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 negleetance at Second 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 failure 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 eligible for G.C :-

→ Eventhough programmer is not responsible to destroy useless objects, it is always a good programming 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.

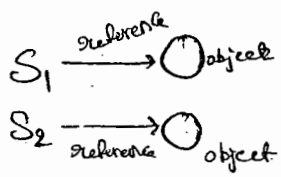
→ The following are various possible ways to make an object eligible for G.C.

(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.

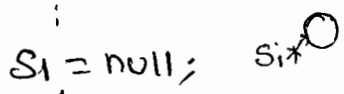
Ex!(1)

```
Student s1 = new Student();
Student s2 = new Student();
```

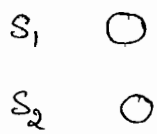


No objects eligible for G.C →

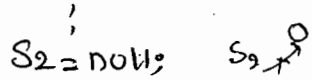
```
s1 = null;
```



one object eligible for G.C →




```
s2 = null;
```



two objects eligible for G.C →

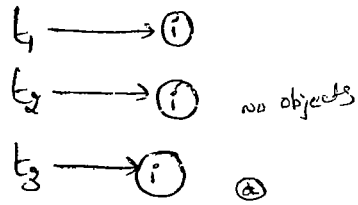
4) Island of isolation :-

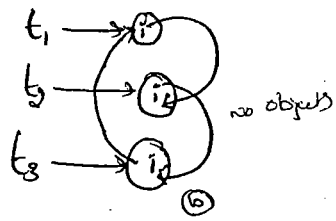
Ex:-

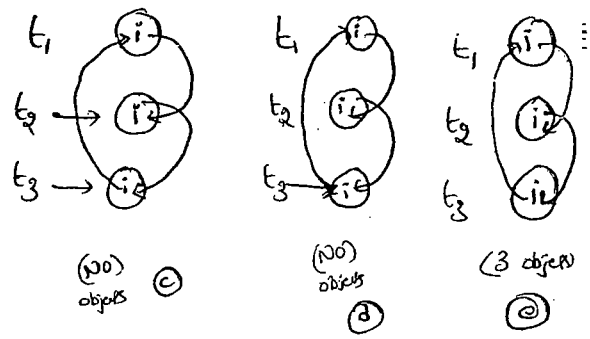
```
Class Test
{
    Test i;
    P.S.V. main(String[] args)
    {
    Test t1 = new Test();
    Test t2 = new Test();
    Test t3 = new Test();
    !
    t1.i = t2;
    t2.i = t3;
    t3.i = t1;
    :
    t1 = null;
    :
    t2 = null;
    :
    t3 = null;
    }
}
```

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 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 making 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 destroyed.

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

→ The following are various ways for 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 a the Heap (Heapsize)

(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());
  }
}
```

d → ○

d ○

Q) Which of the following is the properway of requested JVM to run g.c()?

1) System.gc(); (System is Static method)

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

X 3) (new Runtime()).gc(); (gc is applicable only Static method)

✓ 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.

Ex!.

```
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.

→ Just 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. Bcoz

→ 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

Eg:-
```
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.println("finalize method called");
S.o.println(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?

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

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

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]

Eg:

```
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 Whether 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 leaks:-

→ 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 leaks". (i.e, Memory Leak is a useless object which is nt eligible for G.c)

→ We can resolve memory Leaks by making useless objects for G.c explicitly & by invoking 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 using 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 disabled automatically at runtime. based on our requirement we can enable & disable assert Statements & Bydefault assertions are disabled.

→ 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 -Source 1.3 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 assertion satisfied & rest of the program will be executed normally.

→ If b is false, then our assertion 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 Javae Test ✓

10

*③ Java -ea Test ←

R.E: AssertionError

(2) Augmented Version :-

→ we can Augment some discription by using augmented version to the AssertionError.

```
assert(b) : d;
```

b → should be booleantype

d → any description, can be any type. but recommend 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
{
p.s.v.m(String[] args)
{
int x=10;
=
assert(x==10):++x;
=
S.o.p.ln(x);
}}
```

```
assert(x>10): ++x;
```

✓ javac Test.java ↵

✓ java Test ↵
10

✓ java -ea Test ↵
10

```
javac Test.java

java Test
10

java -ea Test
```
R.E: AssertionError: 11

Conclusion(1):-

```
assert(e1) : e2 ;
```

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

Ex:-
```
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.

Ex(1) :-

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.

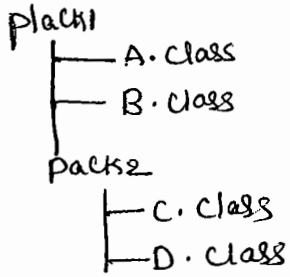
Ex(2) :-

① Java -ea:pack1.A

② Java -ea:pack1.B -ea:pack1.pack2.D

③ Java -ea -da:pack1.B

Ex 2:-



→ To enable assertions in only A class

① java -ea:pack1.A

→ To enable assertions in Both B & D classes

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

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

java -ea -da:pack1.B

→ To enable assertions in every class of pack1 & it's Sub packages

java -ea:pack1...

→ To enable 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)
{
 if(x<100)
 {
  throw new IAE();
 }
}
```

proper way

```
withdraw(int x)
{
 assert(x>=100);
}
```

Inproper 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 arguments.

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

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

6) AssertionError :-

→ 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

250

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 Errors in Exception handling.
18. Customized Exception.
19. Top-10 Exceptions.

Exception :-

→ when unwanted, unExpected Event that disterbes noamal flow of paogram is Called "Exception".

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

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

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

Ex:- If our paograming requirement is to read data from the file locating at Landon & at runtime if that file is not available our paogram should not be teaminated abnoamally. we have to paovide a local file to Continue rest of the program noamally. this is nothing but Exception Handling.

Syn:-
```
try
{
 read data from London file
}
catch(FileNotFoundException e)
{
 use local file and Continue rest of the paogram-
 noamally.
}
```

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.

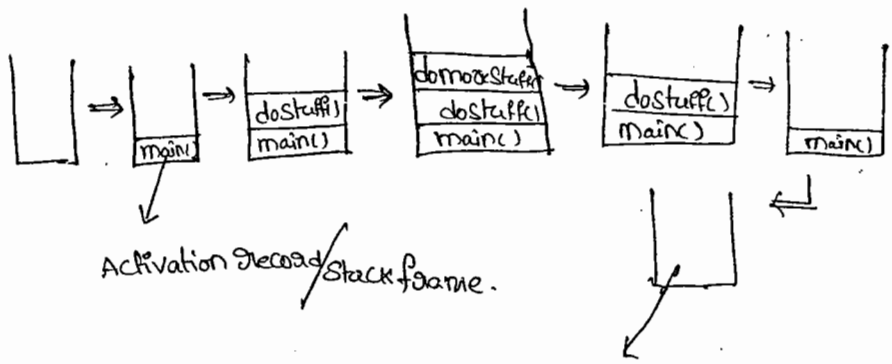
→ Each Entry in the Stack is Called "Activation Record" or "Stack frame".

→ 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.

Eg:-

```
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 handoveras 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 Entay from the Stack.

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

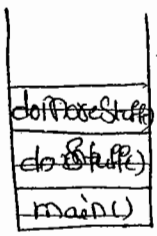
→ Just before terminating the program abnormally JVM handovers the responsibility of Exception handling to the default Exception handler.

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

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.pin(10/0);
}
}
```



Runtime Stack

Exception in thread "main" : java.lang.AE : / by Zero

name of exception → | ← description

at Test.doMoreStuff()
at Test.doStuff()
at Test.main()

} Stack Trace.

Exception hierarchy:-

→ Throwable class acts as a root for entire Java Exception hierarchy.

It has the following 2 child classes

1. Exception
2. Errors

1. Exception :-

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

2. Errors :-

→ 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.
ArithmaticException, FireAccidentException.

→ Whether Exception is checked or unchecked Compulsorily it should occur at runtime only. There is no chance of occurring 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

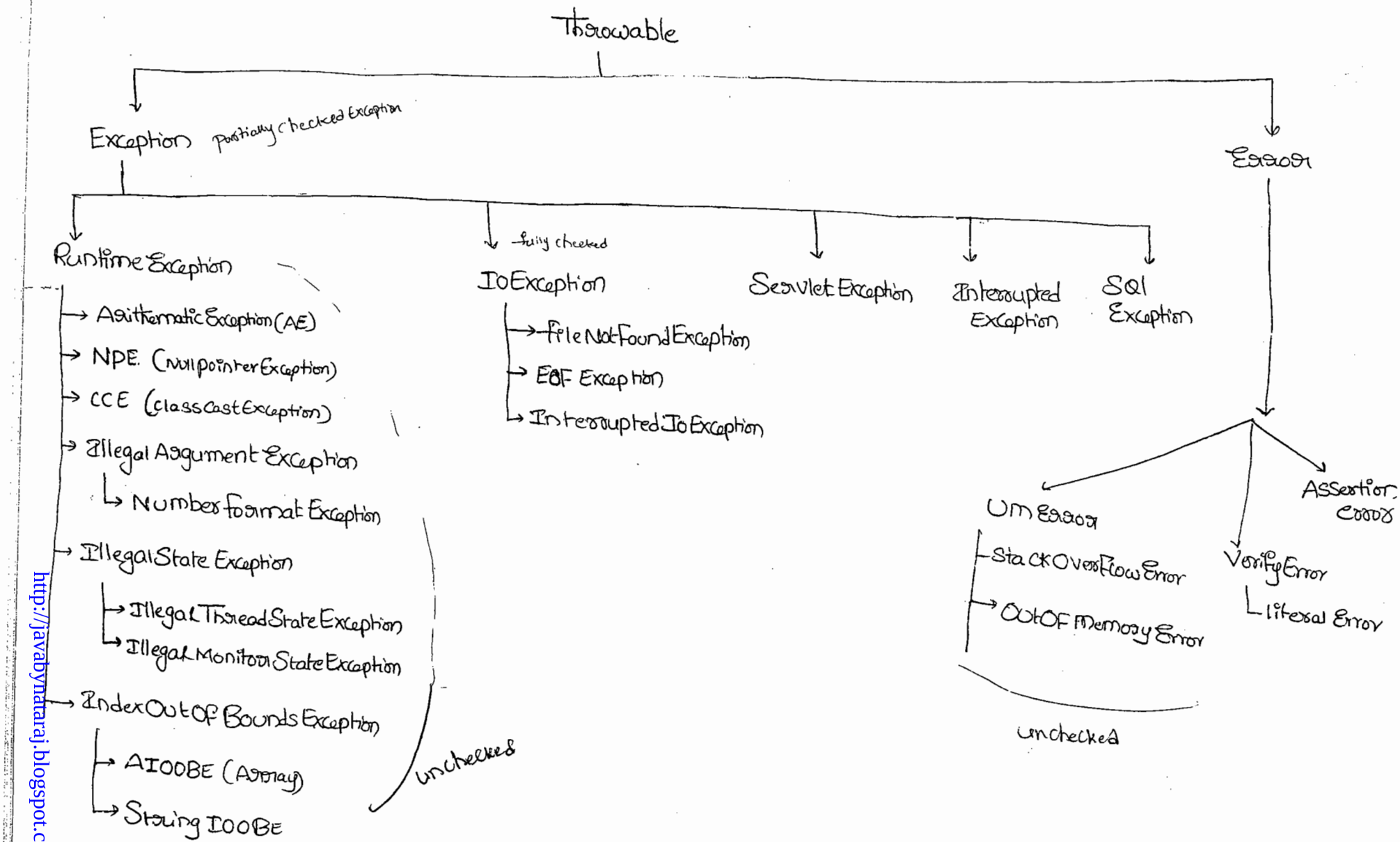
4) NullPointException : unchecked

5) InterruptedIOException : fully checked

6) SQLException : fully checked.

Note:

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



Customized Exception Handling by Try-catch:-

→ We can maintain Risky Code with in the Try block & corresponding Handling Code inside Catch block

```
try
{
   Risky Code;
}
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
{
 Stat 1;
 Stat 2;
 Stat 3;
}
Catch(xxx e)
{
 State 4;
}
State 5;
```

Case 1 :-

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

Case 2 :-

→ If the Exception raised at Statement 2 & corresponding Catchblock matched, 1, 4, 5 are normal terminations

Case 3 :-

→ If an Exception raised at Statement 2 & the corresponding Catchblock not matched, 1 followed by Abnormal Termination.

Case 4 :-

→ 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 format.

Name of Exception : discription
Stack trace

(2) toString() :-

→ It prints Exception information in the following format.

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();
 S.o.p(e); (or) S.o.pln(e.toString()); 
 S.o.pln(e.getMessage());
 }
}
}
```

A.E : / by zero
at test.main()

A.E : / by zero

/ 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(ArE 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"

Child to parent is follows

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

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

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 blocks because there is no garenty for the Execution of every Statement.

→ It is never recommended to define Clean-up code with in the Catch-blocks, 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.

Eg1:-
```
try
{
  risky code;
}
catch(xxx e)
{
  handling code;
}
finally
{
  Clean-up Code;
}
```

Eg2:-

```
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");
  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 overriding 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 Grabage 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(44 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(xxe) { }
S.o.pln("Hello");
A | 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
{
}
```
C.E!- finally without try

(11)
```
try
{
}
Catch(AE e)
{
}
Catch(Exception e)
{
}
```

(12)
```
try
{
}
Catch(exception e)
{
}
Catch(AE 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 corresponding 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 Stat 2 or Stat 6, it is always abnormal-terminate

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 blocks 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 blocks 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 & Coresponding Catch matched Then 1, 2, 3 ..., 10, 11, 12, Normal

Case 10 :-

→ If an Exception raised at State 8 & Coresponding Catch has not matched. Then 1, 2, 3 - - - -, 11, Abnormal

Case 11 :-

→ If an Exception raised at State 9 & Coresponding Catch matched. Then 1, 2, 3, ... , 8, 10, 11, 12, Normal

Case 12 :-

→ If an Exception raised at State 9 & Coresponding 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 ArithmaticException("/ by zero");
```

- `throw` : To hand-over our created Exception object to the JVM manually.
- `new ArithmaticException("/ by zero")` : 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.println(10/0);
 }
}
```

* In this case A.E object created internally & hand-over that object automatically by the main().

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

→ In this case we created A.E object and we hand-over it to the JVM manually by using throw-keyword.

→ In General, we can use throw keyword for customized Exceptions.

Case 1:-

→ If we are saying 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 ~~Compile~~ 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 Incompatible ~~State~~ types.

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

C.E: Incompatible Types
found : Test
Required : Java.lang.Throwable

```
class Test extends RuntimeException
{
 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 Checked Exception Compulsory we should handle it, other wise we'll 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 uncheeked Exceptions, it is not required to use throws keyword.

Eg:-
```
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 can't be Compiled. 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
{
p.v.m1() throws test
{
}
}
```

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

```
class Test extends Exception
{
p.v.m1() throws Test
{
}
}
```

Case(1) :-

(Checked)

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

C.E:- unreported Exception java.lang.Exception must be caught or 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 never we can't define Catch blocks for that Exception otherwise we will get Compiletime Error. but this rule is applicable for only fully checked Exceptions.

Eg!

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

o/p!- 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 Incompatible 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 u r age is already crossed marriage age");
 }
 else if(age <18)
 {
 throw new TooYoungException("ur age is already crossed marriage age..plz wait Some more time");
```

```
    else
    {
      S.o.pln("you will get match details by mail");
    }
}
```

Note:-

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

Top-10 Exceptions :-

21-02-11

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

1. JVM 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 RuntimeException and hence it is unchecked.

→ Raised automatically by the JVM, when ever we are taying to access perfoarm 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 taying to perfoarm recursive method invocation.

Ex:- Class Test

{

p.s.v.m m1()

{ m2();

}

p.s.v.m m2()

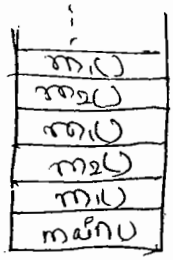
{ m1();

}

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 Sainu ↵

→ If Sainu.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 taying to typeCast parent object to the child type.

Ex):-

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

~~Stack~~:
```
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.println(s.length());
  }
}
```

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

⑦ IllegalArgumentException :-

→ 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 argument

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

⑧ NumberFormatException

→ 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

Eg:-
```
int i = Integer.parseInt("10");
X int i = Integer.parseInt("Ten");
```
RE
↑
IAE
↑
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.

Eg:-

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

Eg①:-

```
HttpSession session = req.getSession();
S.o.pln(session.getId()); 123u---
```

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

Eg②:-

```
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	JVM automatically (JVM Exception)
2. NPE	
3. SOFE	
4. NoClassDefFoundError	
5. ClassCastException	
6. ExceptionInInitializerError	
7. IllegalArgumentException	either programmer or API developer Explicitly (programatic Exceptions)
8. NumberFormatException	
9. IllegalStateException	

Exception propagation :-

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

Inner Classes

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

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

→ But Because of powerfull features & benefits of Innerclasses slowly 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 Innerclass concept.

Ex(1):-

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

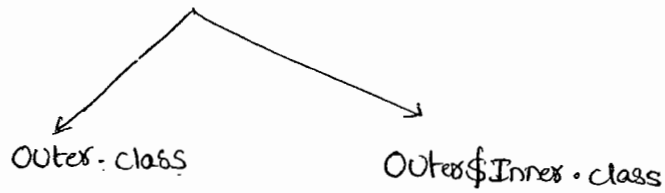
Normal or Regular Inner class :-

→ 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
  {
  }
}
```

Javac Outer.Java



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:- NoSuchMethodError : main.

Ex(3):

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

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

Javac Outer.java

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

23/08/11:-

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

Ex:-
```
class Outer
{
 class Inner
 {
  p.s.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 = o.new Inner();
i.m1();
```

→ `Outer.Inner i = new Outer().new Inner();`

→ `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();
```

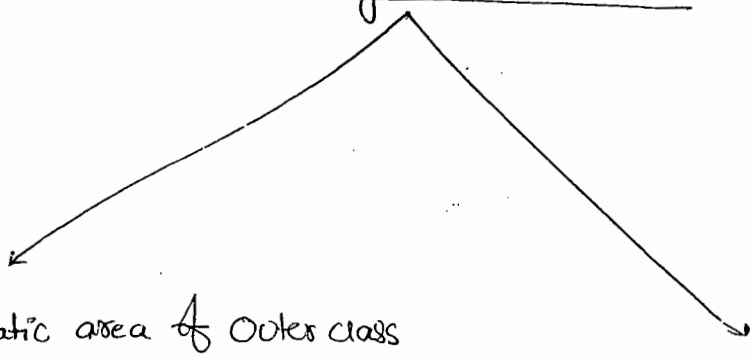
Accessing Inner class Code from Outside of outer class

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()
  }
}
```

Accoading Inner Class Code



from Static area of Outer class (or) from outside of outer class

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

from Instance are of outer class

```
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 Innerclass 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()
      {
        S.o.pln(x);   10
        S.o.pln(y);   20
      }
    }
    Inner i = new Inner();
    i.m2();
  }
```

o/p:- 10 20

```
    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;
    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();
}
}
```

→ if we declare final (final int y=20;)

o/p:- x=10
y=20

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 wont 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 are y, j.

⑤ 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 arrguements.

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("Sweetly");
      }
    };
    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 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 ~~Exec~~ exact output.

(b) Anonymous Inner class that implements an Interface :-

Eg :-

```
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);
 t.start();
 for(int i=0 ; i<10; i++)
 {
  S.o.pln("main Thread");
 }
}
}
```

→ r is an object of Runnable

(c) Anonymous Inner class that define inside method argument:- 29

Eg:-

```
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-i");
                  }
                 }
                }).start();

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

General class Vs Anonymus Inner class:-

→ A General class Can extend only one class at a time. where as Annonymus Innerclass also Can extend only one Class at a time.

→ A General class Can implement any no of Interfaces where as Annonymus 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 Outer class object i.e with out existing outer class object there may be a chance of existing Static Nested class object.

Eg:-

```
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.

Ex1.

```
Class Outer
{
 Static class Nested
 {
   public static void main(String[] args)
   {
    S.o.pln("Static Nested class main method");
   }
 }
 public 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.m1()
   {
    S.o.pln(x); ✗ ——→ C.E:- NON-static variable x can't be
    S.o.pln(y); ✓        referenced from static Nested Content
   }
 }
}
```

Diff b/w Normal Inner class & Static Nested class?

Innerclass	Static Nested Class
1) Inner class object is always associated with Outer class 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	3) 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. Wrapper classes (Auto boxing & Auto unboxing)

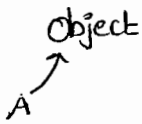
① Object :-

→ 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 made this class as parent for all Java classes So that its methods are bydefault available to every Java class Automatically

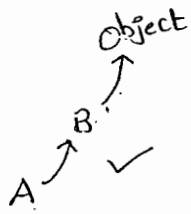
→ Every class in java is the child class of object either directly or Indirectly, If our class doesnt extend any other class then only our class is direct child class of Object.

Ex!- Class A
{
}

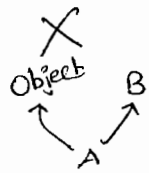


→ if our class extends any other class then our class is not direct child class of Object. it extends Object class indirectly.

Ex!- Class A extends B
{
}



multilevel Inheritance



multiple

→ 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 saying 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("Sathu", 102);
        S.o.pln(s1);  ⇒ 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

→ Top.

→→ 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:"
                                                    + rollno;
}
```

* 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@a23504
    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 over-Ridding 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
1
1
9
a(10)
b(11)
c(12)
d(13)
e(14)
f(15)

16 | 100
6 — 4 ⟶

64

Ex @1:

```
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 | 100
~~64~~ 4 ↑

64

10 ⇒ 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.ph(t1);      10
   S.o.ph(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 over class toString() method than it may not call hashCode() method.

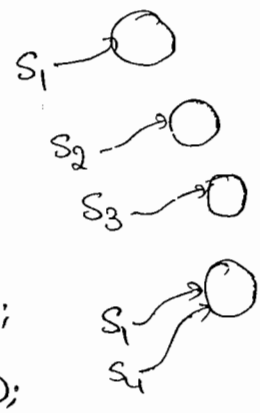
③ equals() method :-

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

```
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 argument 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;
    String 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.

differences 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 Heterogeneous type objects ~~equal~~ == operator ~~gives~~ 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.

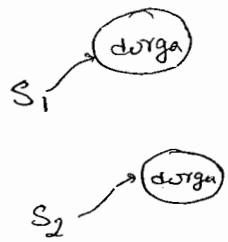
Note:-

(1) 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 overriden for Content Comparision.

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

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

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

1. If two objects are equal by .equals() Compulsory there hashCodes must be Same.
2. If two objects are not equal by .equals() then there are no restrictions 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() compulsory 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.

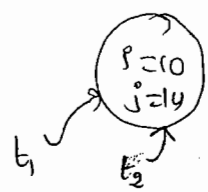
Deep cloning & Shallow cloning:-

→ The process of creating just duplicate reference variable but not duplicate object is called Shallow cloning.

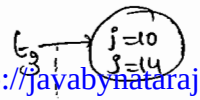
→ The process of creating exactly duplicate independent objects is by bydefault considered as deep cloning.

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

Shallow cloning



By default cloning means deep cloning.



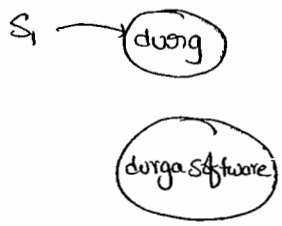
String class

24/05/11

Case(1) :-

Immutable

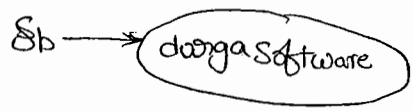
```
String s = new String("durga");
s.concate("software");
S.o.p(s);  durga
```



→ 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 StringBuffer 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
```

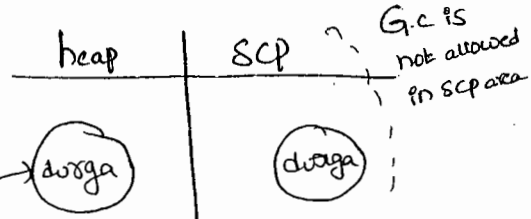
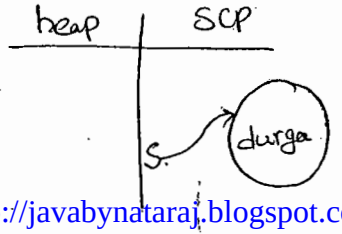
→ In String class .equals() method is overridden for Content Comparision. Hence .equals() method returns True if Content is Same eventhough Objects are different.

```
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 Compasision. 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	→ In this Case only one object will be Created in SCP and 's' is always pointing to that Object
	

Note:-

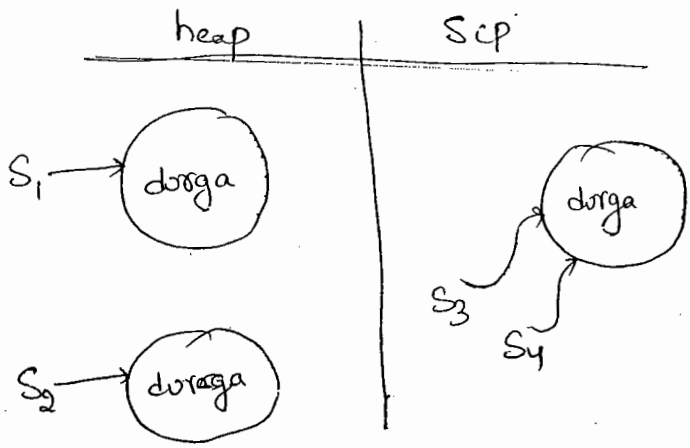
① G.C is not allowed to access in scp area hence eventhough object doesn't have any reference variable still it is not eligible for G.C, if it is present in scp area.

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.

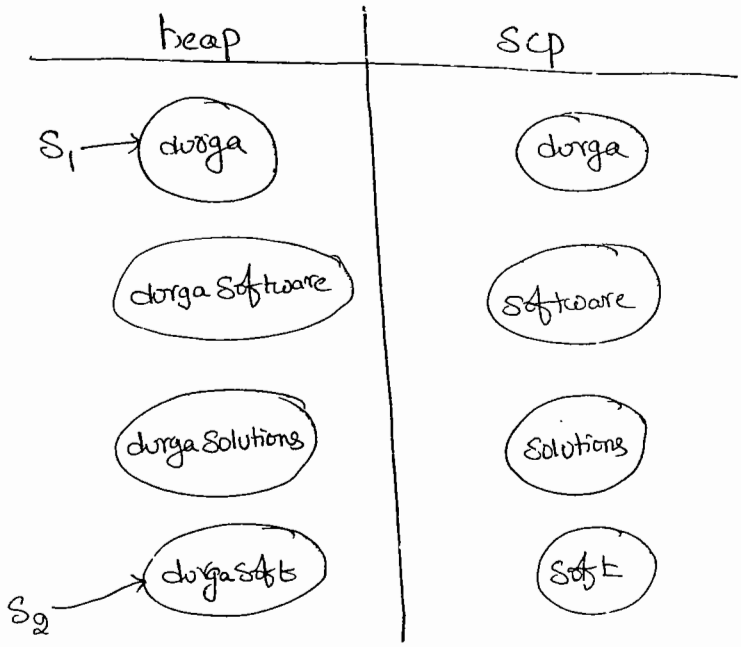
Ex③:-

```
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");
```



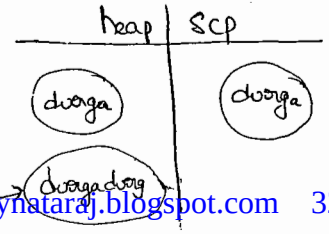
Note :-

→ 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

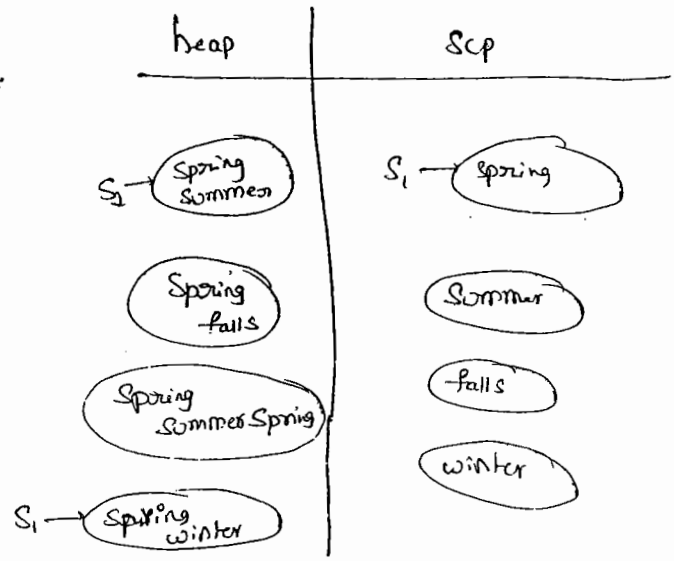
Ex④ :-

```
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:-

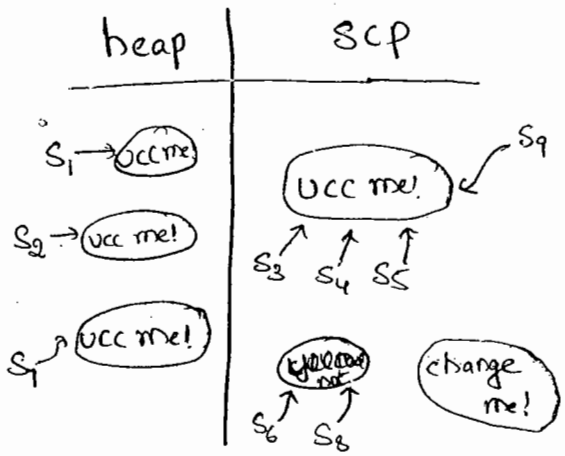
final String S = "raghu"; S is a Constant

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

Ex:-

String S1 = new String("you can't

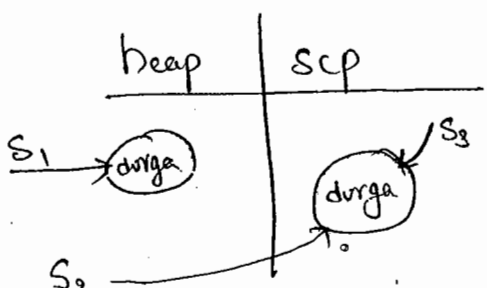
```
String s1 = new String("you cannot change me!");
String s2 = new String("you cannot change me!");
S.o.pln(s1==s2); false
String s3 = "you cannot change me!";
String s4 = "you cannot change me!";
S.o.pln(s1==s4); true
S.o.pln(s1==s3); false
String s5 = "you cannot "+"change me!";
S.o.pln(s3==s5); 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 :-

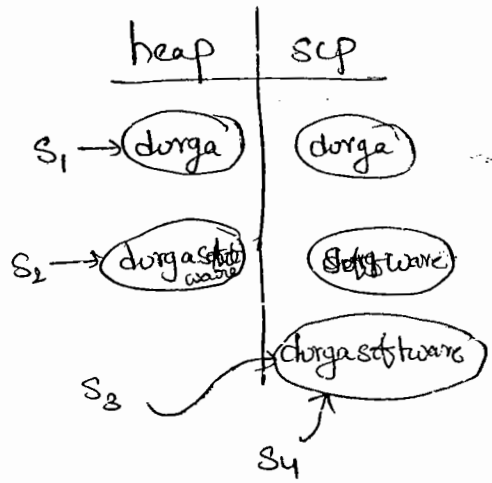
→ By using heap object reference if you want to get corresponding scp object reference then we should go for intern().

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 go for equals() 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();

Eg:- String s = "aabbbb";

S.o.pln(s.length); → 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();

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 : ctypet

Name : Srinivas

fathername: SitaRamaiah

Age : 22

DOB :

H.NO : 9-133

Street : Ramnagar

Substreet: Ramnagar

City: Ganapavaram

District : Guntur

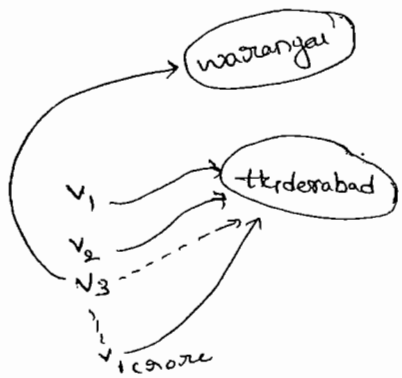
State : A-p

Country : India

PIN : 522619

Identification Name : xxxx xxxx

Submit



→ In our program if any String object required to use repetedely, 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 toying 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?

A) → 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?

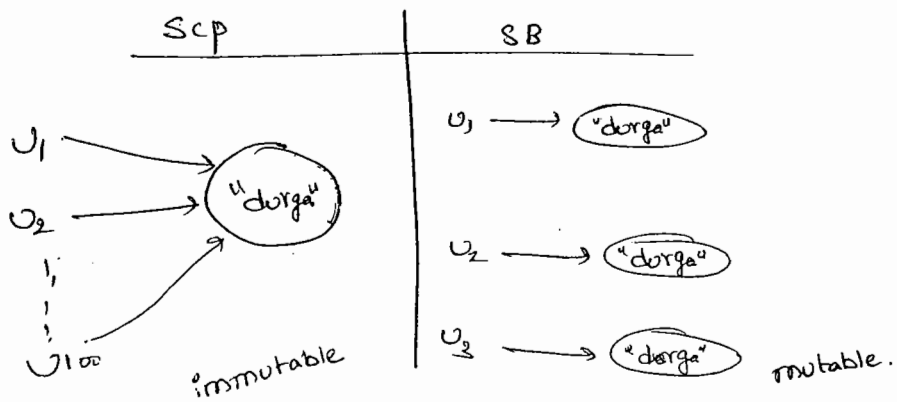
A) → Compulsory we should make String objects as immutable.

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 performaning 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 Commpulsary 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 & StringBuffers objects are mutable.



10/03/11

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

A) Yes,

Note:

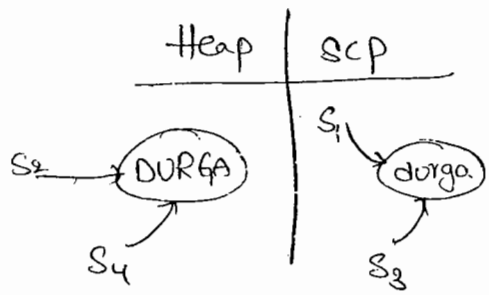
→ Once we created a String object we can't perform any changes in the existing object. If we are trying to perform any change with those changes a new object will be created on the heap.

→ 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.

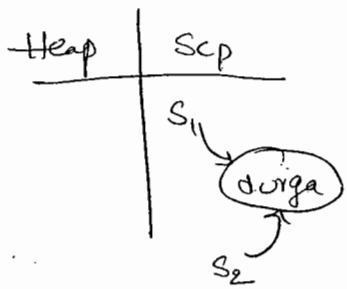
Ex:①

```
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
```



Ex:②

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



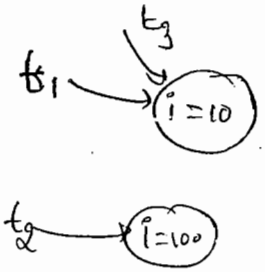
Creation of our own Immutable class :-

Ex:- 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.

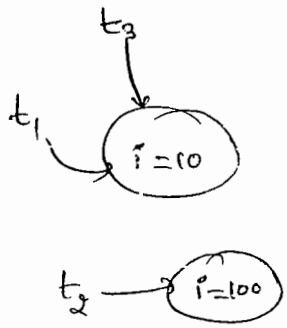
→ Because of our runtime method call if there is no change in the content then existing object only will be returned.

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.

$$\text{New Capacity} = (\text{Current Capacity} + 1) * 2$$

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:- StringIndexOutOfBoundException

S.o.pln(sb.charAt(5)); }

(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)
(double d)
(Object o)
} overloaded methods

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, 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 set 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 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 return type is same type only. Hence after applying a method on the result we can call another method with 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().delete(2,7).append("solutions");
S.o.pln(sb); //agdsolutions
```

final vs immutable :-

300

→ 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 value, in the corresponding object we can perform any type of change Eventhough 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 aerguement & the other can take String as aerguement.

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 properlly formatted then we will get R.E saying NumberformatException.

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 aerguement

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 aargument.

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 aargument & other Can take String as aargument.

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

Ex:- ✓ Boolean B = new Boolean (true);

✗ Boolean B = new Boolean (True);

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

→ If the Content Case insensitive String ~~(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 ("true"); ✓ false

Wrapper classes	Corresponding Constructor arrangement
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 its 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.

```
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);
```

Ex:-

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

radix: 2 to 36

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) :-

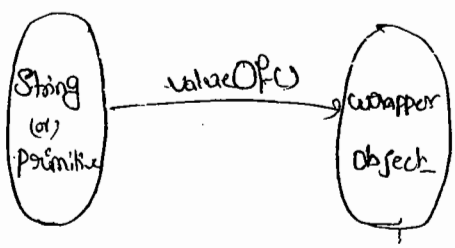
→ Every wrapper class including Character class Contains the following valueOf() to Convert primitive to Corresponding wrapper Object

```
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 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 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 premitive for the given boolean Object.

public boolean booleanValue();

eg:- Boolean B = Boolean.valueOf("durga");

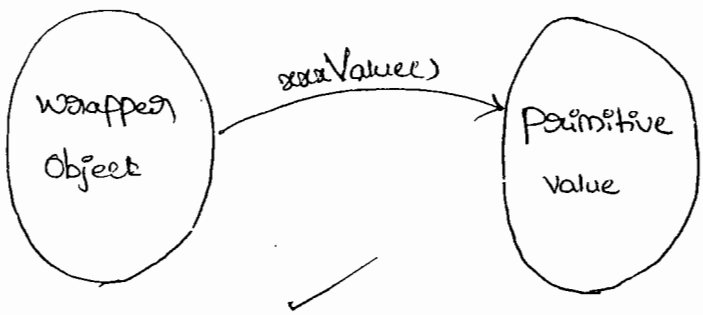
boolean b = B.booleanValue();

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 Character class contains the following parseXxx() to convert String to corresponding primitive.

```
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.

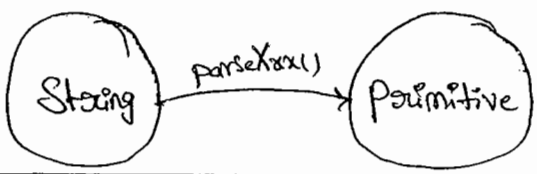
Eg:- public static primitive parseXxx(String s, int radix);

Eg:-

```
int i = Integer.parseInt("1111", 2);
So.pln(i); 15
```

2 to 36.

Note:-



(iv) toString() :-

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

form1() :-

→ Every wrapper class contains the following toString(), 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);

form3 :-

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

public static String toString(primitive p, int radix);

2 to 36

Eg: String s = Integer.toString(15, 2);

s.o.println(s); 1111

Form 4:-

→ Integer & Long classes contains the following toXxxString().

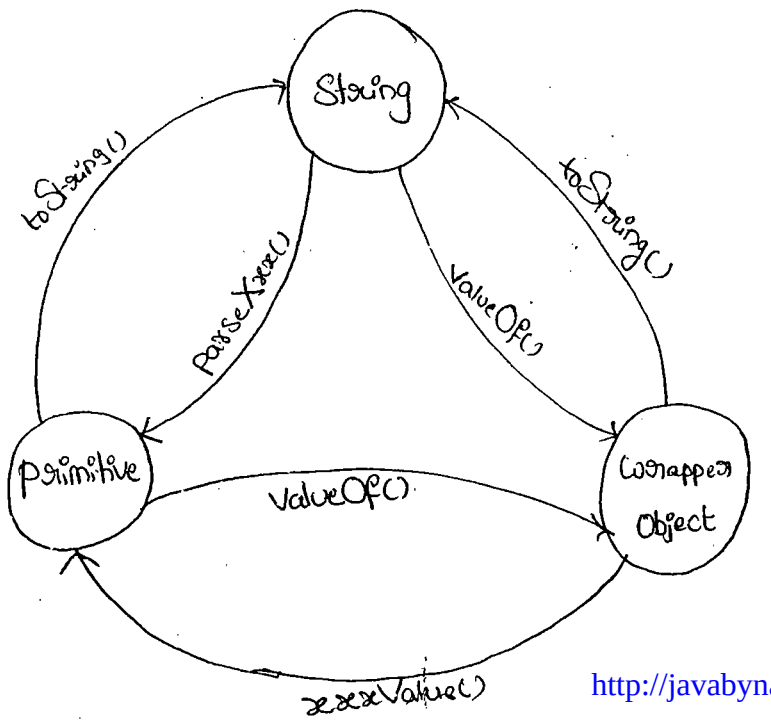
1. public static String toBinaryString(primitive p);
2. public static String toOctalString(primitive p);
3. public static String toHexString(primitive p);

Ex:- String s = Integer.toHexString(123)

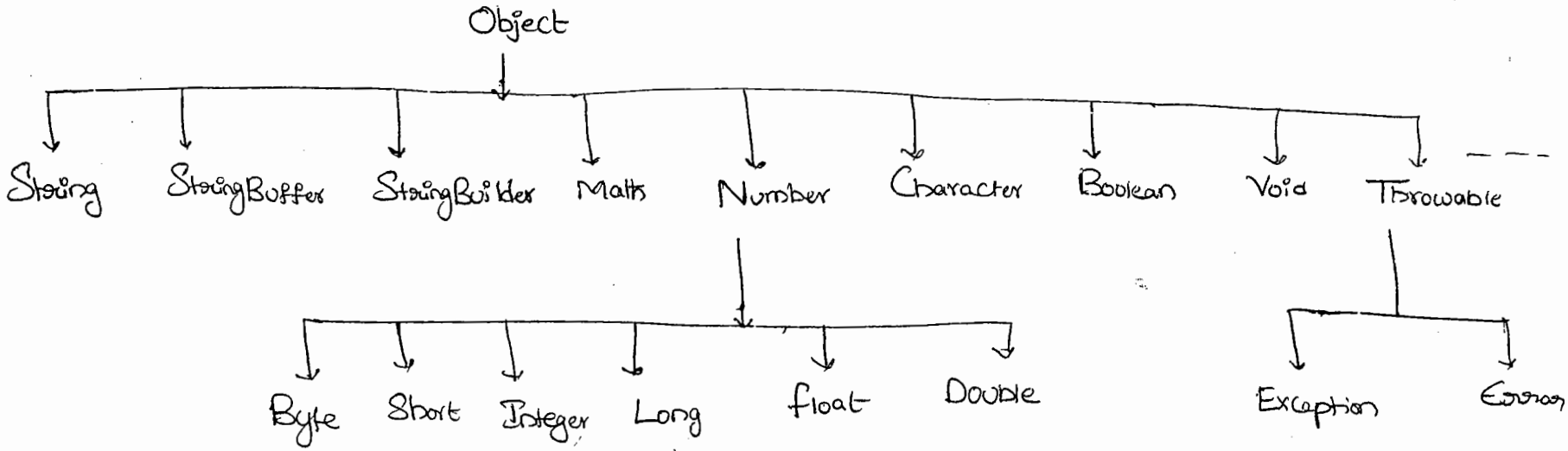
s.o.println(s) : "7b"

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 should be performed explicitly by the programmer.

Ex:-

① ArrayList l = new ArrayList();

l.add(10); ✗ C.E!.

Integer I = new Integer(10);

l.add(I); ✓

② Boolean B = new Boolean(true);

```
if(B)
{
 S.o.pln("Hello");
}
```

C.E!- Incompatible types
found : Boolean
required : boolean

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

→ But from 1.5 Version onwards 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".

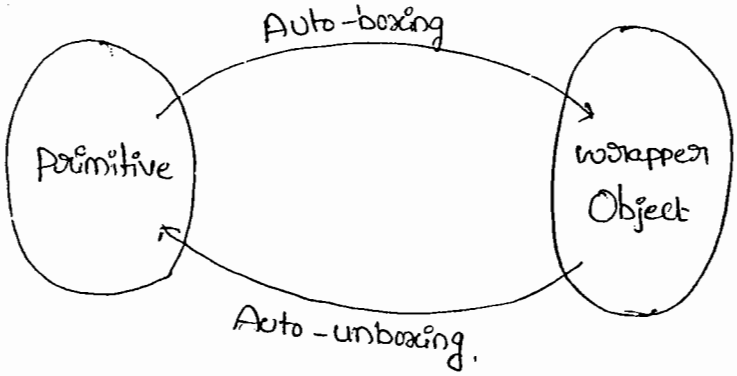
Ex:- ✓D Integer I = 10; [Compiler Converts int to Integer automatically by Autoboxing]

Auto-unboxing:-

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

Ex:- ✓D 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 xxxValue().

Exam purpose:-

ex①:-

```
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;
    S.o.pln(i);
  }
}
```

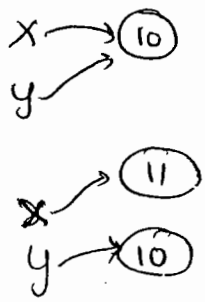
→ R.E:- NPE

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.

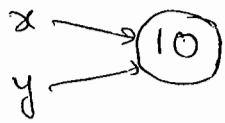
Ex4 :-

① Integer X = new Integer(10);
Integer y = new Integer(10);
S.o.pln(x==y); false ✓

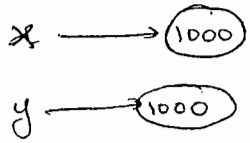
② Integer x = new Integer(10);
Integer y =10;
S.o.pln(x==y); false ✓

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.o.pln("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
}
}
```

→ Widening dominates Auto-boxing

Case(2)!-

→ Widening Vs Var-arg() :-

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 overloaded methods Compiler will always keeps the percidence in the following oerder.

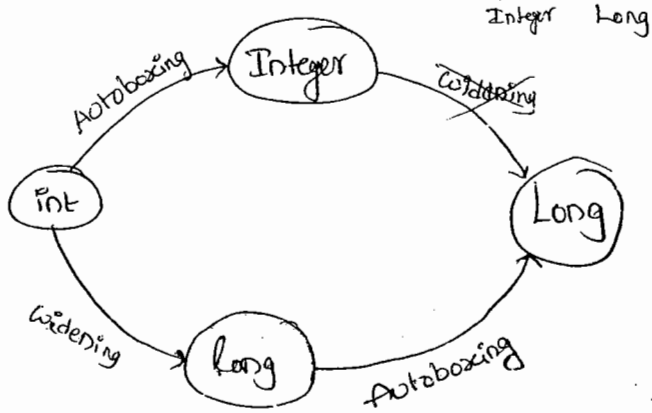
(i) Widdening

(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);
  }
}
```



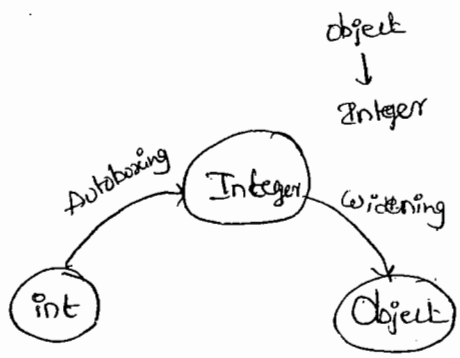
C.E:-

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); // 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;

05/04/11

Java.io package

File I/O :-

1. File
2. FileWriter
3. FileReader
4. BufferedReader
5. BufferedWriter
6. printWriter.

(1) File :-

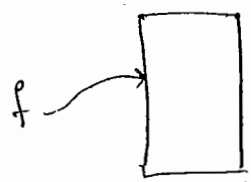
* File f = new File("abc.txt");

→ This line won't create any physical file, first it will check, is there any file named with abc.txt is available or not.

→ If it is available then f simply pointing to that file.

→ If it is not available then f represents just name of the file without creating any physical file.

```
File f = new File("abc.txt");
S.o.pln(f.exists()); // false
f.createNewFile();
S.o.pln(f.exists()); // true
```



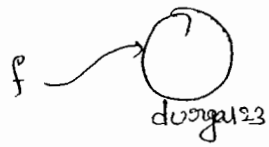
→ A Java file object can represent a directory also.

Ex:- File f = new File("durgai23");

S.o.p(f.exists()); false

f.mkdir();

S.o.p(f.exists()); true



Constructors :-

① File f = new File(String name);

→ Create a java file object to represent name of a file or directory.

② File f = new File(String subdir, String name);

→ To create a file or directory present in some other sub-directory.

③ File f = new File(File subdir, String name);

Ex:- ① Write code to create a file named with abc.txt in current working directory.

```
File f = new File("abc.txt");
f.createNewFile();
```

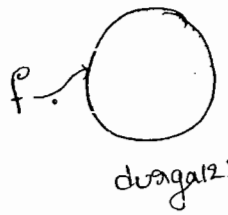
② w.c. to create a directory named with xyz in current working directory.

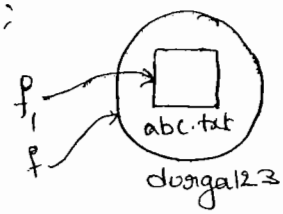
```
File f = new File("xyz");
f.mkdir();
```

③ w.c. to create a directory named with durga123 in current working directory. in that directory create a file named with abc.txt.

A)

```
File f = new File("durga123");
f.mkdir();
File f1 = new File("durga123", "abc.txt");
f1.CreateNewFile();
File f1 = new File(f, "abc.txt");
f1.CreateNewFile();
```





(or)

Important methods of File class :-

① boolean exists();

→ returns true if the physical file or directory present

② boolean CreateNewFile();

→ first this method will check whether the specified file is already available or not. if it is already available then this method won't return false without creating newfile. if it is not already available then this method returns true after creating new file.

③ boolean mkdir();

④ boolean isFile();

(5) boolean isDirectory();

(6) String[] list();

→ It returns the names of all files & Sub-directories present in the Specified directory.

(7) boolean delete();

→ To delete a file or directory

(8) long length();

→ returns the no. of characters present in the Specified-file

Ex:- W.a.p to print the names of all files & Sub-directories present in "D:\durga_classes".

```
import java.io.*;
class Test
{
  p.s.v.m(String[] args) throws Exception
  {
    File f = new File("D:\\durga_classes");
    String[] s = f.list();
    for(String s1 : s)
    {
      S.o.pln(s1);
    }
  }
}
```

(2) FileWriter :-

→ We can use FileWriter object to write character data to the file.

Constructors :-

① FileWriter fw = new FileWriter(String name);

② FileWriter fw = new FileWriter(File f);

→ The above 2 constructors ment for overridding. If we want to perform append instead of overridding then we have to use the following constructors.

③ FileWriter fw = new FileWriter(String name, boolean append);

④ FileWriter fw = new FileWriter(File f, boolean append);

→ If the specified file is not already available then the above constructors will Create that file.

Methods of FileWriter :-

① write(int ch);

To write a single character to the file.

② write(char[] ch);

To write an array of characters to the file.

③ write(String s);

To write a String to the file.

(4) flush() :-

→ to give the guarantee that Last Character of the data also reaches to the file.

(5) close() :-

Ex!- demo program for the FileWriter.

```
import java.io.*;
Class FileWriterDemo2
{
 P.S.V.m(String[] args)
  {
   FileWriter fw = new FileWriter("wc.txt", true);   → appending
   fw.write(100); // adding a single character
   fw.write("durgaInSoftwareSolutions");
   char[] ch = {'a','b','c'};
   fw.write('\n');
   fw.write(ch);
   fw.write('\n');
   fw.flush();
   fw.close();
  }
}
```

100 → d

O/P!- durga
SoftwareSolutions
abc

(3) FileReader :-

→ We can use FileReader to Read character data from the file

Constructors :-

1. FileReader fr = new FileReader(String name);
2. FileReader fr = new FileReader(File f);

Methods of FileReader :-

(i) int read() :-

* It attempts to read next character from the file and returns its unicode value.
* If the next character is not available, then this method returns "-1".

(ii) int read(char[] ch) :-

* It attempts to read enough characters from the file into the char array & returns the no. of characters which are copied from file to the char[].

(iii) close();

Ex!- on FileReader

```
import java.io.*;
Class FileReadDemo
{
    P-S-V-M(String[] args) throws IOExceptions
```

```
File f = new File("wc.txt");
FileReader fr = new FileReader(f);
S.o.pln(fr.read()); //Unicode of first character
Char[] ch2 = new Char[(int)(f.length())];
fr.read(ch2); // file data Copied to array
for(Char ch : ch2)
{
   System.out.print(ch);
}
S.o.pln("***----------");
FileReader fr1 = new FileReader(f);
int i = fr1.read();
while(i != -1)
{
   S.o.pln((Char) i);
   i = fr1.read;
}
}
}
```

* Usage of FileWriter & FileReader is not recommended because :-

1) .. while writing data by filewriter we have to insert line Seperators manually which is a bigger headache to the programmer.

2) By using fileReader we can read data character by character which is not convineiant to the programmer.

3) To resolve these problem SUN people introduced BufferedWriter & BufferedReader classes.

(ii) BufferedWriter :-

→ We can use BufferedWriter to write character data to the file.

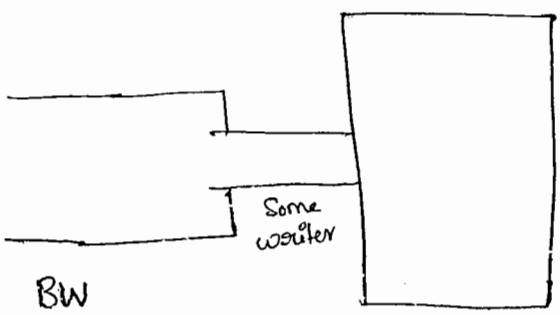
Constructors :-

1) BufferedWriter bw = new BufferedWriter(Writer w);

2) BufferedWriter bw = new BufferedWriter(Writer w, int buffersize)

Note:-

* BufferedWriter never communicates directly with the file compulsory it should communicate via some Writer object only.



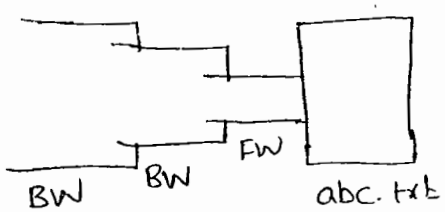
Q) Which of the following are valid.

✗ ① BufferedWriter bw = new BufferedWriter("abc.txt");

✗ ② BufferedWriter bw = new BW(new file("abc.txt"));

✓ ③ BufferedWriter bw = new BufferedWriter(new FileWriter("abc.txt"));

✓ ④ BW bw = new BW(new BW(new FW(new file("abc.txt"))));



Important methods of BufferedWriter :-

① write(int ch)

② write(char[] ch)

③ write(String s)

④ flush()

⑤ close()

⑥ newLine() :- to insert a newline character

Q:- When Compared with FileWriter which of the following capability is available as a Seperate method in BufferedWriter.

A) ④

① Writeing data to the file. ④ inserting a line seperator.

② flushing the Stream.

③ closeing the Stream.

Ex:-

```
import java.io.*;
class BufferedWriterDemo
{
  P.S.V.m(String[] args) throws IOException
  {
    File f = new File("wc.txt");
    FileWriter fw = new FileWriter(f);
    BufferedWriter bw = new BufferedWrite(fw);
    bw.write(100);
    bw.newLine();
    char[] ch1 = {'a','b','c','d'};
    bw.write(ch1);
    bw.newLine();
    bw.write("durga");
    bw.newLine();
    bw.write("Software Solutions");
    bw.flush();
    bw.close();
  }
}
```

o/p:-

```
d
abcd
durga
SoftwareSolutions
```

wc.txt

Note:- When ever we are closing BufferedWriter automatically underlaying writers will be closed

bw.close(); ✓	fw.close(); X	fw.close(); bw.close(); X

(iv) BufferedReader() :-

* The main Advantage of BufferedReader over FileReader is we can read the data Line by Line instead of Reading character by character. This approach improves performance of the System by reducing the no. of read operations.

Constructors :-

(i) BufferedReader br = new BufferedReader(Reader r);

(ii) " " (Reader r, int buffersize);

Note :-

* BufferedReader can't communicate directly with the file compolsary it should communicate via some reader object.

Important methods :-

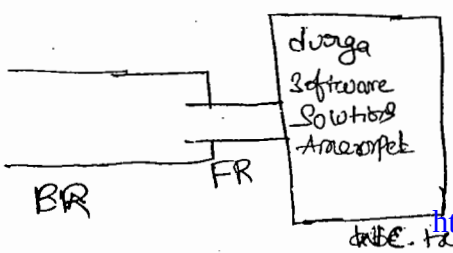
① int read();

② int read(char[] ch);

③ close();

④ String readLine();

* It attempts to find the nextLine & if the nextLine is available then it returns it, otherwise it returns null.



Ex:-
```
import java.io.*;
class Buffered
{
  p.s.v.m(String[] args) throws Exception
  {
    FileReader fr = new FileReader("wc.txt");
    BufferedReader br = new BufferedReader(fr);
    String line = br.readLine();
    while(line != null)
    {
      S.o.println(line);
      line = br.readLine();
    }
    br.close();
  }
}
```

o/p:-
```
durga
Software
Solutions
-Ameerpet
```

Note:-

* when ever your closing BufferedReader underlying Readers will be closed.

⑤ PrintWriter() :-

* This is the most enhanced writer to write a character data to file. By using FileWriter & BufferedWriter we can write only character data but by using PrintWriter we can write any primitive datatypes to the file.

Constructors:-

① PrintWriter pw = new PrintWriter(String name);

② PrintWriter pw = new PrintWriter(File f);

③ PrintWriter pw = new PrintWriter(Writer w);

Methods:-

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

print(char ch)	println(char ch)
print(int i)	println(int i)
print(long l)	println(long l)
print(double d)	println(double d)
print(String s)	println(String s)
print(char[] ch)	println(char[] ch)
⋮	⋮

Ex:-

```
import java.io.*;
Class PrintWriterDemo1
{
  p.s.v.m(String[] args) throws IOException
  {
```

```
FileWriter fw = new FileWriter("cc.txt");
PrintWriter pw = new PrintWriter(fw);
pw.write(100); // d
pw.println(100); // 100
pw.println(true); // true
pw.println('c'); // c
pw.println("durga"); // durga
pw.flush();
pw.close();
}
}
```

o/p:-

```
d100
true
c
durga
```

cc.txt

Q:- what is the diff. b/w the following

(a) pw.write(100);

(b) pw.print(100);

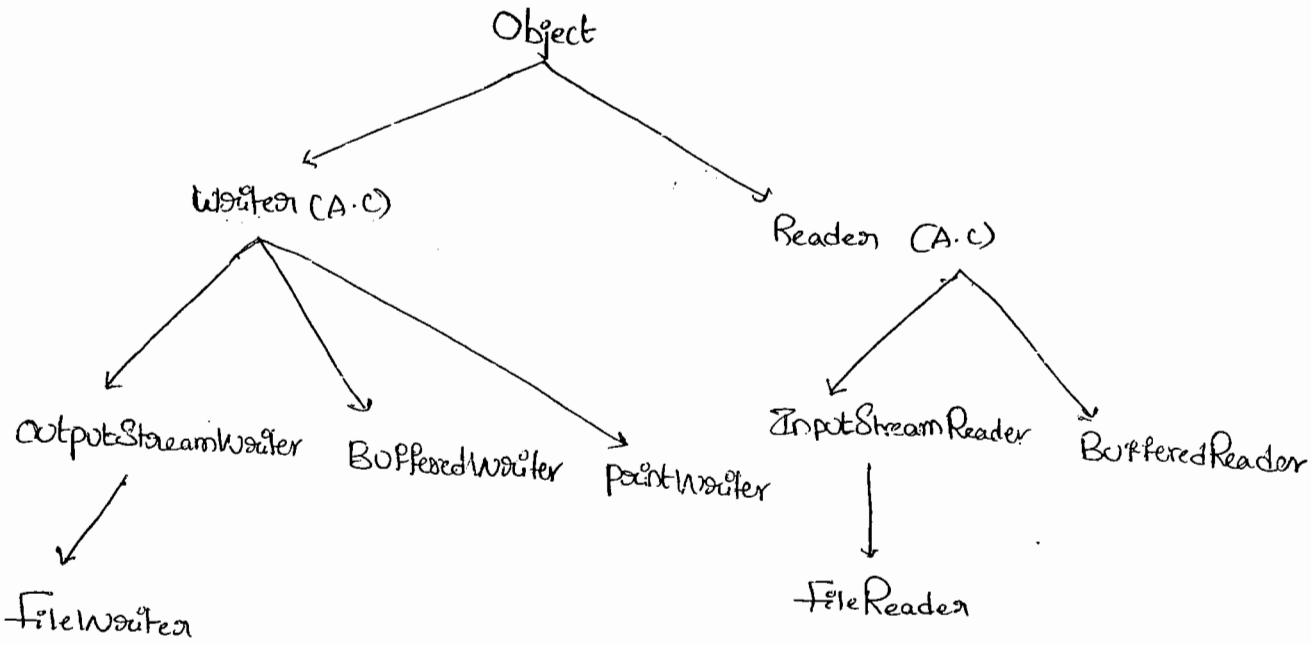
Ans:-

pw.write(100);

→ In this case the corresponding Character d will be added to the file

pw.print(100)

→ In this case the corresponding int value 100 directly will be added to the file

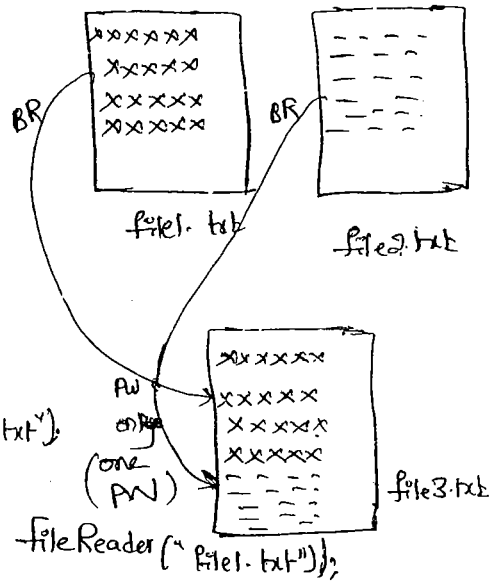


Note:-

* Readers & writers ment for handling character data (any primitive data type) +
* To handle Binary data (like images, movie files, Jar files ....) we should go for Streams.
* We can use InputStream to Read Binary data & OutputStream to write a Binary data.
* We can use ObjectInputStream & ObjectOutputStream to read & write Objects to a file respectively (Serialization).
* The most Enhanced Writer to write character data is PrintWriter whereas the most Enhanced Reader to read character data is BufferedReader.

*) W.a.p to merge data from two files into a 3rd file. 321

file3.txt = file1.txt + file2.txt




```
import java.io.*;
Class FileMerger
{
 P.S.V.M (String[] args) throws Exception
 {
   PrintWriter pw = new PrintWriter("output.txt");
   BufferedReader br = new BufferedReader(new FileReader("file1.txt"));
   String line = br.readLine();
    While (line != null)
    {
       pw.println(line);
        line = br.readLine();
    }
    br = new BufferedReader(new FileReader("file2.txt"));
     line = br.readline();
     while (line != null).
     {
       Pw.println(line);
        line = br.readLine();
     }
   pw.flush();
   br.close();
   pw.close();
 }
}
```

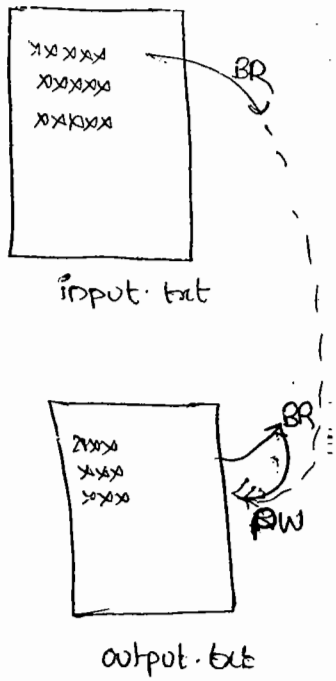
Ex2:-

w.a.p to merge data from 2 files into a 3rd file but merging should be done line by line alternatively.

```
import java.io.*;
Class FileMerger2
{
  P.S.V.m(String[] args) throws IOException
  {
    PrintWriter pw = new PrintWriter("output.txt");
    BufferedReader br1 = new BufferedReader(new FileReader("file1.txt");
    BufferedReader br2 = new BufferedReader(new fileReader("file2.txt");
    String line1 = br1.readLine();
    String line2 = br2.readLine();
    while((line1 != null) | (line2 != null))
    {
     if(line1 != null)
      {
        pw.println(line1);
        line1=br1.readLine();
      }
     if(line2 != null)
      {
       pw.println(line2);
        line2 = br2.readLine();
      }
    }
pw.flush();          pw.close();
br1.close();
br2.close();
}}
```

3) W.a.p to merge data from two files into a 3rd file But merging should be done para by para. assume that there is a Blank Line B/w Every 2 paras?

4) W.a.p to delete duplicates from the given i/p file ?



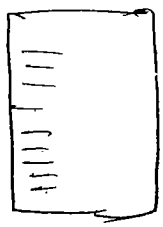

```
import java.io.*;
class DuplicateEliminator
{
 p.s.v.m(String[] args)throws Exception
 {
 BufferedReader br1 = new BufferedReader(new FileReader("i/p.txt"));
 PrintWriter pw = new PrintWriter("o/p.txt");
 String line = br1.readLine();
 while(line != null)
 {
  boolean available = false;
  BR br2 = new BR(new FR("o/p.txt"));
  String target = br2.readLine();
  while(target != null)
  {
   if(line.equals(target))
   {
    available = true;
    break;
   }
  target = br2.readLine();
  }
```

```
if(available == false)
 {
   pw.println(line);
   pw.flush();
 }
  line = bar.readLine();
}
pw.flush();
bar.close();
bar2.close();
pw.close();
}
}
```

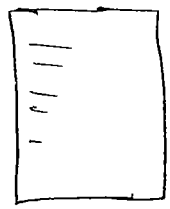
(5) W.a.p to perform file Extraction (result.txt = total.txt - delete.txt)

```
import java.io.*;
Class fileExtraction
{
  p.s.v.m (String[] args) throws Exceptions
  {
    BufferedReader bar = new BufferedReader(new FileReader
                                ("mobile.txt"));
    PrintWriter pw = new printWriter("output.txt");
    String line = bar.readLine();
    while(line != null)
    {
      boolean available = false;
      BR bar2 = new BR(new FR("delete.txt"));
```

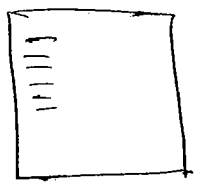
```
String target = br2.readLine();
while(target != null)
{
 if(line.equals(target))
 {
   available = true;
   break;
 }
  target = br2.readLine();
}
if(available == false)
{
  pw.println(line);
}
line = br1.readline();
}
  pw.flush();
  br1.close();
  br2.close();
  pw.close();
 }
}
```

total.txt



desire.txt



result.txt

Profession etc

Heretic = a person who holds Contraversial beliefs, especially Contrary to religion.
opposite

Paralyzed argument

Serialization

① Introduction

② Object Graphs in Serialization

③ Customized Serialization

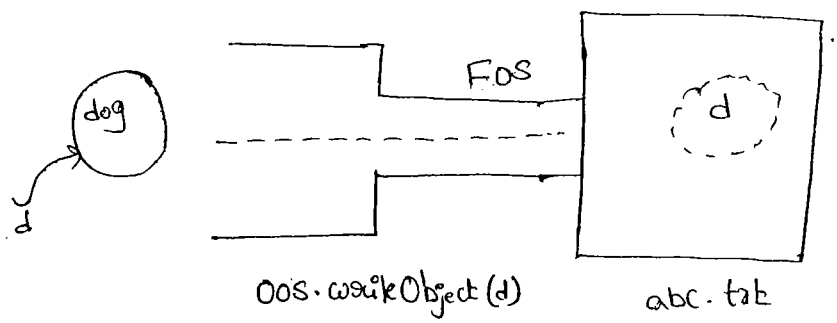
④ Serialization w.r.t Inheritance.

Serilization :-

→ The process of writing state of an object to a file is called Serialization.

→ But strictly it is a process of Converting an object from java Supported form to either file Supported form or network Supported form.

→ By using `FileOutputStream` and `ObjectOutputStream` classes we can achieve Serialization.

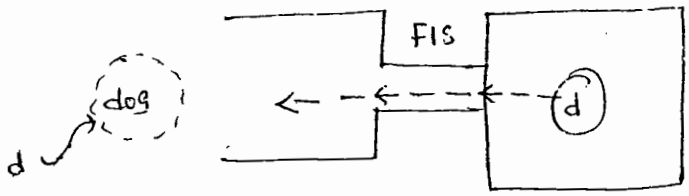


DeSerialization :-

→ The process of reading state of an object from a file is called DeSerialization.

→ But Strictly Speaking, It is the process of Converting an Object from either Network Supported form or File Supported form to java Supported form.

→ By using FileInputStream and ObjectInputStream classes we can achieve De-Serialization.



OIS.readObject(d)

Ex:-

```
import java.io.*;
Class Dog implements Serializable
{
  int i=10;
  int j=20;
}
Class SerializableDemo
{
  p.s.v.m(_____)throws Exception
  {
    Dog d1 = new Dog();
```

Serialization

```
FileOutputStream fos = new FileOutputStream("abc.txt");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(d1);
```

```
FileInputStream fis = new FIS("abc.txt");
OIS oos = new OIS(fis);
Dog d2 = (Dog) ois.readObject();
S.o.println(d2.i + "----" + d2.j);
}
}
```

→ We can perform Serialization only for Serializable objects.

→ An Object is said to be Serializable iff the corresponding class implements Serializable interface.

→ Serializable interface present in java.io package and doesn't contain any methods, it is a marker interface.

→ If we are trying to serialize a non-Serializable object we will get Run time exception saying NotSerializableException.

transient Keyword :-

→ At the time of Serialization if we dont want to Serialize the value of a particular variable to meet the Security Constraint we have to declare those variables with "transient" keyword.

→ At the time of Serialization JVM ignores original value of transient variable and Saves default value.

transient Vs Static :-

→ Static variables are not part of object hence they won't participate in Serialization process. Due to this declaring a Static variable as transistent there is no impact.

transisent Vs final :-

→ final variables will be participated into Serialization directly by their values Hence declaring a final variable with transistent there is no impact.

Summary :-

declaration	o/p
① int i = 10 int j = 20	10 ----- 20
② transient int i = 10; int j = 20	0 ----- 20
③ transient final int i = 10; transient int j = 20;	10 ----- 0
④ transient int i = 10; transient static int j = 20;	0 ---- 20

Object Graph in Serialization :-

→ When ever we are trying to Serialize an object the Set of all objects which are reachable from that object will be Serialized automatically this group of objects is called "Object Graph".

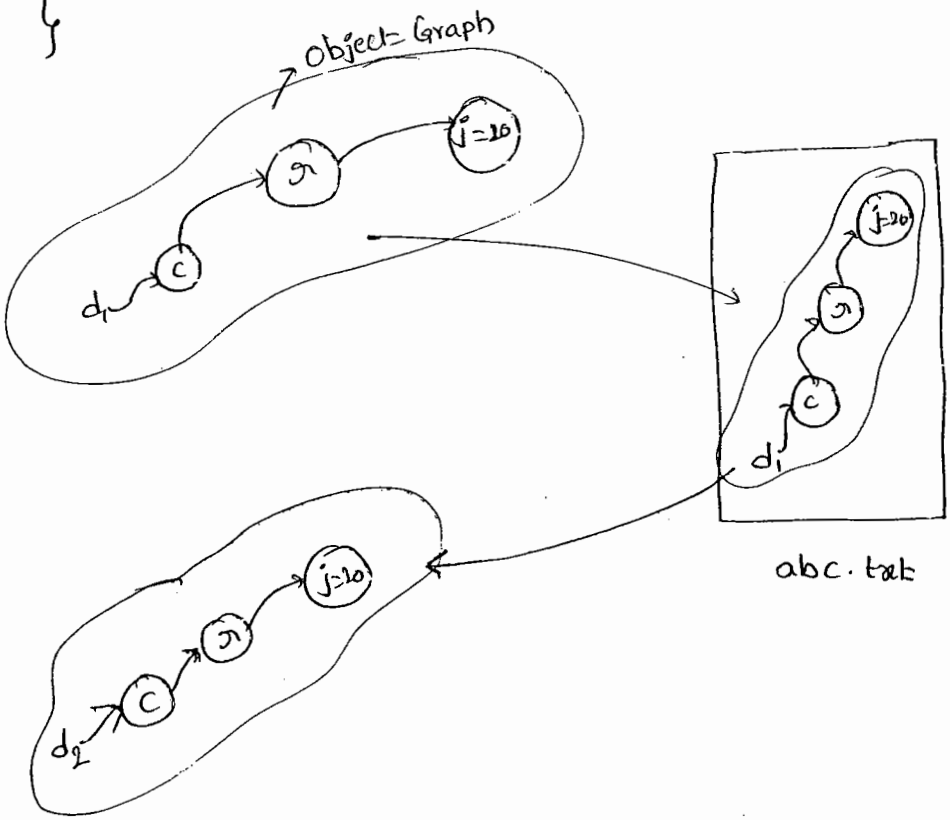
→ In Object Graph every object should be Serializable otherwise we will get "Not-Serializable-Exception".

Ex:-

```
import java.io.*;
class Dog implements Serializable
{
  Cat c = new Cat();
}
class Cat implements Serializable
{
   Rat r = new Rat();
}
Class Rat implements Serializable
{
   int j = 20;
}
Class SerializeDemo2
{
  P.S.V.m(String[] args) throws Exception
  {
     Dog d = new Dog();
     FileOutputStream fos = new FileOutputStream("abc.ser");
     ObjectOutputStream oos = new ObjectOutputStream(fos);
     oos.writeObject(d);
```



```
FileInputStream fis = new FileInputStream("abc.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
Dog d1 = (Dog)ois.readObject();
S.o.pln(d1.c.r.j); //20
}
}
```



→ In the above prog. when ever we are Serializing a Dog object automatically Cat & Rat objects will be Serialized. because these are the part of object graph of dog.

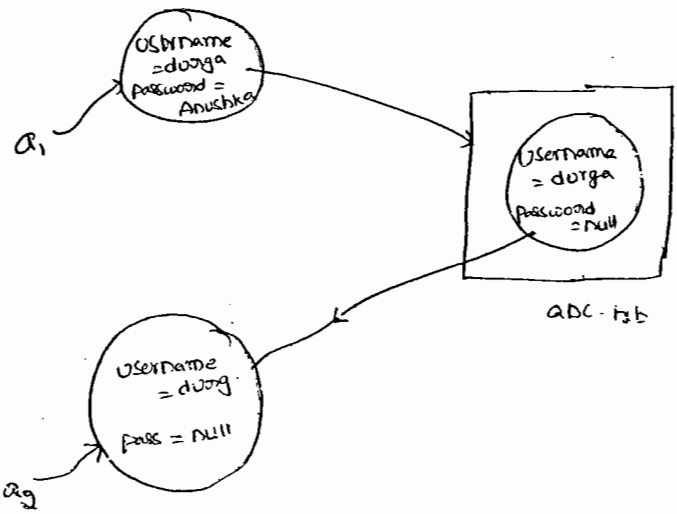
→ Among dog, Cat & Rat if atleast one class is not Serializable Then we will get NotSerializableException.

Customized Serialization :-

→ In the default Serialization there may be a chance of loss of information because of transient Keyword.

Ex:-

```
class Account implements Serializable
{
  String username = "durga";
transient String password = "anushka";
}
Class SerializeDemo3
{
 P.S.V.m(String[] args) throws Exception
 {
   Account a1 = new Account();
   S.o.println(a1.username + "-----" a.password);   durga ... anushka
   FileOutputStream fos = new FileOutputStream("abc.ser")
   ObjectOutputStream oos = new ObjectOutputStream(fos);
   oos.writeObject(a1);
   FileInputStream fis = new FileInputStream("abc.ser");
   ObjectInputStream ois = new ObjectInputStream(fis);
   Account a2 = (Account)readObject();
   S.o.println(a2.username + "-----" + a2.password);
 }
}
```



→ In the above Example before Serialization Account Object Can provide proper username & password But after deSerialization Account Object Can't provide the Original password. Hence during default Serialization there may be a chance of loss of information due to transient Key word. We Can recover this loss of information by using Customized Serialization

→ We Can implement Customized Serialization by using the following 2 methods

(1) private void writeObject(OOS os) throws Exception

→ this method will be Executed automatically at the time of Serialization it is a Call back method.

(2) private void readObject(OIS is) throws Exception

→ this method will be Executed automatically at the time of deserialization it is a Callback method.

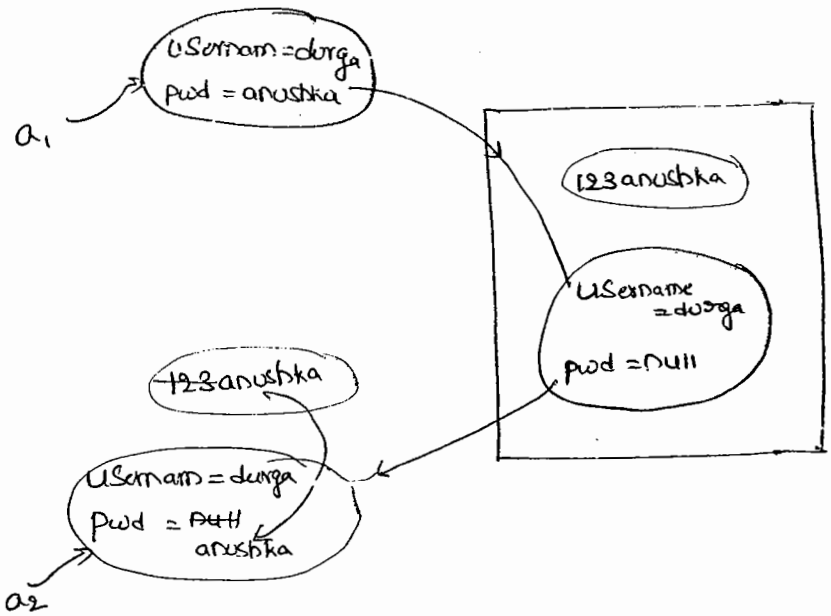
→ The above 2 methods we have to define in the Corresponding Class of Serialized Object.

ex:-

```
import java.io.*;
class Account implements Serializable
{
  String username = "durga";
  transient String pwd = "anushka";
  private void writeObject(ObjectOutputStream os) throws Exception
  {
    os.defaultWriteObject();
    String epwd = (String)is.readObject();
    pwd = epwd.substring(3);
  }
}
class CustSerializeDemo1
{
  P.S.V.m(String[] args) throws Exception
  {
    Account a1 = new Account();
    S.o.pln(a1.username + "----" + a1.pwd);
    FileOutputStream fos = new FileOutputStream("abc.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(a1);
    FileInputStream fis = new FileInputStream("abc.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    Account a2 = (Account)ois.readObject();
```

```
        S.O.println(a2.username + "----" + a2.pwd);
    }
}
```

~~Serialization w.r.t Inheritance~~ :-



Serialization w.r.t Inheritance :-

Case 1:-

→ If the parent class implements Serializable then every child class is by default Serializable. i.e, Serializable nature is inheriting from parent to child (P → C).

ex:-
```
class Animal implements Serializable
{
    int x=10;
}
class Dog extends Animal
{
    int y=20;
}
```

→ we can Serialize dog object eventhough dog class doesn't implement Serializable interface explicitly. because it's parent class Animal is Serializable.

Case 2:-

→ Eventhough parent class doesn't implement Serializable & if the child is Serializable then we can Serialize child class object. At the time of Serialization JVM ignores the original values of instance variables which are coming from Non-Serializable parent & store default values.

→ At the time of deserialization JVM checkes is any parent class is non-Serializable or not, JVM creates a seperate object for every non-Serializable parent & share its instance variables to the current object.

→ for this JVM always calls no arguement Constructor of the non-Serializable parent. if the non-Serializable parent doesn't have no arrguement Constructor then we will get RuntimeException.

Ex:-

```
import java.io.*;
class Animal
{
  int i=10;
   Animal()
    {
      S.o.println("Animal Constructor Called");
} }
```

```
class Dog extends Animal implements Serializable
{
    int j=20;
    Dog()
    {
        S.o.pln("Dog Constructor Called");
    }
}
class SerializeDemo6
{
    p.s.v.m(String[] args)throws Exception
    {
        Dog d = new Dog();
        d.i = 888;
        d.j = 999;
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d);
        S.o.pln("Deserialization Started");
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d1 = (Dog)ois.readObject();
        S.o.pln(d1.i+"---"+d1.j);
    }
}
```

O/P :- Animal Constructor Called

Dog " "

Deserialization started

* Animal Constructor Called

10 ---- 999