have any reference then it is always eligible for Garbage Collector.

→ Eventhough object having ~~obj~~ the reference still it is eligible for G.C Sometimes (island of isolation)

The methods for requesting JVM to Run Garbage Collector:-

→ when ever we are makeing an Object eligible for G.C it may not be destroyed by G.C immediately when ever JVM runs garbage Collector then only that object will be destoryed.

→ We Can request JVM to run garbage Collector, programatically wheather JVM accepts over request are not there is no gaurantee.

→ The following are various ways for ~~this~~ requesting JVM to run G.c.

(1) By System class :-

→ System class Contains a Static method G.C, for this

```
System.gc();
```

(2) By Runtime class :-

→ By using runtime object a Java application Can Communicate with JVM

→ Runtime class is a Singleton class hence we Can't Create Runtime object by using Constructor.

→ we Can Create a Runtime object by using factory method getRuntime(),

```
Runtime r = Runtime.getRuntime();
```

→ Once we got Runtime object we Can apply the following methods on that object.

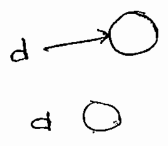
(a) freeMemory() returns freememory in the Heap,

(b) totalMemory() " total " ~~of~~ the Heap (Heap size)

(c) gc() → for requesting JVM to Run garbage Collector,

ex:-

```
class RuntimeDemo
{
  p.s.v.main(String[] args)
  {
    Runtime r = Runtime.getRuntime();
    S.o.pln(r.totalMemory());
    S.o.pln(r.freeMemory());
    for(int i=1; i<=10000; i++)
    {
      Date d = new Date();
      d=null;
    }
    S.o.pln(r.freeMemory());
      r.gc();
      System.out.println(r.freeMemory());
  }
}
```



Q) which of the following is the properway of requested Jvm to run g.c?

✓ 1) System.gc(); (System is static method

✗ 2) Runtime.gc(); (Runtime is instance method)

(gc is applicable only static method)

✗ 3) (new Runtime()).gc();

✓ 4) Runtime.getRuntime().gc();

Note:- gc() present in the System class is a static method, where as gc() present in the Runtime class is instance method & recommended to use System.gc();

Finalization :-

→ Just before destroying any object, garbage collector always calls finalize() method to perform clean-up activities on that object.

→ finalize() method declare in object class with the following declaration.

```
protected void finalize() throws Throwable.
```

Case(1):-

→ Garbage collector always calls finalize() on the object which is eligible for G.C Just before destruction, then the corresponding class finalize() will be executed. If String object eligible for G.C then String class finalize() will be executed. but not Test class finalize method.

Ex1.

```
class Test
{
 p.s.v.m(String[] args)
 {
  String s = new String("durga");
  s = null;
  System.gc();
  System.out.println("end of main");
 }
 public void finalize()
 {
  S.o.pln("finalize method called");
 }}
```

O/P :- end of main

→ In the above Example String object is eligible for g.c. Hence String class finalize() method got executed which has Empty implementation.

→ If we are replacing String object with Test object, then Test class finalize() will be executed.

→ In this case the o/p is ① finalize method called
End of main (or)

② End of main
finalize method Called.

Case 2 :-

→ we can call finalize() Explicitly in this case it will be executed Just like a normal method call & Object won't be destroyed.

→ But Before destruction of an object G.c always call finalize().

Ex:
```
Class Test
{
 p.s.v.m (String [] args)
 {
  Test t = new Test();
    t.finalize();
    t.finalize();
    t=null;
    System.gc();
    S.o.pln("End of main");
 }
 Public void finalize()
 {
  S.o.pln("finalize method Called");
}}
```

o/p:
```
finalize method Called
finalize method Called
end of main
finalize method Called
```

→ In the above program finalize() got Executed 3 times, 2 times Explicitly by the programmer & one time by the Garbage Collector.

Note:-

→ Before destruction of Servelet object Web Container always Calls destroy method, to perform clean-up activities. But

→ It is possible to Called destroy() Explicitly from init() & Service() in this case it will be Executed Just like a normal method Call and Servelet Object won't be destroyed.

Case(3):-

→ If we are Calling finalize() Explicitly & while executing that finalize() if any Exception raised & uncaught, then the program will be terminated abnormally.

→ If G.c calls finalize() & while Executing that finalize(), if any Exception raised is uncaught (no corresponding catch block) Then Jvm Simply ignores that uncaught Exception & rest of the program will be executed normally

Ex:-
```
class Test
{ p.s.v.m (String[] args)
  {
    Test t = new Test();
    t.finalize();    → Line①
    t = null;
    System.gc();
    S.o.pln("end of main");
  }
public void finalize()
{
  S.o.pln(" finalize method called");
  S.o.pln(10/0);
 }
}
```

→ If we are not Comment Line①, then we are Calling the finalize() Explicitly and the program will be terminated abnormally.

→ If we are Commenting Line①, then G.c calls finalize() & the raised A.E is ignored by JVM. Hence in this Case the o/p is

o/p:-
```
end of main!
finalize method Called.
```

Q) which of the following Statement is True?

x 1) while executing finalize() all exceptions are ignored by JVM.

✓ 2) while " " only uncaught exceptions ignored by JVM.
no caught block

Conclusion :-

→ on any object G.c calls finalize() only once.

[Note:-
→ The Behaviour of G.c is vendor dependent & hence we Can't Expect Explicitly because of this we Can't answer]

Ex:
```
Class finalizeDemo
{
  Static finalizeDemo s;
  p.s.v.m(String[] args) throws Exception
  {
   finalizeDemo f = new finalizeDemo();
    S.o.pln(f.hashCode());
    f = null;
    System.gc();
   Thread.sleep(5000);
   System.out.println(s.hashCode());
   S=null;
   System.gc();
   Thread.sleep(5000);
   S.o.pln("End of main method");
  }
  public void finalize()
  {
   S.o.pln("finalize method Called");
   S = this;
  }
}
```
o/p:-
```
4072869
finalize method Called
4072869
  End of main method.
```

Note!- The behaviour of the G.c is vendor dependent & hence we can't Expect exactly because of this we can't answer the following Questions Exactly.

① when Jvm runs G.c exactly.

② what is The Algorithm following by g.c.

③ In which order G.c destroys the Objects.

④ Wheather G.c destroys all eligible objects or not. e.t.c

Note:- We can't tell Exact algorithm followed by G.c, but most of the Cases it is mark & Sweep Algorithm.

Memory leak:-

→ If an object having the Reference then it is not eligible for G.c, eventhough we are not using that object in our program still it is not destroyed by the G.c.. Such type of object is Called "memory leak". (i.e, memory Leak is a useless object which is n't eligible for G.c)

→ We Can resolve memory Leaks by making useless objects for G.c explicitly & by invokeing G.c programmatically.

JpROBE
IBM Tivoli
HP J meter

these are monitoring tools for memory Leak.

(20) Assertions (1.4 version)

(1) Introduction

* (2) Assert as Key-word & identifier

(3) Types of assert Statements

(4) Various Runtime flags

(5) Appropriate & Inappropriate use of assertions

(6) Assertion Errors.

Assertions :-

→ Very Common way of debugging is useing S.o.p statements. But the problem with S.o.p's is after fixing the problem Compulsory we should delete these S.o.p's otherwise these S.o.p's will be Executed at runtime and effects performance & disturbes logging

→ To resolve this problem Sun people introduced Assertions Concept in 1.4 version. Hence the main Objective of assertions is to perform debugging

→ The main Advantage of assertions over Sop is after fixing the Problem it is not required to delete assert Statements because assertions will be desabled automatically at runtime. based on our requirement we can anable & desable assert Statements & Bydefault assertions are desabled.

→ Assertions Concept is applicable for developement & test environment But not for production Environment.

Assert as a Keyword & identifier :-

→ Assert Keyword Introduced in 1.4 Version, Hence from 1.4 version onwards we Can't use assert as identifier. But Before 1.4 we Can use assert as identifier

Ex:-

```
class Test
{
  p.s.v.m(String[] args)
  {
    int assert =10;
    S.o.pln(assert);
  }
}
```

✗ 1) Javac Test.Java

C.E:- as of release 1.4, 'assert' is a Keyword, and may not be used as an identifier

Use -Source1.3 or lower, to use 'assert' as an identifier.

✓ 2) Javac -Source 1.3 Test.Java

```
Java Test ↵
10
```

Types of Assert Statements :-

→ There are 2 types of Assert Statement

(1) Simple version

(2) Augmented version

(1) Simple Version :-

→ assert(b); b → should be boolean-type

→ If b is true, then our assumption satisfied & rest of the program will be executed normally.

→ If b is false, then our assumption fails the program will be terminated by raising runtime Exception saying assertionError. So, that we can able to fix the problem.

Ex:-

```
Class Test
{
 p.s.v.m(String[] args)
 {
  int x =10;
   ⋮
  assert (x >10);
   ⋮
  S.o.pln (x);
 }
}
```

```
① Javac Test.java ✓
② Java Test ✓
   10
*③ Java -ea Test ↵
   R.E: AssertionError
```

(2) Augmented Version :-

→ we can Augment some description by using augmented Version to the AssertionError.

assert(b) : d ;

should be booleantype

any description, Can be any type. but recommended to use String type.

ex:-

```
class Test
{
 p.s.v.m(String[] args)
 {
   int x=10;
   ......
   assert(x>10) : "Here x value should be >10 but it is not";
   ......
   S.o.pln(x);
 }
}
```

① Javac Test.Java

② Java Test ↵
10

③ Java -ea Test ↵

R.E:- AssertionError: Here x value should be >10 but it is not.

Conclusion(1) :-

assert(e1) : e2 ;

→ e2 will be evaluated iff e1 is false. i.e if e1 is True, then e2 won't be evaluated.

ex!- Class Test

```
Class Test
{
p.s.v.m(String[] args)
{
 int x=10;
  ≡
 assert(x==10):~~+~~++x;          assert(x>10): ++x;
  ≡
 S.o.pln(x);
}
}
```

```
✓ javac Test.java ↵                javac Test.java
✓ java Test ↵                      java Test
  10                               10
✓ java -ea Test ↵                  java -ea Test
  10                                 R.E! AssertionError: 11
```

Conclusion(2)!.

assert(e1) : e_2 ;

→ As e2 we can take a method call also but void -type method calls are not allowed.

Eg!- Class Test

```
Class Test
{
 p.s.v.m(String[] args)
 {
  int x=10;
   ≡
  assert(x>10): m1();
   ≡
  S.o.pln(x);
 }
 public static int m1()
 {
  return 8888;
 } }
```

```
✓ javac Test.java ↵
✓ java Test ↵
  10
  java -ea Test
   R.E! AssertionError: 8888
```

→ If m1() return type is void, then we will get CompiletimeError Saying "void type not allowed here."

4) Various Runtime flags:-

① -ea :- To enable assertions in Every non-System class

② -enableassertions:- It is Exactly Same as -ea

③ -da :- To disable assertions in Every non-System class

④ -disableassertions:- Same as -da

⑤ -esa :- To enable assertions in every System class.

⑥ -enableSystemassertions:- It is exactly Same as -esa.

⑦ -dsa :- To disable assertions in Every System class.

⑧ -disableSystemassertions:- It is Same as -dsa .

Ex①:-

java -ea -esa -da ~~-dsa~~ -esa -ea -dsa

Non System class	System class
✓	✓
X	✓
✓	X

→ We can use these flags in together & all these flags Executed from Left to right.

Ex②:-

① java -ea:pack1.A

② java -ea:pack1.B -ea:pack1.pack2.D

③ java -ea -da:pack1.B

Ex 2!-

```
pack1
  ├── A.class
  ├── B.class
  │
pack2
    ├── C.class
    └── D.class
```

→ To anable assertions in only A class

① java -ea:pack1.A

→ To anable assertions in Both B & D classes

java -ea:pack1.B -ea:pack1.pack2.D

→ To anable assertions in every non-System class Except B

java -ea -da:pack1.B

→ To anable assertions in every class of pack1 & it's Sub Packages

java -ea:pack1...

→ To anable assertions in every where with in pack1 Except pack2.

java -ea:pack1... -da:pack1.pack2...

5) Appropriate & Inappropriate Use of assertions:-

1) It is always Inappropriate to mix programming logic with assert Statement because there is no garntee of execution of assert Statement at runtime.

Ex:-

```
withdraw(int x)                  withdraw(int x)
{                                {
  if(x <100)                       assert(x >=100);
  {                              }
    throw new IAE();
  }
}
                  proper way                 Improper way
```

2) In our program if there is any place where the Control not allowed to reach then it is the best place to use assert Statement.

Ex:-
```
switch(x)
{
  Case1: S.o.pln("JAN");
         break;
  Case2: S.o.pln("feb");
         break;
         :
         :
  Case12: S.o.pln("Dec");
         break;
  default:
     assert(false);  → R.E: A.E Can be displayed.
}
```

3) It is always Inappropriate to use assertions for validating public method arguements.

4) It is always Appropriate to use assertions for validating private method arguements

5) It is always Inappropriate to use assertions for validating Command-Line arguements because these are arguements to public main().

6) Assertion Error :-

→ It is the child class of Error & Hence it is unchecked.

→ It is legal to Catch AssertionError by using try-catch but it is stupid kind of activity

Ex:-
```
class Test
{
p.s.v.m(String[]args)
```

Ex:-

```
class Test
{
 p.s.v.m(String[] args)
 {
  int x=10;
   :
   :
  try
  {
  assert (x>10);
  }
  catch(AssertionError e)
  {
   S.o.pln("I am Stupid ...b'z I am Catching
                              AssertionError");
  }
  S.o.pln(x);
 }
}
```

Note:-

→ It is possible to enable assertions either class wise or packagewise.

14|02|10

Exception Handling

1. Introduction
2. Runtime Stack mechanism.
3. Default Exception Handling.
4. Exception Hierarchy.
5. Customized Exception Handling by Try-Catch.
6. Control flow in Try-Catch.
7. Methods to print Exception information.
8. Try with multiple Catch blocks.
9. finally.
10. difference b/w final, finally & finalize.
11. various possible Combinations of Try-Catch-finally.
12. Control-flow in Try-Catch-finally.
13. Control-flow in Nested Try-Catch-finally.
14. Throw.
15. Throws
16. Exception Handling Keywords Summary.
17. Various possible Compile time Error in Exception handling.
18. Customized Exception.
19. Top-10 Exceptions.

Exception :-

→ when unwanted, unexpected Event that distesbes noamal flow of program is Called "Exception".

Ex:- Sleeping Exception, Tyrepunchased Exception, filenotfound-Exception.

→ It is highly recommended to handle Exceptions. the main objective of Exception handling is "Gracefull termination of the program".

→ Exception handling doesnot mean repairing an Exception, we have to define alternative way to Continue rest of the program normally this is nothing but "Exception Handling".

Ex:- If our programing requirement is to read data from the file locating at Landon & at runtime if that file is not available our program should not be terminated abnormally. we have to provide a local file to Continue rest of the program normally. this is nothing but Exception Handling.

Syn:-

```
Try
{
 read data from London file
}
Catch(fileNotFoundException e)
{
 use local file and Continue rest of the program-
 normally.
}
```

Runtime Stack mechanism :-

→ for Every thread JVM will Create a RuntimeStack.

→ All the method Call performed by the thread will be Store in The Stack.

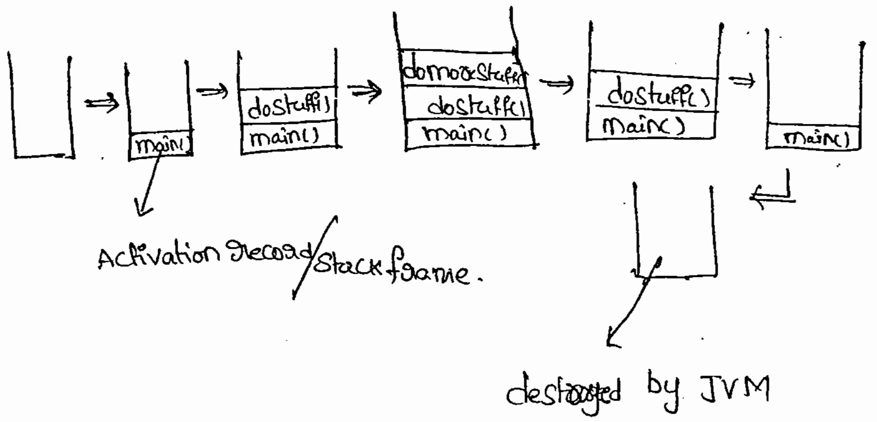
→ Each Entry in the Stack is Called "Activation Record" or "Stack frame"

→ After Completing Every method Call JVM deletes The Corresponding Entry from The Stack.

→ After Completing all method Calls, Just before Terminating The Thread JVM destroyeds the Stack.

Ex:-

```
Class Test
{
p.s.v.m(String args[])
{
 doStuff();
}
p.s.v.doStuff()
{
 doMoreStuff();
}
p.s.v.doMoreStuff()
{
S.o.pln(" don't Sleep");
}
}
```



default Exception handling in Java :-

→ If any Exception raised, the method in which it is raised is responsible to Create Exception object by including the following information.

1. Name of Exception
2. discription of Exception.
3. location of Exception (Stack trace)

→ After Creating Exception object, method handovers that Exception-object to the JVM.

→ JVM checks wheather the method Contains any Exception handling Code or not.

→ If the method Contains any Exception handling Code, then it will be Executed and continue rest of the program normally.

→ If it doesn't Contain handling code, then JVM terminates that method abnormally & removes Corresponding Entry from the Stack.

→ JVM identifies the Caller method & checks wheather Callermethod Contains any handling Code or not. If the Caller method doesn't Contain any handling code, then JVM terminates that Callermethod also abnormally & removes Corresponding Entry from the Stack.

→ this process will Continue until main() & If the main() also doesn't Contain handling Code JVM terminates the main() also abnormally & removes Corresponding Entry from Stack.

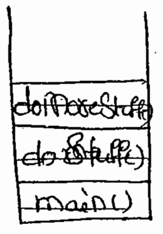
→ Just before terminating the program abnormally JVM handovers the responsibility of Exception handling to the default Exceptionhandler.

→ Default Exception handler just print Exception information to the Console in the following formatt.

Name of Exception : Description
Location (Stack Trace)

15/02/11 :-

```
Class Test
{
P.S.V.m(String []args)
{
 doStuff();
}
P.S.V.doStuff()
{
 doMoreStuff();
}
P.S.V.doMoreStuff()
{
 S.o.pln( 10/0);
}
}
```



Runtime Stack

name of exception
description

```
Exception in thread" main" : java.lang.AE : / by Zero
            at Test.doMoreStuff()
            at Test.doStuff()
            at Test.main ()
```

Stack Trace.

Exception hierarchy:-

→ Throwable class acts as a root for entire Java Exception hierarcy.
It has the following 2 child classes

1. Exception
2. Error

1. Exception:-

→ most of the cases Exceptions are caused by our program & these are Recoverable.

2. Error:-

→ most of the cases Errors are not caused by our program these are due to lack of system resources.

→ Errors are NON-Recoverable.

Checked Vs un-checked Exceptions?

→ the Exceptions which are checked by Compiler for smooth Execution of the program at Runtime are called "checked Exception."

Ex!- HallTicketMissingException,
PenNotWorkingException,
FileNotFoundException.

→ the Exceptions which are not checked by Compiler are called "un-checked Exceptions."

Ex!- BombBlastException.
ArithematicException, FireExcidentException.

→ wheather Exception is checked or unchecked compulsarily it should runtime only. There is no chance of occuring at compile time.

→ RuntimeException and it's child classes
→ Errors & it's child classes are unchecked Exceptions & all remaining are Checked Exceptions

Partially checked vs fully checked :-

→ A checked Exception is said to be fully checked iff all it's child classes also checked.

Ex:- IOException

→ A checked Exception is said to be partially checked iff some of its child classes are unchecked.

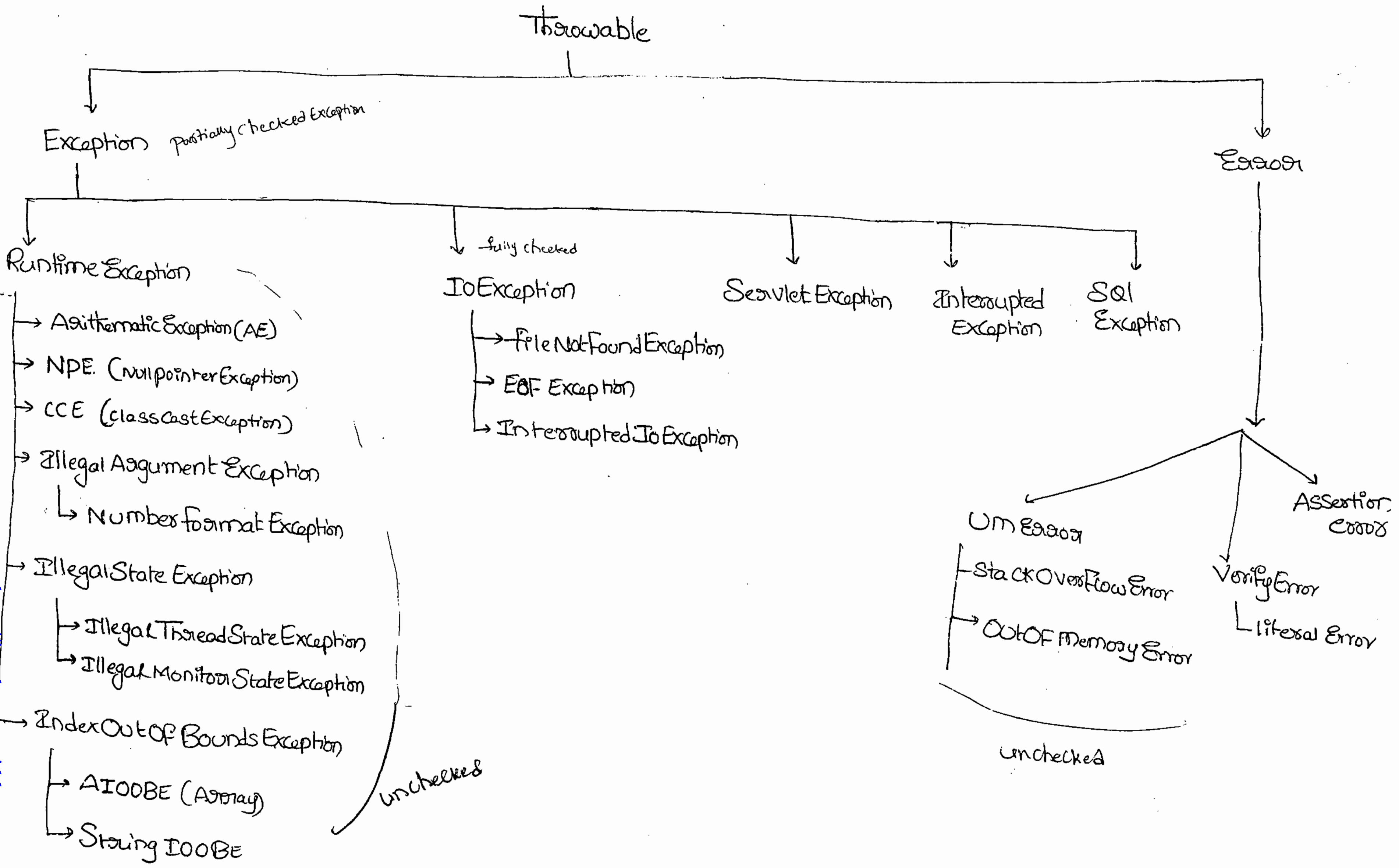
Ex:- Exception.

Q
(1) which of the following are checked

i) IOException	:	fully checked
2) Error	:	unchecked
3) Throwable	:	partially checked
4) Nullpoint Exception	:	unchecked
5) InterruptedIOException	:	fully checked
6) SqlException	:	fully checked.

Note!

→ In Java the only partially checked Exceptions are 1. Exception
2. Throwable.


Throwable
Exception
partially checked Exception
Error
RuntimeException
ArithmaticException (AE)
NPE (NullpointerException)
CCE (classCastException)
Illegal Argument Exception
Number format Exception
IllegalState Exception
Illegal ThreadState Exception
Illegal Monitor State Exception
IndexOutOfBounds Exception
AIOOBE (Array)
String IOOBE
unchecked
fully checked
IoException
FileNotFoundException
EOF Exception
InterruptedIoException
Servlet Exception
Interrupted Exception
SQl Exception
Vm Error
StackOverflowError
OutOF Memory Error
unchecked
Assertion Error
VerifyError
literal Error

Customized Exception Handling by Try-Catch:-

→ We Can maintain Risky Code with in the Try block & Corresponding Handling Code inside Catch block

```
try
{
   RiskyCode;
}
Catch (xxx  e)
{
  handling code.
}
```

```
Class Test
{
p.s.v.m(String[] args)
{
 S.o.pln("State1");
 S.o.pln(10/0);
 S.o.pln("State3");
}
}
```

o/p:- State1
R.E: A.E : / by Zero

Abnormal termination

```
Class Test
{
 p.s.v.m (String [] args)
 {
   S.o.pln ("State1");
   try
   {
    S.o.pln (10/0);
   }
   Catch(AE  e)
   {
    S.o.pln(10/2);
   }
   S.o.pln ("State3");
 }
}
```

o/p:- State1
5
State3
normal termination

Control flow in Try-catch :-

```
try
{
 Stat1;
 State2;
 State3;
}
Catch(xxx e)
{
 State4;
}
State5;
```

Case1 :-

→ If there is no Exception 1,2,3,5 statements are normal terminations

Case2 :-

→ If the Exception raised at Statement2 & Corresponding Catchblock matched, 1,4,5 are normal terminations

Case3 !-

→ If an Exception raised at Statement2 & the Corresponding Catchblock not matched, 1 followed by Abnormal Termination.

Case4 !-

→ If an Exception raised at Statement 4 or Statement 5 it is always A.NT Abnormal Termination

Note !-

→ with in the Try block if any where an Exception raised then rest of the try block won't be Executed even though we handled that Exception. Hence, it is recommended to take only Risky Code with in the Try block. & Length of the Try block should be as less as possible.

2. If an Exception raised at any Statement which is not Part of Try Then it is always Abnormal termination.

Various Methods to Print Exception Information :- 16/02/11

→ Throwable class defines the following methods to print Exception information.

(1) printStackTrace() :-

→ This method prints Exception information in the following formatt.

Name of Exception : discription ~~follow~~ by Stack trace

(2) toString() :-

→ It prints Exception information in the following formatt.

Name of Exception : discription

(3) getMessage() :-

→ This method prints only discription of the Exception.

discription

Ex:-

```
class Test
{
 p.s.v.m(String [] args)
 {
 try
  {
   S.o.pln(10/0);
  }
 catch(A.E e)
 {
  e.printStackTrace();                         → A.E : / by zero
                                                   at test.main()
  S.o.p(e); (or) S.o.pln(e.toString());        → A.E : / by zero
  S.o.pln(e.getMessage());                     → / by zero.
 }
 }
}
```

Note:-
→ default Exceptionhandler internally uses printStackTrace().

Try with Multiple Catch blocks :-

→ The way of handling an Exception is varied from Exception to Exception hence for every Exception it is recommended to take seperate Catch block.

Ex:-

```
try
{
 ---
}
catch(Exception e)        (but not recommended.)
{
 ---
}
```

Ex(2):-

```
try
{
 ---
 ---
 ---
}
catch(A.E e)
{
 perform these Arithmetic operations;
}
catch(FileNotFoundException e)
{
 use local file;
}
catch(NPE e)
{
 use Another resource
}
catch(Exception e)
{
 default Exception handler;
}
```

Highly recommended

→ Hence Try with multiple catch blocks is possible & highly recommended to use.

→ If Try with multiple catch blocks present then order of catch blocks is very important. and it should be from child to parent.

→ If we are taking from parent to child then we will get compile time error saying, "Exception xxxxx has already been caught"

$\boxed{\text{Child to parent is follows}}$

```
try                              try
{                                {
=                                =
=                                =
}                                }
Catch(Exception e)               Catch(A.E e)        ✓
{                                {
=                                }
}                                Catch(Exception e)
Catch(A.E e)                     {
{                                }
=
}
```

X

C.E :- Exception java.lang.A.E has already been Caught

finally Block :-

→ It is never recommended to define Clean-up Code with in the Try block. because there is no garenty for the Execution of Every Statement.

→ It is never recommended to define Clean-up Code with in the Catch-block, because it won't be Executeded if there is no Exception.

→ We required a place to maintain Clean-up Code which should be Executed always irrespective of wheather Exception raised or not raised & wheather handle or not handle, Such type of place is nothing but finally-block.

→ Hence, the main purpose of finally-block is to maintain Clean-up Code which should be Executed always.

Ex1:-
```
try
{
  risky code;
}
catch(xxx e)
{
  handling code;
}
finally
{
  clean-up code;
}
```

Ex②:-

```
class Test
{
p.s.v.m(String []args)
{
 try
 {
  s.o.pln("try");
 }
 catch(AE e)
 {
  s.o.pln("catch");
 }
 finally
 {
  s.o.pln("finally");
 }
}
}
```

o/p:- try
finally

```
class Test
{
p.s.v.m(String []args)
{
 try
 {
  s.o.pln("try");
  s.o.pln(10/0);
 }
 catch(AE e)
 {
  s.o.pln("catch");
 }
 finally
 {
  s.o.pln("finally");
 }
}
}
```

o/p:- Try
Catch
finally

```
class Test
{
p.s.v.m(String []args)
{
try
{
 s.o.pln("try");
 s.o.pln(10/0);
}
Catch(NullPointerEx e)
{
 s.o.pln("catch");
}
finally
{
 s.o.pln("finally");
}
}
}
```

o/p:- try
finally
Abnormal

return vs finally:-

→ finally block dominates return statement also. Hence, if there is any return statement present inside Try or Catch block. first finally will be Executed & then return statement will be Considered.

Ex:-
```
class Test
{
p.s.v.m(String [] args)
{
try
{
 S.o.pln("try");
 ~~So~~return;
}
Catch(A.E e)
{
 S.o.pln("catch");
}
finally
 {
  S.o.pln("finally");
 }
}
}
```

O/P:-
```
try
finally.
```

** → There is only one Situation where the finally-block won't be Executed is, when ever JVM shutdown. i.e, when ever we are using System.exit(0)

(**) Ex:-
```
Class Test
{
 p.s.v.m(String [] args)
 {
 try
  {
   S.o.pln("try");
   System.exit(0);
  }
 Catch(AE e)
  {
   System.out.println("catch");
  }
 finally
  {
  S.o.pln("finally");
  }
 }
}
```
O/P:- try

**difference b/w final, finally & finalize:-

final:-

→ It is a modifier applicable for classes, methodes & variables.

→ If a class declared as final, then child class creation is not possible.

→ If a method declared as final, then overridding of that method is not possible.

→ If a variable declared as the final, then reassignment (changing the value) is not allowed because, it is a Constant.

finally :-

→ It is block always associated with try-catch to maintain Clean-up Code which should be Executed always irrespective of wheather exception raised or not raised & wheather handleded or not handeled.

finalize():-

→ It is a method which should be Executed by Garbage Collector before destroying any object to perform Clean-up activities.

Note:-

→ when Compare with finalize(), it is highly recommended to use finally block to maintain clean-up Code. Because, we Can't Expect exact behaviour of the Garbage Collector.

Various possible Combinations of try-Catch-finally :-

① ✓
```
try
{
}
Catch(xx e)
{
}
```

② ✓
```
try
{
}
Catch(xx e)
{        child
}
Catch(yy e)
{        parent
}
```

③ ✓
```
try
{
}
finally
{
}
```

④ ✗
```
try
{
}
```
C.E:- Try with out Catch or finally

⑤ ✗
```
Catch(xx e)
{
}
```
C.E:- Catch with out try

⑥ ✗
```
finally
{
}
```
C.E:- finally without try

⑦ ✗
```
try
{
}
S.o.pln("Hello");
Catch(xx e)
{
}
```
C.E:- Try without Catch or finally
C.E:- catch without try

⑧ ✓
```
try
{
}
catch(xx e)
{
}
S.opln("Hello");
Catch(xxe)
{
}
```
C.E:- catch without try.

⑨
```
try
{
}
catch(xx e)
{
}
S.o.pln("Hello");
finally
{
}
```
✓ ×

C.E:- finally without try

⑩
```
try
{
}
catch(xx e)
{
}
finally
{
}
finally
{
}
```
✓ ×

C.E:- finally without try

⑪
```
try
{
}
catch(AE e)
{
}
catch(Exception e)
{
}
```
✓

⑫
```
try
{
}
catch(Exception e)
{
}
catch(A.E e)
{
}
```
C.E:-
Exception Java.lang.AE has already been Caught

⑬
```
try
{
}
catch(AE e)
{
}
catch(AE e)
{
}
```
C.E:-
Exception Java.lang.AE has already been Caught

⑭
```
try
{
}
catch(xx e)
{
   try
   {
   }
   catch(yy e)
   {
   }
}
```
✓

⑮
```
try
{
}
catch(xx e)
{
}
finally
{
   try
   {
   }
   ~~finally~~(yy e)
   {
   }
}
```
✓

⑯
```
try
{
   try
   {
   }
}
catch(xx e)
{
}
```
C.E:- try without catch or finally

⑰
```
try
{
}
finally
{
}
catch(x e)
{
}
```
✓ ×

C.E:- catch without try

Control-flow in try-catch-finally :-

```
try
 {
  Stat 1;
  Stat 2;
  Stat 3;
 }
Catch(Xxx e)
 {
 Stat 4;
 }
finally
 {
  statement 5;
 }
Statement 6;
```

Case 1:-

→ If there is no Exception, then 1, 2, 3, 5, 6, normal termination.

Case 2!-

→ If an Exception raised at Statement 2 & the Corresponsiding Catch-block Matched. 1, 4, 5, 6, normal termination.

Case 3:-

→ If an Exception raised at Statement 2 & The Corresponding Catch-block not matched. 1, 5, Abnormal termination.

Case 4!.

→ If an Exception raised at Statement 4, then it is always abnormal termination but before that finally block to be Executed.

Case 5!.

→ If an Exception raised at State 5 or State 6, it is always abnormal terminat

Control flow in Nested try-Catch-finally :-

```
try
{
  State 1;
  State 2;
  State 3;
       try
       {
        State 4;
        State 5;
        State 6;
       }
       Catch(xx  e)
       {
        State 7;
       }
       finally
       {
        State 8;
       }
   State 9;
}
Catch(yy  e)
{
State 10;
}
finally
{
 State 11;
}
State 12;
```

Case 1:-

→ If there is no Exception, Then 1,2,3,4,5,6,8,9,11,12, Normal termination

Case 2:-

→ If an Exception raised at Statement 2 and Corresponding Catch block Matched. Then 1,10,11,12, Normal termination

Case 3:-

→ If an Exception raised at Statement 2 and Corresponding Catch block not matched. Then 1,11, abnormal termination.

Case 4:-

→ If an Exception raised at Statement 5 & Corresponding inner Catch has matched 1,2,3,4,7,8,9,11,12, Normal termination.

Case 5:-

→ If an Exception raised at Statement 5 & Corresponding inner Catch has not matched but outer Catch has matched. Then 1,2,3,4,8,10,11,12, Normal

Case 6:-

→ If an Exception raised at State 5 & inner & outer Catch blocks are not matched Then 1,2,3,4,8,11, Abnormal

Case 7:-

→ If an Exception raised at State 7 & Corresponding Catch block matched Then 1,2,3,...,8,10,11,12, Normal

Case 8:-

→ If an Exception raised at Statement 7 & The Corresponding Catch not matched Then 1,2,3,...,8,11, Abnormal.

Case 9 :-

→ If an Exception raised at State 8 & Corresponding Catch matched

Then 1, 2, 3 ..., 10, 11, 12, Normal

Case 10:-

→ If an Exception raised at State 8 & Corresponding Catch has not matched.

Then 1, 2, 3 ----, 11, Abnormal

Case 11:-

→ If an Exception raised at State 9 & Corresponding Catch matched.

then 1, 2, 3, ..., 8, 10, 11, 12, Normal

Case 12:-

→ If an Exception raised at State 9 & Corresponding Catch block not matched Then 1, 2, 3, ..., 8, 11, Abnormal

Case 13:-

→ If an Exception raised at State 10 it is always Abnormal termination but before the finally-block will be executed.

Case 14:-

→ If an Exception raised at State 11 or State 12 it is always Abnormal termination.

18/02/11

Throw :-

→ Some times we can Create Exception Object manually & hand-over that object to the JVM Explicitly by using throw keyword.

throw new ArithematicException("/ by zero");

$$\underbrace{\text{throw}}\quad \underbrace{\text{new ArithematicException("/ by zero");}}$$

To hand-over our Created Exception object to the JVM manually.

Creation of A.E object Explicitly

→ Hence, the main purpose of throw key-word is to hand-over our Created Exception object manually to the JVM.

→ The Result of following two programs is Exactly Same.

```
class Test
{
 p.s.v.m(String []args)
 {
  S.o.pln (10/0);
 }
}
```

```
class Test
{
 p.s.v.m (String []args)
 {
  throw new ArithmeticException("/by
                       zero");
 }
}
```

→ In this Case A.E object Created internally & hand-over that object automatically by the main().

→ In this Case we Created A.E object and we hand-over it to the Jvm manually by using throw-keyword.

→ In Generall, we Can use throw keyword for Customized Exceptions.

Case 1:-

→ If we are trying to throw null reference, we will get NullPointerException

```
class Test
{
Static A.E e;
P.S.V.m (String [] args)
{
throw e;
}
}
```

RE:- NPE

```
Cass Test
{
Static A.E e = new A.E ();
P.S.V.m (String [] args)
{
throw e;
}
}
```

R.E: A.E

Case 2:-

→ After throw Statement we are not allow to write any Statement directly otherwise we will get ~~Compiletie~~ Compiletime Error Saying "unreachable Statement".

```
class Test
{
P.S.V.m(String [] args)
{
S.o.pln(10/0);
S.o.pln("Hello");
}
}
```

R.E:- AE / by Zero

```
class Test
{
P.S.V.m(String [] args)
{
throw new A.E ("/ by zero");
S.o.pln(" Hello");
}
}
```

C.E:- unreachable Statement.

Case 3 :-

→ We Can use throw keyword Only for throwable type otherwise we will get Compiletime Error Saying InCompatiable ~~State~~ types.

```
Class Test
{
 p.s.v.m(String []args)
 {
  throw new Test();
 }
}
```

C.E: InCompatiable Types
found : Test
Required: Java.lang.throwable

```
Class Test extends RuntimeExceptions
{
 p.s.v.m(String [] args)
 {
  throw new Test()
 }
}
```

R.E:
Exception in thread main : Test.

Throws :-

→ In our program, if there is any chance of raising CheckedException Commpulsory we should handle it, Other wise will get Compiletime Error Says "unreported Exception xxxxx must be Caught or declare to be thrown".

Ex :-
```
class Test
{
 p.s.v.m(String [] args)
 {
  thread.sleep (5000);
 }
}
```

C.E :- unreported Exception Java.lang.IE must be Caught

→ we Can handle this by using the following two-ways.

(1) By using Try-catch

(2) " " throws

(1) By using Try-catch:-

```
Class Test
{
 p.s.v.m(String []args)
 {
   try
   {
    Thread.sleep(5000);
   }
   Catch (I.E e)
   {
   }
}
```

✓

(2) By using throws keyword !-

→ we Can use throws keyword to delegate the responsibility of Exception handling to the ~~Handler~~ Caller method.

```
class Test
{
 p.s.v.m(String [] args) throws IE
 {
  thread.sleep(5000);
 }
}
```

✓

→ Hence, The main purpose of Throws Keyword is to delegate responsibility of Exception handling to the Caller methodes in the Case of checked Exception, to Convence Compiler.

→ In The Case of unchecked Exceptions, it is not required to use throws Keyword.

Ex:

```
class Test
{
 p.s.v.m (String [] args)  throws IE
 {
 doStuff();
 }
p.s.v. doStuff() throws IE
{
 doMoreStuff();
 }
p.s.v. doMoreStuff() throws IE
{
  Thread.sleep(5000);
 }
}
```

→ In the above program, If we are removing any throws keyword, The Code won't be Compiles. Compulsory we should use 3 throws Statements.

We can use throws keyword only for Throwable types otherwise we will get compile-type Error saying, incompatable types

```
class Test
{
X p.v.m1() throws test
  {
  }
}
```

C.E:- incompatable type
found : Test
Required : java.lang.throwable

```
Class Test extends ~~Throwable~~ Exception
{
p.v.m1() throws Test
{
}
}
```

✓

Case(1):-

(Checked)

```
class Test
{
  p.s.v.m(String []args)
  {
  throw new Exception();
  }
}
```

CE: unreported Exception java.lang.Exception must be caught at declared to be thrown.

→ As Exception is checked compulsory we should handle either by Try-catch or by throws keyword

(unchecked)

```
class Test
{
 p.s.v.m(String[]args)
 {
 throw new Error()
 }
}
```

R.E:- Exception in thread "main" java.lang.Error.

→ As Error is unchecked, it is not required to handle by Try-catch or by throws

Case 2:-

→ In our program, if there is no chance of raising an Exception then, ~~it is not~~ we can't define catch block for that Exception otherwise we will get Compiletime Error. but this rule is applicable for only fully checked Exceptions.

Ex:-

```
try
{
S.o.pln("Hello");
}
Catch(A.E e)
{
}
//Hello ✓
```

```
try
{
S.o.pln("Hello");
}
Catch(Exception e)
{
}
o/p:- Hello ✓
```

```
try                                   ✗
{
S.o.pln("Hello");
}
Catch(IOException e)
{
}
```

C.E:- Exception java.lang.IOException is never thrown in body of corresponding try statement.

```
try         ✗
{
S.o.pln("Hello");
}
Catch (InterruptedException e)
{
}
C.E:
```

```
try ✓
{
S.o.pln("Hello");
}
Catch(Error e)
{
}
o/p:- Hello
```

Keywords for Exception:

try
catch
finally
throw
throws

Exception Handling Keywords Summary :-

1) try :- To maintain Risky code.

2) catch :- To maintain Handling Code.

3) finally :- To maintain Clean-up Code.

4) throw :- To hand-over our Created Exception object to the JVM manually.

5) throws :- To delegate the Responsibility

Various Possible Compiletime Error in Exception Handling :-

① Exception xxxxx has already been Caught (try with multiple Catch)

② unreported Exception xxxx must be caught or declared to be thrown

③ Exception xxxx is never thrown in body of Corresponding try Statement

④ try without catch or finally

⑤ finally without try

⑥ Catch without try

⑦ unreachable Statement

⑧ Incompatable types

found : Test

required : Java.lang.Throwable.

Customized Exceptions :

→ To meet our programming requirement Sometimes we have to create Our own Exceptions. Such types of Exceptions are Called "Customized Exceptions".

Eg: TooYoungException, TooOldException, ~~InSufficientFundsExceptions~~..etc

```
Class TooYoungException extends RuntimeException
{
   TooYoungException(String s)
   {
     Super(s);
   }
}
Class TooOldException extends RuntimeException
{
  TooOldException(String s)
  {
    Super(s);
  }
}
class Test
{
p.s.v.m(String[] args)
{
 int age = Integer.parseInt(args[0]);
 if(age >60)
 {
 throw new TooYoungException("plz wait Some more time"); ??
                              age is already Crossed Marriage age");
 }
else if(age <18)
{
 throw new TooYoungException("Ur age is already crossed marriage
                              age .. [illegible]
else
{
  S.o.pln("you will get match details by mail");
  }
}.
```

<u>Note</u>:-

→ It is highly recommended to keep our Customized Exception class as unchecked, i.e we have to Extend runtime Exception Class but not Exception class while defining our customized Exceptions.

Top-10 Exceptions :-

<u>21-02-11</u>

→ Based on the Source, who triggares the Exception, all Exceptions are devided into 2 types.

1. J.V.M Exceptions
2. programmatic Exceptions.

1. JVM Exceptions :-

→ The Exceptions which are raised automatically by the JVM when ever a particular Event occurs are called JVM Exceptions.

Ex:- (i) ArrayIndexOutOfBoundsException.

(ii) NullPointerException.....

2. programatic Exceptions :-

→ The Exceptions which are raised Explicitly either by the programmer or by the API developer, are Called programmatic Exception.

Ex:- IllegalArgumentException, NumberFormatException . . .

(1) ArrayIndexOutOfBoundsException :-

→ It is the child class of RuntimeException & hence it is unchecked.

→ Raised automatically by the JVM, whenever we are trying to access Array element with out ~~of~~ range index.

Ex:-
```
int [] a = new int[10];
    S.o.pln(a[0]);  0 ✓
    S.o.pln(a[100]);  R.E:- AIOOBE
```

(2) NullPointerException :-

→ It is the child class of Runtime Exception and hence it is unchecked.

→ Raised automatically by the JVM, when ever we are trying to ~~access~~ perform any operation on Null.

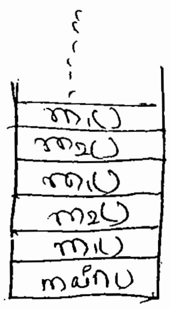
Ex:-
```
String s = null;
    S.o.p(s.length());  RE: NPE
```

(3) StackOverFlowError :-

→ It is the child class of Error and hence it is unchecked.

→ Raised automatically by the JVM, when ever we are trying to perform recursive method invocation.

Ex:-
```
Class Test
{
  p.s.v.m m1()
  {  m2();
  }
  p.s.v.m m2()
  {  m1();
  }
                p.s.v.m (String [] args)
                {  m1();
                }
}
```



R.E: SOFE

(4) NoClassDefFoundError :-

→ It is the child class of Error and hence it is unchecked

→ raised automatically by the JVM, when ever JVM unable to find required class.

Ex: Java Srinu ↵

→ If Srinu.class file is not available then we will get R.E Saying NoClassDefFoundError.

(5) ClassCastException :-

→ It is the child class of RuntimeException and hence it is unchecked.

→ raised automatically by JVM whenever we are trying to typeCast parent object to the child type.

Ex):-

```
String s = new String("durga");
Object o = (Object) s;
```
✓

~~Strin~~
```
Object o = new Object();
String s = (String) o;
```
X

R.E :- CCE

(6) ExceptionInInitializerError :-

→ It is the child class of Error and hence, it is unchecked

→ raised automatically by the JVM, if any Exception occurs while performing initialization for static variables and ~~by~~ while Executing static blocks.

Eg:-

```
Class Test
{
  Static int i = 10/0;
}
```

R.E:-
ExceptionInInitializerError
Caused by java.lang.AE : / by zero.

```
Class Test
{
  Static
  {
   String s = null;
   S.o.pln(s.length());
  }
}
```

R.E:- ExceptionInInitializerError
Caused by java.lang.NPE

7) Illegal Arguement Exception :-

→ It is the child class of RE & hence it is unchecked.

→ Raised Explicitly by the programmer or by API developer to indicate that a method has been invoked with invalid arguement.

Ex:-
```
Thread t = new Thread();
t.setPriority(10); ✓
t.setPriority(100); ✗ R.E: IAE
```

8) NumberFormattException

→ It is the child class of R.E & hence it is unchecked.

→ Raised Explicitly by the programmer or by API developer to indicate that we are trying to convert String to number type but the String is not properly formatted

$$\begin{tikzcd} RE \\ IAE \arrow[u] \\ NFE \arrow[u] \end{tikzcd}$$

Eg:-
```
✓ int i = Integer.parseInt("10");
✗ int i = Integer.parseInt("Ten"); R.E: NFE
```

⑨ IllegalStateException :-

→ It is the child class of RuntimeException and hence, it is unchecked.

→ Raised Explicitly by the programmer or by the API developer to indicate that a method has been invoked at in appropriate time.

Ex:-

One Session Expires we Can't Call any method on that object otherwise we will get IllegalStateException.

Ex①:-

```
HttpSession session = req.getSession();
S.o.pln(session.getId());   1234---  ✓

Session.invalidate();
S.o.pln(Session.getId());   R.E:- ISE   ✗
```

Ex②:-

```
Thread t = new Thread();
   t.start(); ✓
      |
      |
      |
✗  t.start();  R.E:- IllegalThreadStateException.
```

→ After Starting a thread, we are not allowed to restart the Same thread, otherwise we will get R.E:- IllegalThreadStateException

10) AssertionError:-

→ It is the child class of Error & hence it is unchecked.

→ Raised Explicitly either by the programmer or by API developer to indicate that ~~a method has~~ assert statement fails.

Ex:- Assert(false);

R.E:- AssertionError.

Exception/error	Raised by
1. AIOOBE 2. NPE 3. SOFE 4. NoClassDefFoundError 5. ClassCastException 6. ExceptionInInitializerError	JVM automatically (JVM Exception)
7. IllegalArgumentException 8. NumberFormatException 9. IllegalStateException	either programmer or API developer Explicitly (Programatic Exceptions)

Exception Propagation:-

→ The process of delegating the Resposibility Exception handling from one method to another method by using throws keyword is Called Exception propagation

22/02/11

Inner classes

→ Sometimes we can declare a class inside another class, such type of classes are called 'Innerclasses'.

difficult or doubtful situation.

→ Innerclasses concept introduced in Java1.1 version to fix GUI bugs as the part of Eventhandling.

→ But because of powerfull features & benefits of Innerclasses slowlly programmers started using even in regular coding also.

→ without existing one type of object if there is no chance of existing another type object, then we should go for Inner class concept.

Ex(1):-

① without existing Car object, if there is no chance of existing wheel object then we should go for Inner classes.

② we have to declare wheel class with in the Car class.

```
class Car
{
  class Wheel
  {
  }
}
```

②:- without existing Bank object there is no chance of existing account object, Hence we have to define account class inside Bank class.

```
class Bank
{
  class Account
  {
```

(3) -A map is a Collection of Key value pairs and each Key-value pair is called Entry. without existing map object there is no chance of existing Entry object. hence interface Entry is define inside Map interface.

```
interface Map
{
  interface Entry
  {
  }
}
```

Note:-

→ The Relationship b/w Outer & inner classes is not parent to child Relationship. It is has-A Relationship.

→ Based on the purpose & position of declaration all innerclasses are divided into 4 types

1) Normal (or) Regular Inner classes.
2) Method Local Inner classes
3) Annoymous Inner classes (without class name)
4) Static Nested Classes

Note:-
From static Nested class we can access only static members of outer class directly. But in normal inner classes we can access both static & Non-static members of outer class directly.

Normal or Regular Inner class :-

→ If we declare any named class directly Inside a class without Static modifier, Such type of class is Called "Normal or Regular Inner class"

Ex(1):-

```
Class Outer
{
  Class Inner
  {
  }
}
```

$$\begin{tikzcd}
 & \text{Javac Outer.Java} \arrow[dl] \arrow[dr] & \\
\text{Outer.class} & & \text{Outer\$Inner.class}
\end{tikzcd}$$

```
Java Outer ↵
  R.E:- NoSuchMethodError : main
Java Outer$Inner
  R.E:- NoSuchMethodError :main
```

Ex(2) :-

```
Class Outer
{
  Class Inner
  {
  }
  public static void main(String [] args)
  {
   S.o.pln(" Outer class main method");
  }
}
```

O/P. Javac Outer.java

```
Java Outer ↵
    O/P!. Outer Class main method.
Java Outer$Inner ↵
    O/P!- NoSuchMethod Error : main.
```

Ex(3):

→ Inside Inner classes we can't declare static members hence it is not possible to declare mainmethod & hence we can't invoke inner class directly from command prompt.

Ex:-
```
class Outer
{
 class Inner
 {
  p.s.v.m(String [] args)        X
  {
   S.o.pln("inner class method");
  }
 }
}
```

Javac Outer.java

C.E:- Inner classes can't have static declarations

23/02/11:-

Accessing Inner class Code from Static area of outer class:-

Ex:-
```
class Outer
{
 class Inner
 {
  p.v.m1()
  {
   S.o.pln("Inner class method");
  }
 }
 p.s.v.m(String [] args)
 {
  Outer o = new Outer();
  Outer.Inner i = O.new Inner();
```

```
            i.m1();
        }
    }
```

o/p:- Javac Outer.java ↵

java Outer ↵

Inner class method.

```
Outer o = new Outer();                 ⎫ → Outer.Inner i = new Outer().new Inner();
Outer.Inner i = o.new Inner();         ⎬
i.m1();                                ⎭ → new Outer().new Inner().m1();
```

Accessing Inner class Code from Instance Area of outer Class:-

Eg.
```
Class Outer
{
  class Inner
  {
     p.v.m1()
     {
      S.o.pln("Inner class method");
     }
  }
  p.v.m2()
  {
    Inner i = new Inner();
    i.m1();
  }
  p.s.v.m (String [] args)
  {
    Outer o = new Outer();
  } o.m2();
}
```

Accessing Inner class Code from Outside of outer Class :

Eg:

```
Class Outer
{
   Class Inner
   {
     p.v.m1()
     {
      S.o.pln(" Inner Class method");
     }
   }
}
```

```
Class Test
{
  p.s.v.m(String [] args)
  {
    Outer o = new Outer();
    Outer.Inner i = o.new
                 Inner();
   i.m1()
  }
}
```

$$\begin{tikzcd}
 & \text{According Inner Class Code} \arrow[dl] \arrow[dr] & \\
\text{from static area of Outer class (or) from outside of outer class} & & \text{from Instance are of outer class}
\end{tikzcd}$$

```
Outer o = new Outer();
Outer.Inner i = o.new Inner();
i.m1();
```

```
Inner i = new Inner();
i.m1();

outer o = new Outer();
```

→ From The Inner class we Can access all members of Outer class (both Static & non-static) directly.

Ex:-
```
class Outer
{
    Static int x=10;
     int y = 20;
    class Inner
    {
      public void m1()
       {
         S.o.pln(x);     10
         S.o.pln(y);     20
       }
    }
  p.s.v.m (String [] args)
  {
   new Outer().new Inner().m1();
  }
}
```

O/P:-
```
10
20
```

→ With in the Inner class this always pointing to Current Inner class Object.

→ To refer Current Outer class object we have to use "Outerclassname.this"

"Outerclassname.this".

Ex:-

Eg:-

```
Class Outer
{
  int x=10;
  class Inner
  {
    int x=100;
    Public void m1()
    {
      int x=1000;
      S.o.pln(x); 1000
      S.o.pln(this.x); 100
      S.o.pln(Outer.this.x); 10
    }
  }
  p.s.v.m(String [] args)
  {
   new Outer().new Inner().m1();
  }
}
```

→ For the Outer classes (Top-level classes) the applicable modifiers are public, <default>, final, abstract, Strictfp.

But for the Inner classes In addition to above The following modifiers are also applicable.

only for Outer classes { public, default, final, abstract, Strictfp } + private, protected, Static = Inner classes

2) Method Local Inner classes :-

→ Some-times we can declare a class inside a method such type of classes are called "Method Local Inner classes".

→ The main purpose of method local Inner class is to define method specific functionality.

→ The scope of method Local Inner class is the method in which we declared it. that is from outside of the method we can't access method Local Inner classes.

→ As the scope is very less, this type of Inner classes are most rarely used Inner classes.

Ex:

```
Class Test
{
    Public void m1()
    {
        Class Inner
        {
            public void sum(int x, int y)
            {
                S.o.pln("sum is :"+(x+y));
            }
        }
        Inner i = new Inner();
        i. sum(10, 20);
        i. sum(100, 200);
        i. sum(1000,2000);
        i. sum(10000, 20000);
    }
p.s.v.m(String [] args)
{
  new Test().m1();
}
}
```

O/P:-

```
Sum is 30
Sum is 300
Sum is 3000
Sum is 30000
```

→ We Can declare Inner class either in Instance method or in Static-method.

→ If we declare Inner class inside Instance method then we Can access Both Static & NON-Static Variables of outerclass directly from that Inner class.

→ If we declare Inner class inside Static method then we Can access Only Static members of OuterClass directly from that Inner class.

Ex:-

```
class Test
{
 int x=10;
 Static int y=20;
 public void m1()   → if Static is there
 {
  class Inner
  {
   p. void m2()            o/p:- 20
   {
    S.o.pln(x);  10
    S.o.pln(y);  20
   }}
  Inner i = new Inner();
  i.m2();
}}
```

```
p.s.v.m (String [] args)
{
 new Test().m1();
}
}
```

o/p!- 10, 20

→ from method local Inner class we can't access local variables of the method in which we declared it. But if that local variable declared as the final then we can access.

Eg:-

```
class Test
{
 int x=10;
 public void m1()
 {
  int y=20;        → if we declare final
                     ( final int y=20;)
                     o/p:- x=10
                           y=20
  class Inner
  {
   public void m2()
   {
    System.out.println(x);
    System.out.println(y);
   }
  }
  Inner i = new Inner();
  i.m2();
 }
p.s.v.m (String [] args)
{
 new Test().m1();
}
}
```

o/p!
C.E:- Local variable y is accessed from with inner class; needs to be declared final.

→ If we declare y as final Then we won't get any Compiletime Error.

O/P:- x= 10
      y= 20

24/02/11:

Q) Consider The following Code

```
class Test
{
  int x=10;
  Static int y = 20;
  Public void m1()
  {
    int i = 30;
    final int j =40;

  Class Inner
  {
   public void m2()
   {
    ————→ Line①
   }
  }
 }
}
```

→ At line① which Variables we Can access
① x ✓
② y ✓
③ i ✗
④ j ✓

Note:- If declare m1() as Static then at Line① which variables we Can access as y, j.

② If we declare m2() as static, then which variable we can access Line①

we will get C.E, because inside inner classes we can't have static declarations.

→ The only applicable modifiers for method Local inner classes are final, abstract, strictfp,

(3) Annonymus Inner Class :-

→ Some-times we can declare a class without name also. Such type of nameless inner classes are called Annoymus inner classes.

→ This type of inner classes are most commonly used type of inner classes.

→ There are 3 types of Annonymus inner classes.

1. Annonymus inner class that extends a class.
2. " " " implements an Interface.
3. " " " defined inside method arguements.

① Annonymus Inner class that extends a class:-

Ex:-
```
Class popcorn
{
  public void taste()
  {
    S.o.pln("Salty");
  }
    //100 more methods
}
Class Test
{
  P.S.V.m(String[] args)
  {
```

```
Popcorn p = new Popcorn
{
  public void taste()
  {
   S.o.pln("Sweety");
  }
};
P.taste(); Sweety
Popcorn p1 = new Popcorn();
p1.taste(); Salty
  }
}
```

Note:-

① The internal class name generated for Annoymus Inner class is "Test$1.class".

② parent class reference can be used to hold child class object but by using that reference we can call only methods available in the parent class & we can't call child specific methods. In the annoymus inner classes also we can define new methods but we can't call these method from outside of the class because, ~~these are~~ we are depending on parent reference. this methods just for internal purpose only.

Analysis:-

```
popcorn p = new Popcorn();
```

→ Just we are creating an object of PopCorn class.

→ ~~Pop~~

```
PopCorn p = new popCorn()
{

};
```

→ we are creating child class for the popCorn & for that child class we are creating an object with parent reference.

Eg:

```
class Test
{
 p.s.v.m (String[] args)
 {
  Thread t = new Thread()
  {
   p.v. run()
   {
    for(int i=0 ; i<10 ; i++)
    {
     S.o.pln ("child Thread");
    }
   }
  };
  t.start();
  for(int i=0 ; i<10; i++)
  {
   S.o.pln("main Thread");
  }
 }
}
```

→ In the above Example both main & child threads will be Executed Simultaneously & Hence we can't ~~Exe~~ exact output.

(b) Anonymus Inner Class That implements an Interface :-

Ex :-

```
Class Test
{
 p.s.v.m(String [] args)
 {
  Runnable r =new Runnable()
  {
   public void run()
   {
    for (int i=0 ; i<10 ; i++)
    {
     S.o.pln("child thread");
    }
   }
  };
  Thread t = new Thread (r);        It is an object of
                                     Runnable
  t.start();
  for (int i=0 ; i< 10; i++)
  {
   S.o.pln(" main Thread");
  }
 }
}.
```

(c) Anonymous Inner Class that define Inside method argument :-

Ep:-
```
Class Test
{
 Public static void main (String [] args)
 {
  New Thread (new Runnable()
                    {
                     public void run()
                     {
                     for (int i=0 ; i<10 ; i++)
                     {
                      S.o.pln ("child thread-1");
                     }
                    }
                   }).start();

  for (int i=0 ; i<10 ; i++)
  {
   S.o.pln (" main thread -1);
  }
 }
}
```

General class Vs Anonymus Inner class:-

→ A General class can extend only one class at a time. where as Anonymos Innerclass also can extend only one class at a time.

→ A General class can implement any no. of Interfaces where as Annoymus Innerclass can implement only one interface at a time.

→ A General class can Extend another class & can implement an interface simultaneously. where as Annonymus Inner class can extend another or can implement an interface but not both simultaneously.

) Static Nested classes:-

→ Some times we can declare Inner class with static modifier Such type of Inner classes are called "Static Nested classes".

→ In the normal Inner class, Inner class object always associated with outer class object.

→ ie, with out existing outer class object, there is no chance of existing Inner class object.

→ But Static Nested class object is not associated with Outerclass object, i.e with out existing outer class object there may be a chance of existing static Nested class object.

Ex1:-
```
class Outer
{
  static class Nested
  {
   public void m1()
   {
    S.o.pln("static Nested class method");
   }
  }
public static void main(String[] args)
{
  Outer.Nested n = new Outer.Nested();
  n.m1()
}
}
```

→ With in the Static Nested Class we can declare static members including main() also. Hence it is possible to invoke Nested Class directly from Command prompt.

Ex:
```
Class Outer
{
  Static class Nested
  {
    public static void main(String[] args)
    {
      S.o.pln("Static Nested class main method");
    }
  }
  public ~~static~~ Static void main(String[] args)
  {
    S.o.pln(" Outer class main method");
  }
}
```

```
Javac Outer.java ↵
Java Outer ↵
   Outer class main method
Java Outer$Nested ↵
   Static Nested class main method
```

→ from the Normal Inner class both Static & Non-static members directly. but from, Static Nested class we can access only static members of outer class directly.

Ex:-

```
class Outer
{
  int x=10;
  Static int y=20;

  Static class Nested          ✗
  {
    p.v.m()
    {
      S.o.pln(x); ✗ ——→ C.E:- Non-static variable x can't be
      S.o.pln(y); ✓           referenced from static ~~class~~ Nested Content
    }
  }
}
```

diff b/w Normal Inner class & Static Nested Class ?

Innerclass	Static Nested Class
1) Inner class object is always associated with Outerclass object. i.e without existing outerclass object there is no chance of existing Innerclass object	1) Static Nested Class object is not associated with Outer class object, i.e, without existing Outer class object there may be a chance of existing Static Nested class object.
2) Inside Normal Inner class we can't declare Static members	2) Inside Static Nested class we can declare static members.
3) Inside normal Inner class we can't declare main() and hence it is not Possible to invoke inner class directly from Commandprompt	4) Inside static Nested class we can declare main() & hence we can invoke Static Nested Class directly from Command prompt.

Java . lang package

→ The most Commonly required classes & Interfaces which are required for writing any java program whether it is simple or Complex, are encapsulated into a Seperate package which is nothing but lang package

→ It is not required to import lang package explicitly because bydefault it is available to every java program.

→ The following are some of the commonly used classes in lang Package

1. Object
2. String
3. StringBuilder
4. StringBuffer
5. ~~W~~Wrapper classes (Auto boxing & Auto unboxing)

1. Object :-

→ The most Common methods which are required for any java object are encapsulated into a Seperate class which is nothing but object class.

→ SUN people make this class as parent for all Java classes so that its methods are bydefault available to every Java Class Automatically

→ Every class in java is the child class of object either directly or Indirectly, if our class doesnt extend any other class then only our class is direct child class of Object.

Ex:- Class A

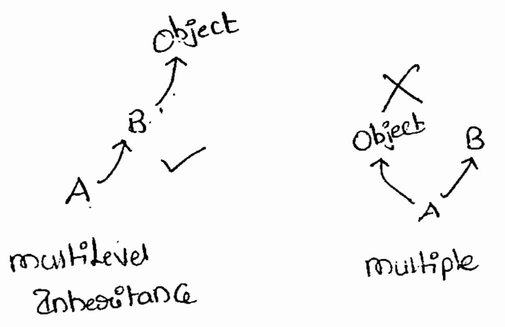
```
{
  =
}
```

$$\begin{tikzcd} & \text{Object} \\ A \arrow[ur] & \end{tikzcd}$$

→ if our class extends any other class then our class is not direct child class of Object. it extends object class indirectly.

Ex:- Class A extends B

```
{
  =
}
```



→ Object class defines the following 11 methods

(1) public String toString()

(2) public native int hashCode()

(3) public boolean equals(Object o)

(4) protected native Object clone() throws CloneNotSupportedException

(5) public final Class getClass();

(6) protected void finalize() throws Throwable

(7) public final void wait() throws InterruptedException

(8) public final native void wait(Long ms) throws IE

(9) public final ~~native~~ void wait (long ms, int ns) throws IE

(10) public final native void notify();

(11) public final native void notifyAll();

① toString() method :-

→ we can use this method to find String representation of an Object

→ whenever we are trying to print any object reference internally toString() method will be executed.

Ex:-

```
Class Student
{
  String name;
  int rollno;

  Student(String name, int rollno)
  {
     this.name = name;
     this.rollno = rollno;
  }
  p.s.v.m(String[] args)
  {
    Student s1 = new Student("durga", 101); ✓
    Student s2 = new Student("Ravi", 102); ✓
    S.o.pln(s1);  (or) ⇒ S.o.pln(s1.toString());   Student@3e25a5
    S.o.pln(s2);                 Student@19821f.
  }
}
```

→ In the above case Object class toString() method got executed which is implemented as follows.

```
Public String toString()
{
  return getClass().getName + "@" + Integer.toHexString(hashCode());
}
                 Student        @   3e25a5
```

→ To-P.
→ Classname@hexadecimal String representation of hash Code.

→ To provide our own String representation we have to override toString() in our class which is highly recommended.

→ when ever we are trying to print Student Object reference to return his name & roll number we have to override toString() as follows

```
public String toString()
{
 // return name;
 //return name + "-----" + rollno;
 // return "this is Student with name:" + name + ", with rollno:"
                                                  + roll no;
}
```

*, In String, StringBuffer & In all wrapper classes toString() method is Overridden to return proper String form. Hence, it is highly recommended to override toString() method in our class also.

Ex :-

```
Class Test
{
  ~~p s v m~~
  Public String toString( )
  {
    return "test";
  }
  Public . s. v. m( ——)
  {
   Test t = new Test();
   String s = new String("durga");
   Integer i = new Integer(10);
   S.o.pln(t);      test                    test@a235a4
   S.o.pln(s);      durga
   S.o.pln(i);      10
  }
}
```

(ii) hashCode() :-

→ for every object Jvm ~~will always~~ will assign one unique id. which is nothing but hashCode().

→ Jvm uses hashCode will saving objects into hashtable or hashSet or hashmap

→ Based on our requirement we can generate hashCode by overriding hashCode() method in our class.

→ If we are not overriding hashCode() method then Object class hashCode() method will be executed which generates hashCode based on Address of the Object But whenever we are overriding hashCode() method then hashCode is no longer related to Address of the Object.

→ Overriding hashCode() method is said to be proper iff for every Object we have to generate a unique number.

Ex:- ① Case ①:-

```
Class Student
{
  ---
  ---
  ---
  public int hashCode()
  {
    return 100;
  }
  :
}
```

Case ②:-

```
Class Student
{
  ---
  ---
  public int hashCode()
  {
    return rollno;
  }
  :
}
```

Case ①:- It is improper way of overriding hashCode() because we are generating same hashCode for every object

Case ②:- It is proper way of overriding hashCode() because we are generating a different hashCode for every object.

toString() Vs hashCode() :-

Ex:-

```
Class Test
{
   int i;
   Test(int i)
   {
     this.i = i;
   }
   p.s.v.m(——)
   {
    Test t1 = new Test(10);
    Test t2 = new Test(100);
    S.o.pln(t1);      Test@1a3b2b
    S.o.pln(t2);      Test@2a4b2a
   }
}
```

Object → toString()
⇓
Object → hashCode()

0-15

0
1
2
|
|
9
a(10)
b(11)
c(12)
d(13)
e(14)
f(15)

$16 \overline{)100}$
6 - 4 →

64

10

Ex ②:-

```
Class Test
{
  int i;
  Test(int i)
  {
    this.i = i;
  }
  Public int hashCode()
  {
    return i;
  }
  p.s.v.m(——)
  {
  Test t1 = new Test(10);
  Test t2 = new Test(100);
  S.o.pln(t1);  Test@a
  S.o.pln(t2);  Test@64
  }
}
```

Object → toString()
⇓
Test → hashCode()

$16 \overline{)100}$
~~64~~ 4 ↑
64

$\boxed{10 \Rightarrow a}$

In hashCode

ex3:-

```
Class Test
{
   int i;

  Test (int i)
  {
   this.i = i;
  }

 Public int hashCode()
{
 return i;
}
 Public String toString()
 {
   return i+" ";
 }
 p.s.v.m(______)
 {
   Test t1 = new Test(10);
   Test t2 = new Test(100);

    S.o.pln(t1);        10
    S.o.pln(t2);       100

  }
}
```

Test → toString()

Note:-

→ if we are giving opportunity to object class toString() method than it will call internally hashCode() method.

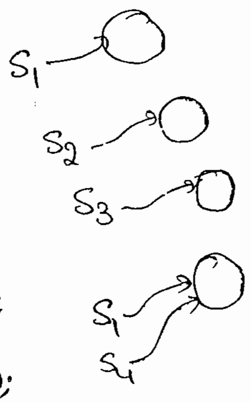
→ if we are giving opportunity to our class toString() method than it may not call hashCode() method.

③ equals() method :-

→ we can use equals() method to check equality of two objects

$\boxed{\text{public boolean equals(Object o)}}$

Ex:.
```
Class Student
{
 String name;
  int rollno;
  Student (String name , int rollno)
  {
    this.name = name;
    this.rollno = rollno;
  }
  P. S. v. m (————)
  {
   Student S1 = new Student ("durga", 101);
   Student S2 = new Student (" pavan", 102);
   Student S3 = new Student ('durga', 101);
   Student S4 = S1;
   S.o.pln (S1.equals(S2));    false
   S.o.pln (S1.equals(S3));    false
}} S.o.pln (S1.equals(S4));    true
```



→ In the above Case Object class .equals() method will be executed which is always ment for reference Comparision (address Comparision).

→ i.e, if two references pointing to the Same object Then only .equals() method returns true. This behaviour is Exactly Same as == operator.

→ If we want to perform Content Comparision instead of reference Comparision we have to Override .equals() method in our Class.

→ when ever we are overriding .equals() method we have to Consider The following things,

(1) What is the meaning of equality

(2) In the Case of diff. Type of Objects (Hetrogeneous) equals method should return false but not ClassCastException.

(3) If we are passing Null arguement our .equals method should returns false but not a NullpointerException.

→ The following is the valid way of overriding equals() method in Student class.

ex:
```
        public boolean equals(object o)
        {
          try
          {
            String name1 = this.name;
            int rollno1 = this.rollno;
            Student s2 = (Student) o;
            Student name2 = s2.name;
            int rollno2 = s2.rollno;
```

```
if( name1.equals(name2) && rollno1 == rollno2)
 {
   return true;
 }
 else
 {
   return false;
  }
}
Catch (CCE e)
 {
   return false;
 }
Catch(NPE e)
  {
   return false;
  }

Student s1 = new Student ("durga" 101);
Student s2 = new Student (" pavan" ,102);
Student s3 = new Student ( "durga", 101);

Student s4 =s1;

 S.o.pln (s1.equals( s2));        false
 S.o.pln (s1.equals(s3));        True
 S.o.pln (s1. equals (s4));       True
 S.o.pln (s1.equals(" durga")); false
 S.o.pln (s1.equals(null));       false
```

Short way of writting equals() method :-

```
public boolean equals(Object o)
{
    try
    {
     Student s2 = (Student)o;

     if(name.equals(s2.name) && rollno == s2.rollno)
         return ~~true~~;
      else
         return false;

    Catch(CCE e)
    {
     return false;
    }
    Catch(CCE e)
    {
     return false;
    }
  }
}
```

Relationship b/w == operator & .equals() method :-

→ if $r_1 == r_2$ is True, then r_1.equals(r_2) is always True.

→ if $r_1 == r_2$ is false, then we can't expect about r_1.equals(r_2) Exactly it may returns True or false.

→ if r_1.equals(r_2) returns True, we can't conclude anything about $r_1 == r_2$. it may returns either True or false.

→ if r_1.equals(r_2) is false, then $r_1 == r_2$ is always false.

difference b/w == operator & .equals() method :-

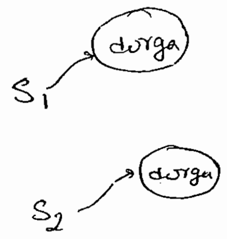
== operator	.equals()
① It is an operator applicable for both primitives & object references	① It is a method applicable only for object references but not for primitives.
② In the case of object references == operator is always meant for reference comparision. i.e, if two references pointing to the same object then only ~~=~~== operator returns T	② By default .equals() method present in object class is also meant for reference comparision only.
③ we can't override == operator for content comparision	③ we can override .equals() method for content comparision.
④ In the case of Hetrogeneous type objects ~~equal~~ == operator ~~Sets~~ causes compiletime error saying incomparable types	④ In the case of Heterogenous objects .equals() method simply return false & we won't get any compiletime or runtime error
⑤ for any object reference r, r == null is always false.	⑤ for any object reference r, r.equals(null) is always false.

Note:-

Q) what is the difference b/w Double equal operator (==) & .equals()

→ "==" Operator is always meant for reference Comparision, where as .equals() method meant for Content Comparision.

ex:-
```
String s1 = new String("durga");
String s2 = new String("durga");
S.o.pln (s1 == s2);   false
S.o.pln (s1.equals(s2));   true
```



→ In String, ~~All wrapper~~ classes .equals() is overridden for Content Comparision.

→ In StringBuffer Class .equals() is not overridden for Content Comparision hence object class .equals() got executed which is meant for reference Comparison.

→ In wrapper class .equals() is overridden for Content Comparision

Contract b/w .equals() & hashCode():-

1. If two objects are equal by .equals() Compulsary thire hashCodes must be Same.

2. If two objects are not equal by .equals() then there are no restructions on hashCode(), they Can be Same or different.

3. If hashCodes of 2 objects are equal, then we Can't Conclude above .equals(), It may returns True or false.

4. If hashcodes of 2 objects are not equals then we can always conclude .equals() returns false.

Conclusion :-

→ To Satisfy the above Contract b/w .equals() and hashcode(), whenever we are overriding .equals() compulsary we should override hashcode().

→ If we are not overriding we won't get any compile time & run-time errors.

→ But it is not a good program practice.

Q) Consider the following .equals()

```
public boolean equals(Object obj)
{
    if ( ! (obj instanceof person))
    {
        return false;
    }
    person p = (person) obj;
    if (name.equals(p.name) && (age==p.age))
        return true;
    else    return false;
}
```

1) Which of the following hashcode() are said to be properly implemented.

✗ ①
```
public int hashCode()
{
    return 100;
}
```

```
✗ ② public int hashCode()
     {
       return age + (int)height;
     }
✓ ③ public int hashCode()
    {
      return name.hashCode() + age;
    }
✗ ④ public int hashCode()
    {
      return (int)height;
    }
  ⑤ public int hashCode()
     {
       return age + name.length();
     }
```

Note:-
To maintain a contract b/w .equals() and hashCode(), what ever the parameters we are using while over riding .equals() we have to use the same parameters while overriding hashCode() also.

Clone():-

→ The process of creating exactly duplicate objects is called cloning

→ The main objective of cloning is to maintain backup.

① we can get cloned object by using clone() of Objects class.

```
protected native object clone() throws CloneNotSupportException
```

```
Class Test implements Cloneable
{
    int i =10;
    int j =20;
  p.s.v.m (——) throws CloneNotSupportException
  {
    Test t1 = new Test();
    Test t2 = (Test) t1.clone();
      t2.i = 888;
      t2.j = 999;
    S.o.pln ( t1.i + "-----" + t1.j);
  }
}
  S.o.pln(t1.hashCode() == t2.hashCode()); // false
  S.o.pln ( t1 == t2); // false.
```

→ We Can Call clone() only on Cloneable Objects.

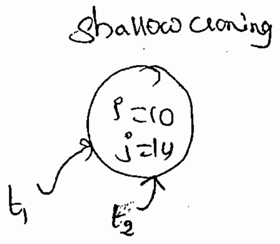
→ An object is Said to Clonable iff the Corresponding class implements Clonable interface. cloneable interface presently java.lang package & doesn't contain any methods. It is a marker interface.

Deep cloning & shallow Cloning:-

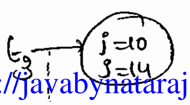
→ The process of Creating just duplicate reference variable but not duplicate object is Called Shallow cloning.

→ The process of Creating exactly duplicate independent objects is by bydefault Considered as deep cloning.

ex:-
```
Test t1 = new Test();
Test t2 = t1; // shallow cloning
Test t3 = (Test)t1.clone(); //Deep cloning
```



By default cloning means deep cloning.



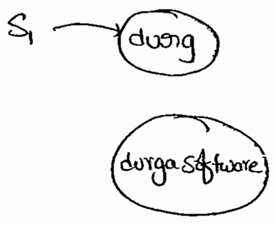
String class

2/05/11

Case(1):-

Immutable

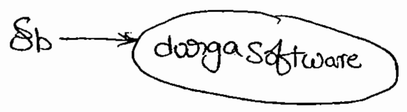
```
String s = new String("durga");
s.concate("software");
S.o.p(s);  durga
```



→ Once we created a String object we can't perform any changes in the Existing object. if we are trying to perform any changes with those changes a new object will be created this behaviour is nothing but "immutability of String object"

mutable

```
SB  s = new  SB("durga");
s.append ("software");
S.o.pln(s);  // durgasoftware
```



→ Once we created a StringBuffer Object we Can perform any changes in The existing object. This behaviour is nothing but "mutability of String-Buffer object".

getClass():

This method returns run-time class definition of an object

eg:-
```
Test ob = new Test();
S.o.pln (" class name: " + ob.getClass().getName());
```

Case(2) :-

```
String s1 = new String("durga");
String s2 = new String("durga");
S.o.pln(s1 == s2); false
S.o.pln(s1.equals(s2)); true
```

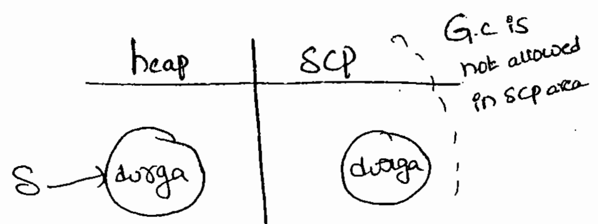
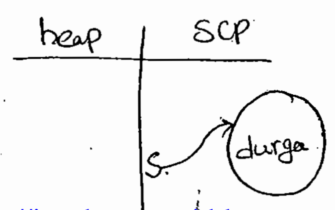
→ In String class .equals() method is overridden for content comparision. Hence .equals() method returns True if content is same eventhough objects are different.

```
StringBuffer sb1 = new StringBuffer("durga");
SB sb2 = new SB("durga");
S.o.pln(sb1 == sb2); false
S.o.pln(sb1.equals(sb2)); false
```

→ In StringBuffer class .equals() method is not overridden for content comparision. Hence object class .equals() method will be executed which is ment for reference comparision due to this .equals() method returns false eventhough content is same if objects are different

Case (3) :-

* What is the difference b/w following?

String s = new String("durga");	String s = "durga";
→ In this case two objects will be created one is in heap, & the other is in SCP. and 's' is always pointing to heap object  heap SCP G.C is not allowed in SCP area S durga durga	→ In this case only one object will be created in SCP and 's' is always pointing to that object 

Note:-

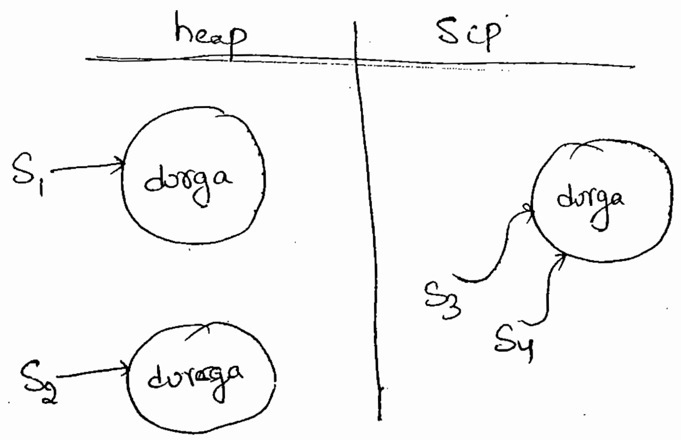
① G.c is not allowed to access in scp area hence eventhough object doesn't have any reference variable still it is not eligible for G.c, if it is present in scp area.

② All objects present on scp will be destroyed automatically at the time of Jvm shutdown.

③ Object creation in scp is always optional. first Jvm will check is any object already present in scp with required content or not. if it is already available then it will reuse existing object instead of creating new object. if it is not already available then only a new object will be created. Hence, there is no chance of two objects with the same content in scp. i.e, Duplicate objects are not allowed in scp.

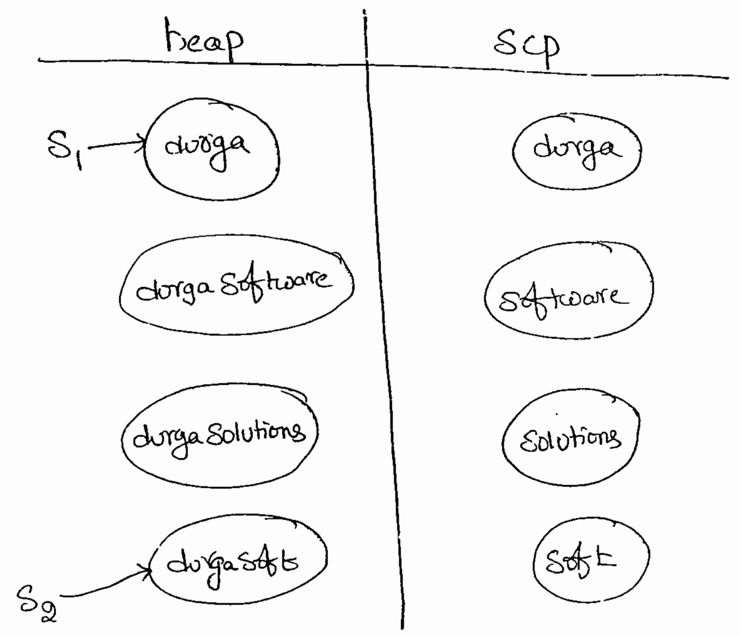
Ex②:-

```
String s1 = new String("durga");
String s2 = new String("durga");
String s3 = "durga";
String s4 = "durga";
```



Ex③ :-

```
String s1 = new String("durga");
        s1.concate("software");
        s1.concate("solutions");
String s2 = new s1.concate("soft");
```



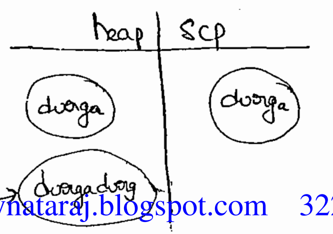
Note :-

→ for every String constant compulsory one object will be created in SCP area.

→ Because of some runtime operation if an object is required to created that object should be created only on heap but not in SCP

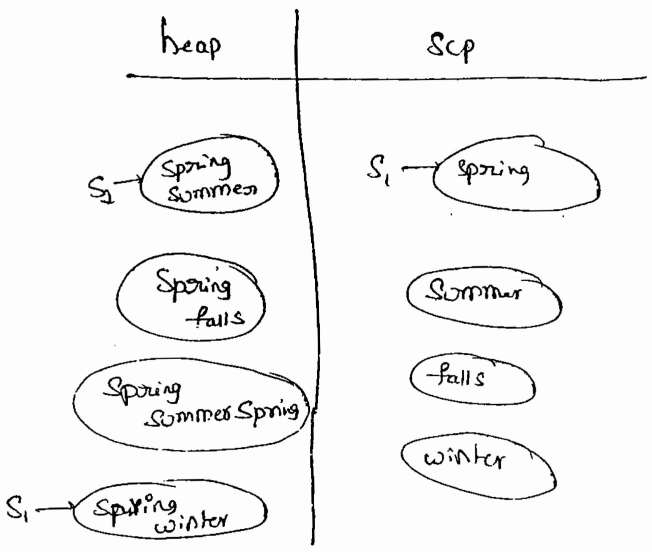
Ex④ :-

```
String s = "durga" + new String("durga");
```



Ex3:-

```
String S1 = "spring";
String S2 = S1 + "Summer";
S1.concat("falls");
S2.concat(S1);
S1+ = "winter";
S.o.pln(S1);
S.o.pln(S2);
```



Ex:- Note:-

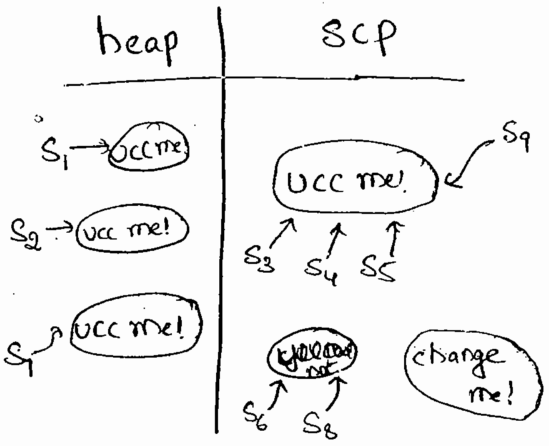
final String S = "raghu"; S is a Constant

String S = "raghu"; S is a normal variable.

Ex:-

String S1 = new String(" you can't

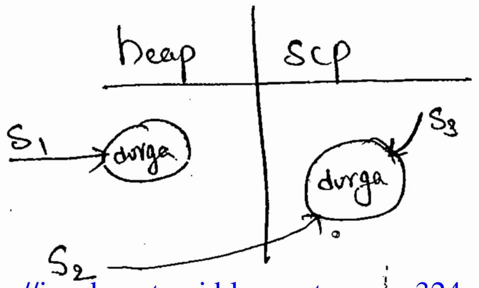
```
String s1 = new String("you cannot change me!");
String s2 = new String("you cannot change me!");
S.o.pln(s1 == s2);  false
String s3 = "you cannot change me!";
String s4 = "you cannot change me!";
S.o.pln(s1 == s4);  true
S.o.pln(s1 == s3);  false
String s5 = "you cannot " + "change me!";
S.o.pln(s3 == s5);  true
String s6 = "you cannot ";
String s7 = s6 + "change me!";
S.o.pln(s3 == s7);  false
final String s8 = "you cannot ";
String s9 = s8 + "change me!";
S.o.pln(s3 == s9)  ;  true
S.o.pln(s6 == s8);  true
```



Interning of String :-

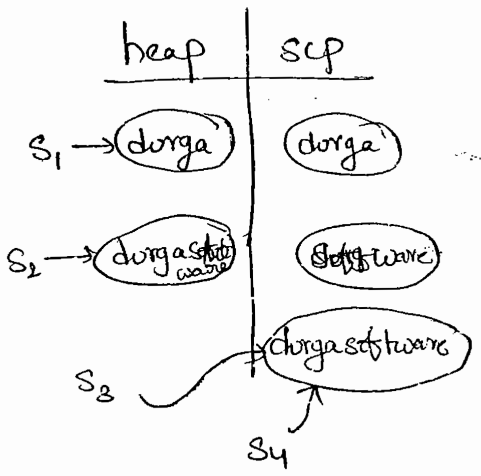
→ By using heap object reference if you want to get Corresponding Scp object reference then we should go for intern().

Ex:-
```
String s1 = new String("durga");
String s2 = s1.intern();
S.o.pln(s1 == s2);  false
String s3 = "durga";
S.o.pln(s2 == s3);  true
```



→ If the Corresponding object not available in scp, then intern() Creates that object & returns it.

Eg:-
```
String s1 = new String("durga");
String s2 = s1.concat("software");
String s3 = s2.intern();
String s4 = "durgasoftware";
S.o.pln(s3 == s4); true
```



Constructors of the String class:

```
① String s = new String();
② String s = new String(String constant);
③ String s = new String(StringBuffer sb);
④ String s = new String(char[] ch);
   Eg:- char[] ch = {'a', 'b', 'c', 'd'}
        String s=new String(ch);
        S.o.pln(s); abcd
⑤ String s = new String(byte[] b)
   eg:- byte[] b = {100, 101, 102, 103};
        String s = new String(b);
        S.o.pln(s); defg
```

important methodes of String class :-

① public char charAt (int index);

eg:-
```
String s = "durga";
S.o.pln(s.charAt(3]); g
S.o.pln (s.charAt[30]); R.E :- StringIndexOutOfBoundsException
```

② public String concat (String s);

eg:-
```
String s = "durga";
s = s.concat ("software");
//s = s+ "software";
// s += "software";
S.o.pln(s) ; durgasoftware
```

→ The overloaded + , += operators also ment for Concatination only

③ public boolean equals(Object obj) ment for Content Comparision where the Case is also important.

④ public boolean equalsIgnoreCase(String s) ment for Content Comparision where the Case is not important.

Ex:-
```
String s = "JAVA";
S.o.pln (s.equals ("Java")); false
S.o.pln(s.equalsIgnoreCase("java")); true
```

Note:- In General to perform Validation of User name we have to go for equalsIgnoreCase method where the Case is not important. where as to perform password Validation we have to [illegible] where the Case is important.

⑤ public String substring(int begin); returns the substring from begin index to End of the String.

⑥ public string substring(int begin, int end); returns the substring from begin index to End-1 index.

Ex:- String s = "abcdefg";
  S.o.pln(s.substring(3)) ; defg
  S.o.pln(s.substring(2,6)); cdef

⑦ public int length();

-g:- String s = "aabbb";
  S.o.pln (s.length); $\longrightarrow$ C.E: Can't find Symbol
    Symbol: variable length
    location: class java.lang.string
  ✓ S.o.pln(s.length()); 5

Note:-
length variable applicable for arrays where as Length() is applicable for string objects.

⑧ public String replace(char old, char new);
  eg:- String s = "aabbb";
    s.o.pln (s.replace('a','b')); bbbbb

⑨ public String toLowerCase();
⑩ Public String toUpperCase();

9/3/2011

⑪ public String trim() :-

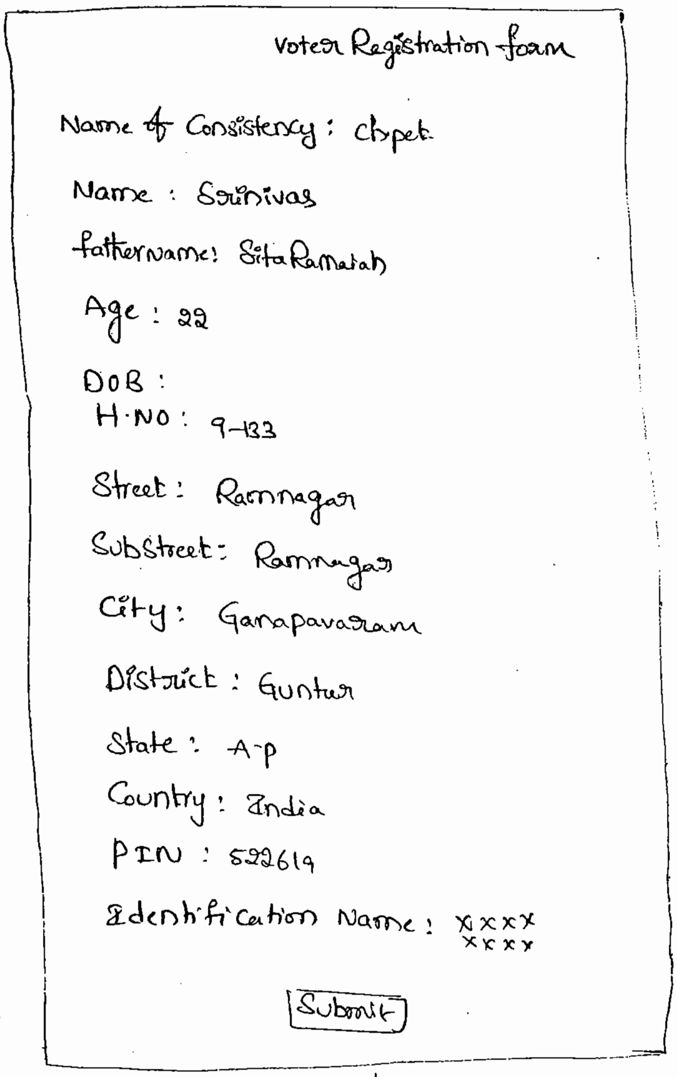
→ To remove the blank Spaces present at begining & End of the String But not blankspaces present at middle of the String.

⑫ public int indexOf(char ch) :-

→ It returns indexof first occurance of the Specified character

⑬ public int lastIndexOf(char ch) :-

** Importance of String Constant Pool (SCP) :-



Voter Registration form

Name of Consistency : chpet

Name : Srinivas

fatherName: SitaRamaiah

Age : 22

DOB :

H·NO : 9-133

Street : Ramnagar

SubStreet : Ramnagar

City : Ganapavaram

District : Guntur

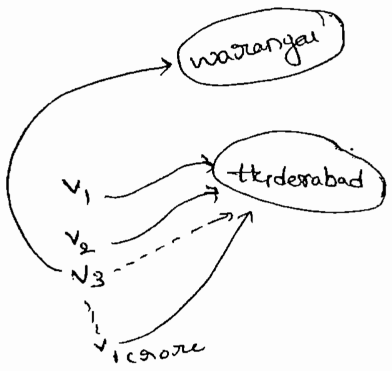
State : A·P

Country : India

PIN : 522619

Identification Name : xxxx xxxx

Submit



→ In our program if any String object required to use repetadely, it is not recommended to create a seperate object for every requirement. This approach reduces performance & memory utilization.

→ we can resolve this problem by creating only one object & share the same object with all required references.

→ This approach improves memory utilization & performance. we can acheive this by using String constant pool.

→ In scp, a single object will be shared for all required References. Hence the main advantages of scp are memory-utilization & performance will be improved.

→ But the problem in this approach is, As several references pointing to the same object by using one reference, if we are perform any change all remaining references will be impacted.

→ To resolve these SUN people declare String objects as immutable.

→ According to that once we created a String object we can't perform any change in the existing object. if we are trying to perform any change with

So, that there is no effect on remaining references

→ Hence, "the main disAdvantage of scp is we should compulsary maintain String objects as immutable".

Q) why Scp like Concept is defined only for String object But not for StringBuffer?

Ans) → In any Java program, the most Commonly used Object is String. Hence with respect to memory & performance Special arrangement is required. for this Scp Concept required.

→ But StringBuffer is not Commonly used object. Hence Special Concepts like Scp is not required.

Q) what are the Advantages of Scp?

A) → Instead of Creating a Seperate object for every requirement we Can Create only one object in Scp & we Can reuse the Same object for Every requirement. So that performance & memory utilization will be increased.

Q) what is the disAdvantage of Scp?

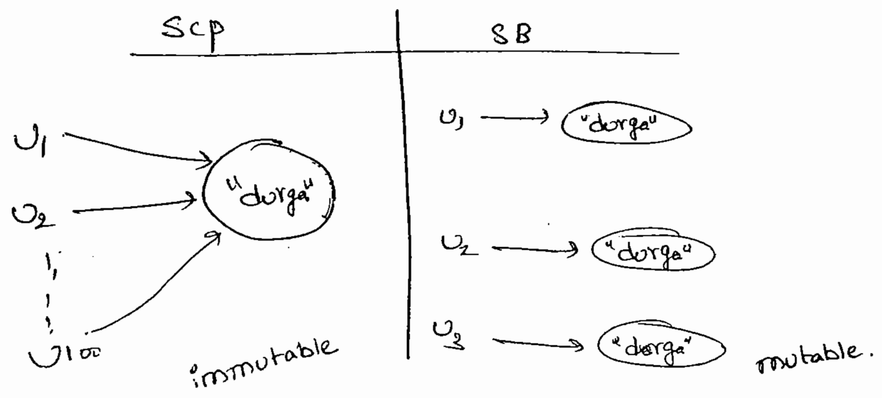
A) → Compulsary we should make String objects as immutable.

Q) why String objects are immutable where as StringBuffer objects are mutable?

A) → In the Case of String Several references Can Pointing to the Same object. By using one reference, if we are Performing any chang in the Existing object the remaining references will be impacted. to resolve this problem SUN people declared as String objects are immutable. According to this once we Created a String object we can't perform any changes in the existing object.

If we are trying to perform any changes, with those changes a new object is Created. i.e Scp is the only reason why the String objects are immutable.

→ But in case of StringBuffer for every requirement compulsary a Seperate object will be Created. Reusing the Same StringBuffer object, there is no chance. In one StringBuffer object if we are performing any change there is no impact of remaining references. Hence we can perform any changes in the StringBuffer object & StringBuffer objects are mutable.



10/03/11

Q) Is it possible to Create our own immutable class?

A) Yes,

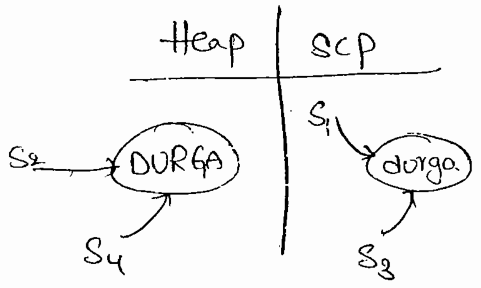
Note:

→ Once we Created a String object we Can't perform any Changes in the existing object. If we are trying to perform any change with those changes a new object will be Created on the Heap.

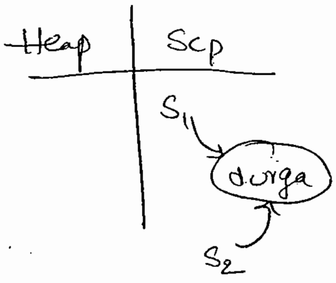
→ Because of our runtime method call if there is a change in Content then only new object will be Created.

→ If there is no change in Content Existing object only will be reused.

Ex①
```
String s1 = "durga";
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
String s4 = s2.toUpperCase();
S.o.pln(s1 == s2);   false
S.o.pln(s1 == s3);   True
S.o.pln(s2 == s4)    True
```



Ex②:-
```
String s1 = "durga";
String s2 = s1.toString();
S.o.pln(s1 == s2);   True
```



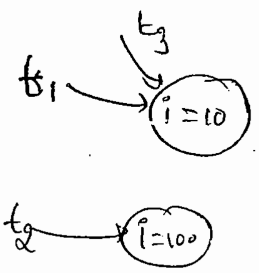
Creation of Our Own Immutable class :-

Sol: We Can Create Our own immutable Classes also.

→ Once we Created an object we Can't perform any change in the existing object. If we are trying perform any change with those changes a new object will be Created.

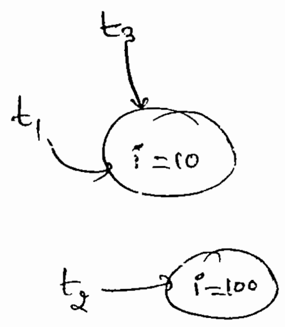
→ Because of our runtime method Call if there is no change in the Content then Existing object only will be returned.

Ex:-



Ex:-

```
final class Test
{
 private int i;
 Test (int i)
 {
 this.i = i;
 }
public Test modify (int i)
{
 if(this.i == i)
 return this;
 return (new Test (i));
 }
}
Test t1 = new Test (10);
Test t2 = new Test (100);
Test t3 = new Test (10);
S.o.pln (t1 == t2); false.
S.o.pln (t1 == t3); true
```



Q) In Java which objects are Immutable ?

A) (1) String objects &

(2) All wrapper objects are immutable

StringBuffer :-

→ If the Content will change frequently then it is never recommended to go for String. Because for every change Compulsary a new object will be created.

→ To handle this requirement Compulsary we should go for StringBuffer where all changes will be performed in existing object only instead of Creating new object.

Constructors :-

→① StringBuffer Sb = new StringBuffer( );

→ Creates an Empty StringBuffer object with default initial Capacity 16

→ Once StringBuffer reaches its max. capacity a new SB object will be Created with,

New Capacity = (Current Capacity + 1) * 2

Ex:-

```
StringBuffer sb = new StringBuffer();
S.o.pln (sb.capacity()); //16
Sb. append ("abcdefghijklmnop");
S.o.pln (Sb.capacity()); 16
Sb.append ("q");
S.o.pln (Sb.capacity()); 34.
```

(2) StringBuffer sb = new StringBuffer(int initialCapacity);

→ Creates an Empty SB object with specified initialCapacity

(3) StringBuffer sb = new StringBuffer(String s);

→ Creates an equivalent SB object for the given String with,

Capacity = 16 + s.length();

Important Methods of StringBuffer class:-

(1) public int length()

(2) public int Capacity()

(3) public char charAt(int index);

ex:-
```
StringBuffer sb = new StringBuffer("durga");
  S.o.pln (sb.charAt(3)); g
  S.o.pln (sb.charAt(30));  } R.E:- StringIndexOutOfBound
  S.o.pln (sb.charAt(5));   }       Exception
```

(4) public void setCharAt(int index, char ch);

→ To replace the character Locating at specified index with the provided Character.

(5) public StringBuffer append(String s)
```
                        append (int    i)     |
                        append (boolean b)    |  overloaded methods
                        (double  d)           |
                        (Object  o)           |
```

ex:- StringBuffer Sb = new StringBuffer();

```
Sb.append("Pi value is");
Sb.append(3.14);
Sb.append("It is exactly");
Sb.append(true);
S.o.pln(sb);
```

⑥ public StringBuffer insert(int index , String s) ;
(int index , ~~String~~ int i) ;
(" boolean b);
(" double d);
:
:

ex:- StringBuffer sb = new StringBuffer("durga");

```
Sb.insert(3 ,"sainu");
S.o.pln(sb) ; dursainuga .
```

⑦ public StringBuffer delete(int begin , int end);

→ To delete the characters Present at begin index to End-1 index

⑧ public StringBuffer deleteCharAt(int index);

→ To delete the character Locating at Specified index.

⑨ public StringBuffer reverse( ) :

eg:- SB sb = new SB("durga");

```
S.o.pln(Sb.reverse()); agrud .
```

⑩ public void setLength(int Length) ;

⑩ ** public void setLength(int Length);

eg:-
```
StringBuffer sb = new StringBuffer("durga123456");
     sb.setLength(8);
     S.o.pln(sb);  durga123
```

** ⑪ public void ensureCapacity(int Capacity);

→ ~~To get~~ the Capacity based on our requirement.

eg:-
```
StringBuffer sb = new StringBuffer();
  System.out.println(sb.capacity());     16
  Sb.ensureCapacity(2000);
  System.out.println(Sb.Capacity());     2000
```

⑫ public void trimToSize()

→ To release extra allocated free memory. after Calling this method, Length & Capacity will be equal.

eg:-
```
StringBuffer sb = new StringBuffer();
  sb.ensureCapacity(2000);
  Sb.append("durga");
  Sb.trimToSize();
  S.o.pln(Sb.Capacity());   5
```

StringBuilder :-

→ Every method present in StringBuffer is Synchronized, Hence at a time only one Thread is allowed to access StringBuffer object. It Increases waiting time of the Threads & effects performance of the System.

→ To resolve this problem SUN people introduced StringBuilder in 1.5 version.

→ StringBuilder is exactly Same as StringBuffer (including methods & Constructors) except the following differences.**

(*)

StringBuffer	StringBuilder
① Every method is Synchronized	① No method is Synchronized.
② SB object is thread Safe. Because SB object can be accessed by only one Thread at time.	② StringBuilder is not Thread Safe Because it can be accessed by multiple-Threads Simultaneously.
③ relatively performance is Low	③ relatively Performance is High.
④ Introduced in 1.0 Version	⑤ Introduced in 1.5 Version

*String Vs StringBuffer Vs StringBuilder :-

→ If the Content ~~only~~ will not change frequently Then we should go for String

→ If Content will change frequently & ThreadSafety is required. then we should go for StringBuffer.

→ If Content will change frequently & ThreadSafety is not required. Then we should go for StringBuilder.

Method chaining :-

→ for most of the methods in String, StringBuffer & StringBuilder The returntype is Same type only. Hence after applying a method on The result we Can Call another method which forms methodchaining

```
sb.m1().m2().m3().m4().m5().....;
```

→ In method Chaining all methods will be executed from Left to Right.

Ex :-

```
StringBuffer sb = new StringBuffer();
sb.append("durga").insert(2,"xyz").reverse().~~delet~~
                    delete(2,7).append("solutions");
S.o.pln(sb); //agdsolutions
```

final vs immutable :-

→ If a reference variable declareded as the final then we can't reasign That reference variable to some other object.

Ex:-

```
final StringBuffer sb = new StringBuffer("durga");
      sb = new StringBuffer("software");
```

C.E:- Can't assign a value to final variable sb.

→ declaring a reference variable as final we wont get any immutability nature, in the corresponding object we can perform any type of change eventhrough reference variable declared as final.

Ex:-

```
final StringBuffer sb = new StringBuffer("durga");
      sb.append("software");
      S.o.pln(sb); durgasoftware
```

→ Hence final variable & Immutability both Concepts are different.

* Wrapper classes :-

→ The main objectives of wrapper classes are

(i) To wrap primitives into object form, So that we can handle primitives Just like objects.

(ii) To define Several utility methods for the primitives.

Constructors of wrapper classes (or) Creation of wrapper objects :-

→ Allmost All wrapper classes Contains two Constructors, one Can take Corresponding premitive as arguement & the Other can take String as arguement.

Ex!-

```
✓ | Integer   I = new  Integer(10);
  | Integer   I = new  Integer("10");

✓ | Double    D = new  Double (10.5);
  | Double    D = new  Double ("10.5");
```

→ If the String is not properly formatted then we will get R.E Saying NumberFormattException.

Ex!-

```
Integer  I = new  Integer("ten");  R.E!- NFE
```

→ Float class Contains 3 Constructors one Can take float premitive, and the other Can take String & 3rd one Can take double arguement.

Ex: 1) Float F = new Float (10.5F); ✓

2) Float F = new Float ("10.5f"); ✓

3) Float F = new Float (10.5); ✓ ⟶ double.

* Character class Contains only one Constructor which Can take Char premitive as arrguement.

Ex:- 1) Character ch = new Character('a'); ✓

2) Character ch = new Character("a"); ✗

* Boolean class Contains two Constructors one Can take Boolean premitive as the arrguement & other Can take String as arrguement.

→ If we are passing boolean premitive as arrguement the only allowed values are true, false. by mistake if we are providing any other we will get Compiletime Error.

Ex:-

✓ Boolean B = new Boolean (true);

✗ Boolean B = new Boolean (True);

→ If we are passing String arrguement to the Boolean Constructor then the Case is not important & Content also not important.

→ If the Content Case insensitive String ~~a~~ true}, otherwise it is treted as false.

Ex:- (1) Boolean b = new Boolean ("true"); ✓ true

(2) Boolean b = new Boolean ("True"); ✓ true

(3) Boolean b = new Boolean ("TRUE"); ✓ true

(4) Boolean b = new Boolean ("durga"); ✓ false

(5) Boolean b = new Boolean ("[illegible]"); ✓ false

Wrapper classes	Corresponding Constructor arguement
Byte	byte or String
Short	short or String
Integer	int or String
Long	long or String
* Float	float or String or double
Double	double or String
* Character	char
* Boolean	boolean or String

Q): which one is True & false

(1) Boolean b1 = new Boolean("yes");

(2) Boolean b2 = new Boolean("no");

```
S.o.pln(b1.equals(b2)); → true
S.o.pln(b1==b2); → false
S.o.pln(b1); false
S.o.pln(b2); false.
```

Note:-

→ In Every wrapper Class toString() is overridden to return it's Content.

→ In Every wrapper Class .equals() is overridden for Content Comparision.

Utility Methods :-

There are 4 methods

(i) ValueOf()

(ii) xxxValue()

(iii) parseXxx()

(iv) toString()

(i) valueOf() :-

→ We Can use valueOf() methods for Creating wrapper object as alternative to Constructor.

Form 1:-

→ Every wrapper class Except Character Class Contains a Static valueOf() method for Converting for Converting String to the wrapper Object.

$$\boxed{\text{Public Static wrapper valueOf(String s)}}$$

Eg:- Integer I₁ = Integer.ValueOf("10"); ✓

Boolean b₁ = Boolean.ValueOf("true"); ✓

Double D = Double.valueOf("10.5"); ✓

form(2):-

→ Every Integral type wrapper class (Byte, Short, Integer, Long) Contains the following valueOf() method to Convert Specified Radix String form to Corresponding Wrapper object.

```
public static wrapper valueOf(String s, int radix);
```

2 to 36

Ex:-

```
Integer I1 = Integer.valueOf("1010", 2);
    S.o.pln(I1); 10
Integer I2 = Integer.valueOf("1111", 2);
    S.o.pln(I2); 15
```

base-10 : 0-9
base-11 : 0-9,a
!
base-16 : 0-9, a-f
base-17 : 0-9, a-g
!
base-36 : 0-9, a-z.
10 + 26
= 36

form (3):-

→ Every wrapper class including Character class Contains the following valueOf() to Convert primitive to Corresponding wrapper Object

```
public static wrapper valueOf(primitive p);
```

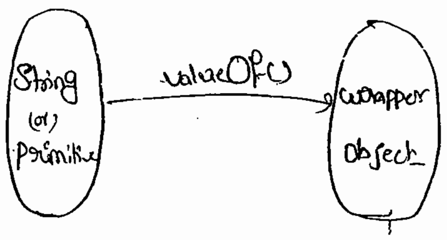
Eg:-

1) `Integer I = Integer.valueOf(10);` ✓

2) `Character ch = Character.valueOf('a');` ✓

3) `Boolean B = Boolean.valueOf(true);` ✓

Note:



15/03/11

(ii) xxxValue() :-

→ we can use xxxValue() methods to convert wrapper object to primitives.

→ Every number type wrapper class contains the following six(6) xxxValue() methods.

→ The methods are

```
public byte byteValue();
public int intValue();
public short shortValue();
public long longValue();
public float floatValue();
public double doubleValue();
```

eg:-

```
(1)  Double D = new Double(130.456);

     S.o.pln(D.byteValue());     -126
     S.o.pln(D.shortValue());    130
     S.o.pln(D.intValue());      130
     S.o.pln(D.longValue());     130
     S.o.pln(D.floatValue());    130.0
     S.o.pln(D.doubleValue());   130.0
```

charValue():-

→ Character class contains char Value method to convert Character Object to the ~~Exea~~ char primitive.

* public char charValue();

eg:- Character ch = new Character('@');
char ch1 = ch.charValue();
S.o.pln(ch1); '@'

booleanValue():

→ Boolean class contains booleanValue() to find boolean primitive for the given boolean Object.

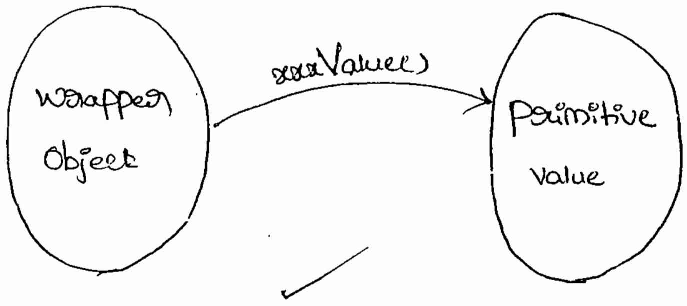
```
public boolean booleanValue();
```

eg:- Boolean B = Boolean.valueOf("durga");
boolean b = B.booleanValue();
S.o.pln(b); false.

6x6=36
+1
+1
= 38

Note:-

→ In total 38 (=6x6+1+1) xxxValue() are vailable.



(iii) parseXxx() :-

→ We can use parseXxx() to convert String to corresponding primitive.

form1 :-

→ Every wrapper class except Char class contains ~~the~~ the following parseXxx() to convert String to corresponding primitive.

$$\boxed{\text{public static primitive parseXxx(String s);}}$$

```
Eg:- ✓ int i = Integer.parseInt("10");
     ✓ double d = Double.parseDouble("10.5");
     ✓ long l = Long.parseLong("10l");
     ✓ Boolean b = Boolean.parseBoolean("durga");   o/p- false
```

form2:-

→ Every Integral type wrapper class contains the following parseXxx() to convert specified radix String to corresponding primitive.

```
Eg:-     public static primitive parseXxx(String s, int radix);
                                                        2 to 36.
Eg!-  int i = Integer.parseInt("1111", 2);
      S.o.pln(i); 15
```

Note:-

$$\begin{tikzcd} \text{String} \arrow[r, "\text{parseXxx()}"] & \text{Primitive} \end{tikzcd}$$

(iv) toString() :-

→ we can use toString() to Convert Wrapper Object or primitive to String.

form(1) :-

→ Every wrapper class Contains the following toString() , ~~to Co~~ to Convert Wrapper Object to String type.

```
public String toString();
```

→ It is the Overriding Version of Object class toString()

Eg:- ① Integer I = new Integer(10);

S.o.pln(I.toString()); 10 ✓

form2 :-

→ Every wrapper class Contains a Static toString(), to Convert primitive to String form.

```
public static String toString(primitive p);
```

✓ String s = Integer.toString(10);

✓ String s = Boolean.toString(true);

form(3) :-

→ Integer & Long classes Contains toString() to Convert primitive to Specified radix String form.

$\boxed{\text{public static String toString(primitive p, int radix);}}$

2 to 36

Eg:

```
String s = Integer.toString(15, 2);
S.o.pln(s); 1111
```

form 4:-

→ Integer & Long classes Contains the following toXxxString()

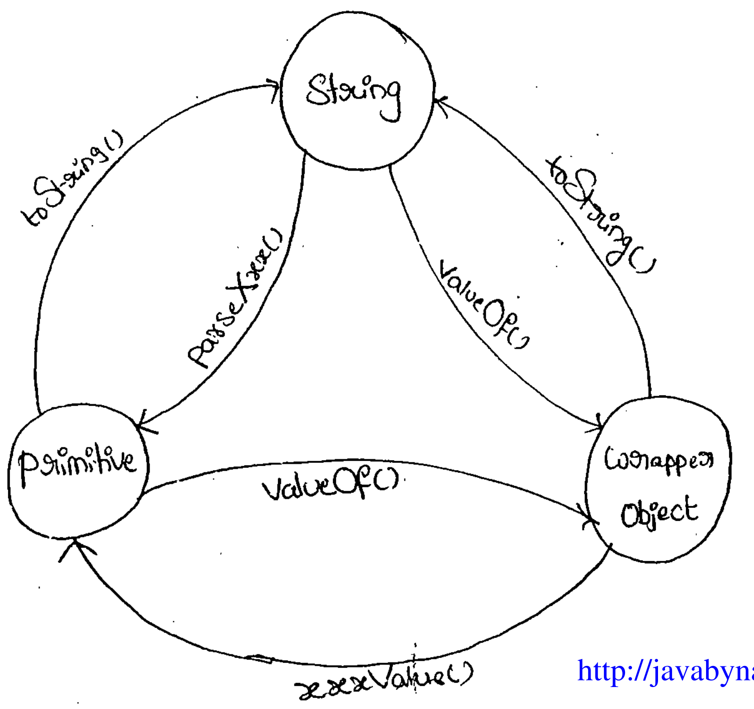
1. public static String toBinaryString(primitive p);
2. public static String toOctalString(primitive p);
3. public static String toHexString(primitive p);

Ex:-

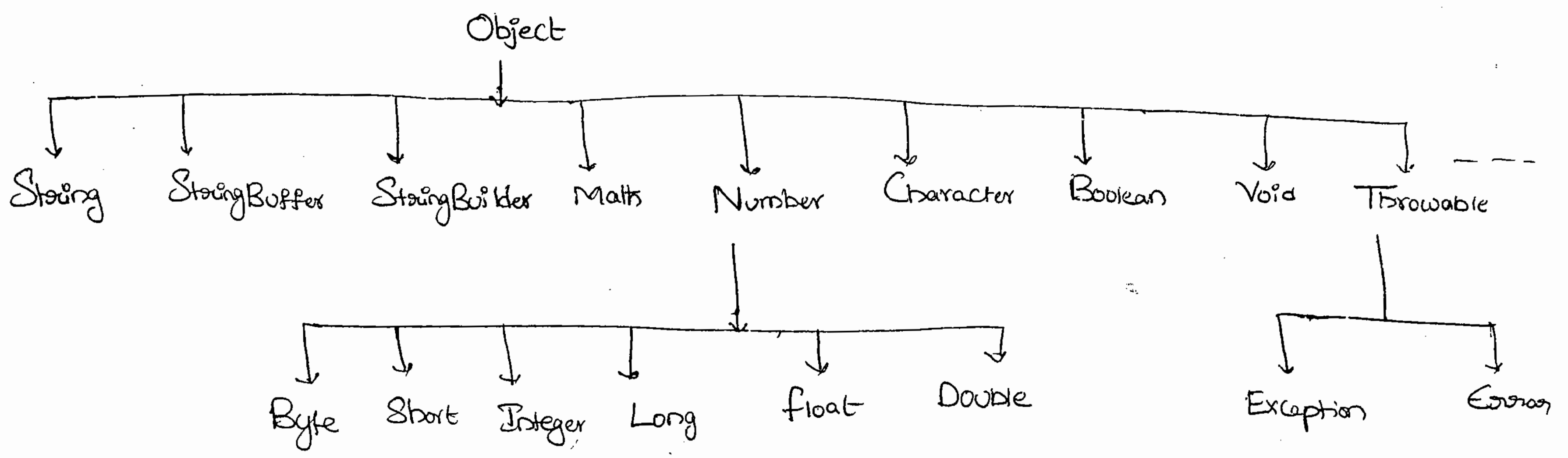
```
String s = Integer.toHexString(123)
S.o.pln(s); "7b"
```

16 | 123
   7 - "b"

Dancing b/w String, Wrapper Object & primitive Value:-



Partial hierarchy of java.lang package:-



* String, StringBuffer, StringBuilder, All Wrapper Classes are final.

* The wrapper Classes which are not child Classes of Number are, Character & Boolean.

* The wrapper classes which are not direct child classes of Object are Byte, Short, Integer, Long, Float, Double

* Some-times we Can Consider Void also as Wrapper Classes

* In addition to String object all wrapper objects are Immutable.

16-3-11

Autoboxing & Autounboxing :- (1.5v)

→ until 1.4 version we can't provide primitive value in the place of wrapper objects & wrapper objects in the place of primitive. All the required conversions ~~P~~ should be performed Explicitly by the programmer.

Ex:-

①
```
ArrayList l = new ArrayList();
    l.add(10);  X C.E!.

Integer I = new Integer(10);
    l.add(I); ✓
```

②
```
Boolean B = new Boolean(true);
  if(B) ──────────→ C.E!-
  {                  Incompatible types
   S.o.pln("Hello");  found : Boolean
  }                   required : boolean

boolean b = B.boolenValue();
  if(b) ✓
  {
   S.o.pln("Hello"); ✓
  }
```

→ But from 1.5 version on wards in the place of wrapper objects we can provide primitive value & in the place of primitive value we can provide wrapper objects. All the required conversions will be performed automatically by the compiler. these automatic

Conversions are Called Autoboxing & Auto unboxing.

Autoboxing:-

→ Automatic Conversion of primitive value to the wrapper Object by Compiler is Called "Autoboxing".

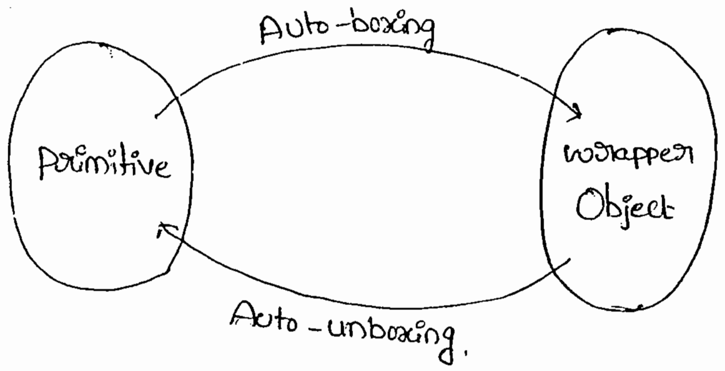
Ex:- ✓ ⅰ) `Integer I = 10;` [Compiler Converts int to Integer automatically by Autoboxing].

Auto-unboxing:-

→ Automatic Conversion of wrapper Object to the primitive type by Compiler is Called "Auto-unboxing".

Ex:- ✓ ⅰ) `int i = new Integer(10);` [Compiler Converts Integer to int automatically by Auto-unboxing].

Note:-



Ex:- ① `Integer I = 10;`

↳ after Compilation this Line will become

```
Integer I = Integer.valueOf(10);
```

i.e, Autoboxing Concept internally implimented by using ValueOf()

Ex②:-

```
Integer I = new Integer(10);
   int i = I;
```

→ After Compilation this Line will become

```
   int i = I.intValue();
```

i.e, AUtounboxing Concept internally implemented by using ~~xxx~~Value().

Exam purpose:-

ex(1):-

```
Class Test
{
  Static Integer I =10; ——→ ① A.B
  p.s.v.m(String[] args)
  {
    int i = I; ——→ ②. A.U.B
    m1(i);
  }               →③ A.B
  p.s.v.m1(Integer I)
  {
    int k = I; ——→ ④ A.U.B
    S.o.pln(k);  10
  }
}
```

Note:-

→ Because of Autoboxing & Auto-unboxing, from 1.5Version onwards There is no diff. b/w primitive Value & wrapper Object. we Can use interchangeably.

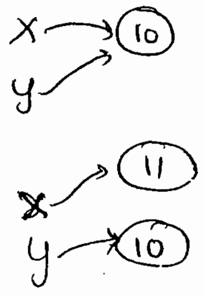
Ex2:-

```
class Test
{
 Static Integer I=0;
 p.s.v.m(String[] args)
 {
   int i = I;
   S.o.pln(i); //0
 }
}

     int i= I.intValue();
```

```
class Test
{
  Static Integer I;
  p.s.v.m(String[] args)
  {
    int i= I;  → R.E:- NPE
    S.o.pln(i);
  }
}

      int i= I.intValue()
             ↓
            null
```

Ex3:-

```
Integer x = 10;
Integer y = x;
 x++;
✓ S.o.pln(x); 11
✓ S.o.pln(y); 10
✓ S.o.pln(x==y); false
```

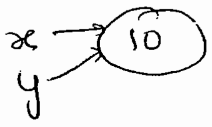


Note:-
because if we want to changes after creating an object, then that new changed object is created with the same reference name.

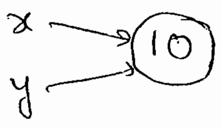
Ex4:-

```
① Integer x = new Integer(10);
  Integer y = new Integer(10);
  S.o.pln(x ==y); false ✓

② Integer x = new Integer(10);
  Integer y =10;
  S.o.pln(x = =y); false ✓
```

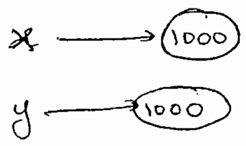
③
```
Integer x =10;
Integer y =10;
S.o.pln(x == y) ; true ✓
```



④
```
Integer x=100;
Integer y=100;
S.o.pln(x==y); true ✓
```



⑤
```
Integer x = 1000;
Integer y =1000;
S.o.pln(x==y); false ✓
```



Conclusion :-

→ By AutoBoxing if an object is required to create compiler won't create that object immidiately. first check is any object already created

→ If it is already created then it will reuse existing object. instead of creating new one.

→ If it is not already there, then only a new object will be created.

→ But this rule is applicable only in the following cases.

① Byte → Always
② Short → -128 to 127
③ Integer → -128 to 127
④ Long → -128 to 127
⑤ Character → 0 to 127
⑥ Boolean → Always

→ Except the above range in all other cases compulsory a new object will be created.

Ex:-

① Integer I_1 =127;
Integer I_2 = 127;
S.o.pln (I_1==I_2); true

② Integer I_1 =128;
Integer I_2 =128;
S.o.pln (I_1==I_2); false

③ Float f_1 =10.0f;
Float f_2 =10.0f;
S.o.pln(f_1==f_2); false

④ Boolean b_1 = true;
Boolean b_2 =true;
S.o.pln(b_1==b_2); true.

① Byte	→	Always
② Short	→	-128 to 127
③ Integer	→	-128 to 127
④ Long	→	-128 to 127
⑤ Character	→	0 to 127
⑥ Boolean	→	Always

→ Overloading w.r.t Auto-boxing, widening & Var-Arg methods:-

case(1):-

Widening Vs Auto-boxing:-

Ex:
```
Class Test
{
 p.s.v.m1(long l)
  {
   S.opln ("widening");
  }
 p.s.v.m1 (Integer I)
  {
   S.o.pln("Autoboxing");
  }
p.s.v.m(String[] args)
{
   int x=10;
     m1(x);    o/p:-
               widening
  }
}
```

1.0V 2010

→ <u>widening</u> dominates <u>Auto-boxing</u>

<u>Case(2)</u>:-

→ <u>Widening</u> Vs <u>Var-arg()</u>:-

```
Ex:-  Class Test
       {
        p.s.v.m1(long l)
         {
          S.o.pln("widening");
         }
        p.s.v.m1(int... i)
         {
          S.o.pln("Var-arg");
         }
       p.s.v.main(String[] args)
        {
         int x=10;
         m1(x);      o/p:- widening
        }
     }
```

→ widening dominates Var-arg().

Case 3:-

→ Auto-boxing Vs Var-arg :-

Ex:-
```
Class Test
{
 p.s.v.m1(Integer I)
 {
  S.o.pln("Autoboxing");
 }
 p.s.v.m1(int... i)
 {
  S.o.pln("var-arg");
 }
 p.s.v.m(String[] args)
 {
   int x=10;
    m1(x);        o/p:- Autoboxing.
 }
}
```

→ In General Var-arg() will get least priority, If no other method matched then only Var-arg() will be Executed.

→ while resolveing over loaded methods Compiler will always keeps the percidence in the following order.

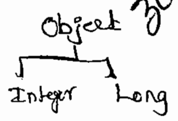
(i) Widening

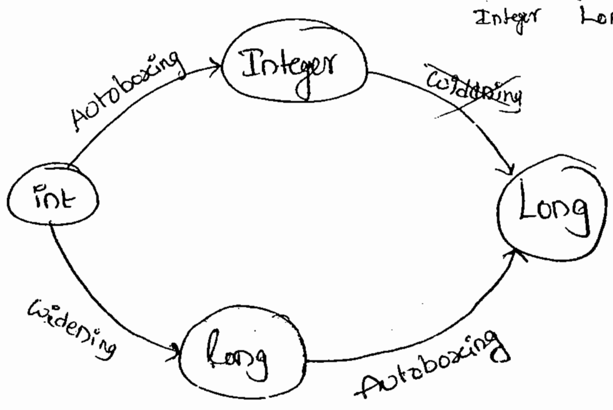
(ii) Auto-boxing

(iii) Var-arg().

Case 4 :-

```
Class Test
{
 p.s.v.m1(Long l)
 {
  S.o.pln("Long");
 }
 p.s.v.main(String[] args)
 {
  int x=10;
  m1(x);
 }
}
```

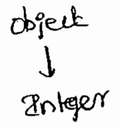


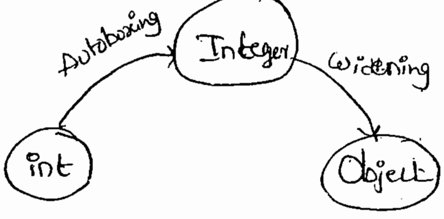


C.E:-
m1(java.lang.Long) in Test cannot be applied to (int)

→ widening followed by Auto-boxing is not allowed in java. where as Autoboxing followed by widening is allowed.

Ex!-
```
Class Test
{
  p.s.void m1(Object o)
  {
    S.pln("Object");
  }
  p.s.void.main(String[] args)
  {
    int x=10;
     m1(x); o/p Object ✓
  }
}
```





Q) Which of the following declarations are Valid.

✓ ① long l = 10;

✗ ② Long l = 10;

✓ ③ Object o = 10;

✓ ④ double d = 10;

✗ ⑤ Double d = 10;

✓ ⑥ Number n = 10;