14/02/10

Exception Handling

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

Exception :-

→ when unwanted, unexpected Event that disterbes normal flow of program is called "Exception".

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

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

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

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

Syn:-
```
Try
{
  read data from London file
}
catch(FileNotFoundException e)
{
  use local file and continue rest of the program-
  normally.
}
```

Runtime Stack mechanism :-

→ For Every thread JVM will Create a RuntimeStack.

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

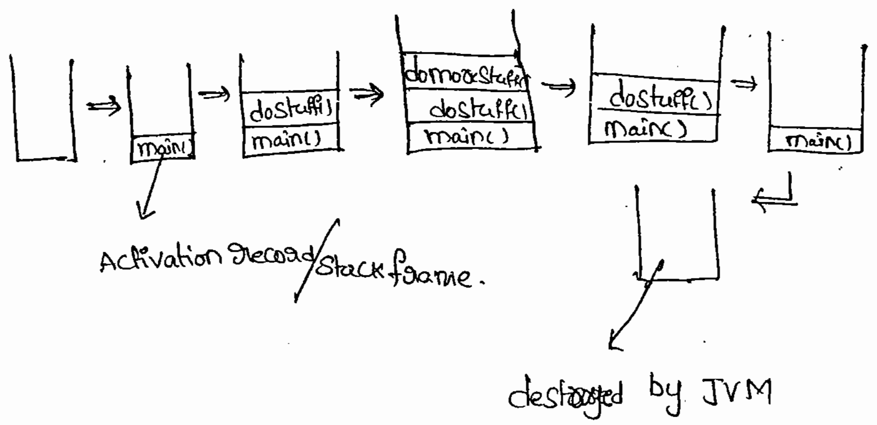
→ Each Entry in the Stack is Called "Activation Record" or "Stack frame."

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

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

Ex:-

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



default Exception handling in Java :-

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

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

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

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

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

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

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

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

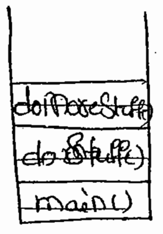
→ Just before terminating the program abnormally JVM handovers the responsibility of Exception handling to the default Exception handler.

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

Name of Exception : Description
Location (Stack Trace)

15/02/11 :-

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



Runtime Stack

name of exception

description

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

Stack Trace.

Exception hierarchy:-

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

It has the following 2 child classes

1. Exception
2. Error

1. Exception:-

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

2. Error:-

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

→ Errors are NON-Recoverable.

Checked Vs un-checked Exceptions?

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

Ex!- HallTicketMissing Exception,
PenNotWorking Exception,
FileNotFound Exception.

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

Ex!- BombBlast Exception.
ArithematicException, FireExcidentException.

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

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

Partially checked vs fully checked :-

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

Ex:- IOException

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

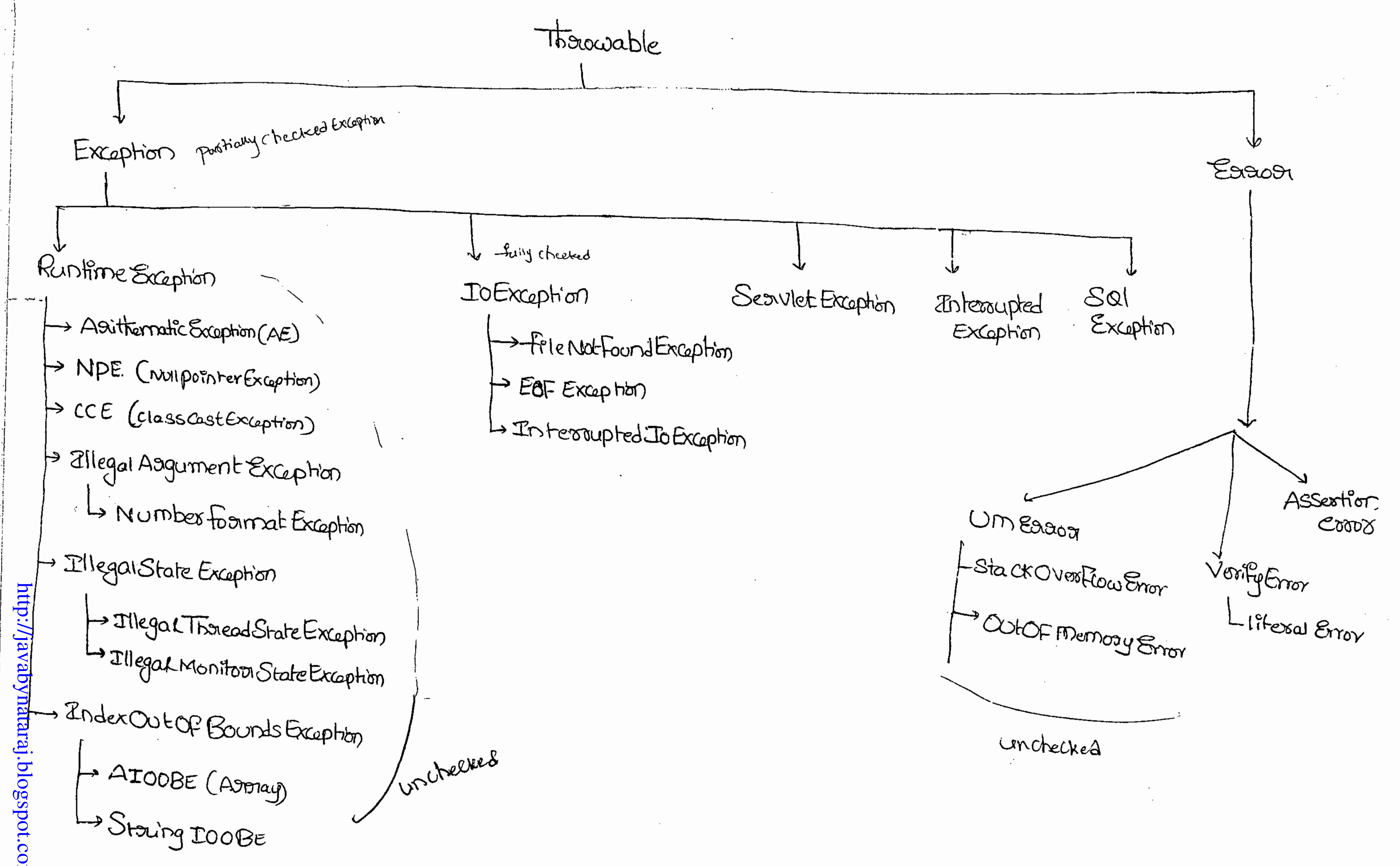
Ex:- Exception.

Q
(1) which of the following are checked

1) 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.


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

Customized Exception Handling by Try-Catch:-

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

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

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

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

Abnormal termination

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

o/p:- State1
      5
      State3
      normal termination

Control flow in Try-Catch :-

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

Case1 :-

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

Case2 :-

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

Case3 :-

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

Case4 :-

→ If an Exception raised at Statement4 or Statement 5 it is always A.NT Abnormal Termination

Note :-

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

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

Various Methods to print Exception information :-

16/02/11

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

(1) printStackTrace() :-

→ this method prints Exception information in the following formatt.

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

(2) toString() :-

→ It prints Exception information in the following formatt.

$\boxed{\text{Name of Exception : discription}}$

(3) getMessage() :-

→ this method prints only discription of the Exception.

$\boxed{\text{discription}}$

Ex:-

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

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

Try with Multiple Catch blocks :-

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

Ex:-

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

Ex(Q):-

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

✓ Highly recommended

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

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

→ If we are taking from parent to child then we will get Compile time Error Saying, "Exception xxxxx has already been Caught"

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

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

X

```
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 block. because there is no garenty for the Execution of every statement.

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

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

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

Ex1:-

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

Ex(2):-

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

O/P:- try
finally

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

O/P:- Try
catch
finally

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

o/p:- try
finally
Abnormal

return vs finally:-

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

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

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

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

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

difference b/w final, finally & finalize :-

final :-

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

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

→ If a method declared as final, then 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(yy e)
{       parent
}
```

③ ✓
```
try
{
}
finally
{
}
```

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

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

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

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

⑧ ✓
```
try
{
}
catch(xx e)
{
}
S.op ln("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
{
}
finally
{
}
```
C.E:- finally without try

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

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

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

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

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

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

⑰
```
try
{
}
finally
{
}
catch(x e)
{
}
```
C.E:- catch without try

Control-flow in try-catch-finally :-

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

Case 1 :-

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

Case 2 :-

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

Case 3 :-

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

Case 4 :-

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

Case 5 :-

→ If an Exception raised at Stat 5 or Stat 6, it is always abnormal terminati

Control flow in Nested try-catch-finally :-

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

Case 1:-

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

Case 2:-

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

Case 3:-

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

Case 4:-

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

Case 5:-

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

Case 6:-

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

Case 7:-

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

Case 8:-

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

Case 9 :-

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

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

Case 10:-

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

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

Case 11:-

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

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

Case 12:-

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

Case 13:-

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

Case 14:-

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

18/02/11

Throw :-

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

throw new AsithematicException("/ by zero");

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

↓ Creation of A.E object Explicitly

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

→ The Result of following two programs is Exactly Same.

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

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

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

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

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

Case 1:-

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

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

RE:- NPE

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

R.E: A.E

Case 2:-

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

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

R.E!- AE / by Zero

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

C.E!- unreachable Statement.

Case 3 :-

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

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

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

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

R.E:
Exception in thread
main : Test.

Throws :-

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

Ex :-

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

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

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

(1) By using Try-catch

(2) " " throws

(1) By using Try-catch:-

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

✓

(2) By using throws keyword:-

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

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

✓

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

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

Ex:

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

✓

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

18/10/11

We Can use throws Keyword Only for Throwable types otherwise we will get Compile type Error Saying, inCompatable types

```
class Test                          Class Test extends ~~Throwable~~ Exception
{                                   {
X p.v.m1() throws test              p.v.m1() throws Test
  {                                   {
  }                                   }
}                                   }
                                              ✓
C.E:- inCompatable type
      found : Test
      Required : java.lang.throwable
```

Case(1) :-

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

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

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

→ As Exception is checked Compulsory we should handle either by Try-catch or by throws Keyword

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

Case 2!:

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

Ex!:

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

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

```
try          ✗
{
S.o.pln("Hello");
}
Catch(IOException e)
{
}
C.E!- Exception java.lang.IOException is
never thrown in body of corresponding
try statement.
```

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

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

Keywords for Exception!:

try
catch
finally
throw
throws

Exception Handling Keywords Summary :-

1) try :- To maintain Risky code.

2) catch :- To maintain Handling Code.

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

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

5) throws :- To delegate the Responsibility

Various Possible Compiletime Error in Exception Handling:-

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

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

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

④ try without Catch or finally

⑤ finally without try

⑥ Catch without try

⑦ unreachable Statement

⑧ Incompatable types

found : Test

required : Java.lang.Throwable.

Customized Exceptions:

→ To meet our programming requirement some times we have to create our own Exceptions. Such types of Exceptions are called "Customized Exceptions".

Eg: ~~TooyoungException~~, ~~TooOld Exception~~, ~~InSufficient fundsExceptions~~..etc

```
class TooYoungException extends RuntimeException
{
  TooYoungException(String s)
  {
   Super(s);
  }
}
class TooOldException extends RuntimeException
{
  TooOldException(String s)
  {
   Super(s);
  }
}
class Test
{
p.s.v.m(String[] args)
{
 int age=Integer.parseInt(args[0]);
 if(age >60)
 {
 throw new TooYoungException("plz wait some more time")
                    age is already crossed marriage age");
 }
 else if(age <18)
 {
 throw new TooYoungException("Ur age is already crossed marriage
                    age
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 Runtime Exception Class but not Exception Class while defining our customized Exceptions.

Top-10 Exceptions :- 21-02-11

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

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

1. JVM Exceptions :-

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

Ex:- (i) ArrayIndexOutOfBoundsException.
(ii) NullPointerException . . . . .

2. programatic Exceptions :-

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

Ex:- IllegalArgumentException, NumberFormatException . . .

(1) AarrayIndexOutOfBoundsException :-

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

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

Ex:- int [] a = new int[10];

S.o.pln(a[0]); 0 ✓

S.o.pln(a[100]); R.E:- AIOOBE

(2) NullPointerException :-

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

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

Ex:- String s = null;

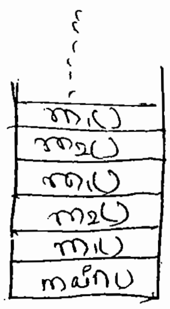
S.o.p(s.length()); RE: NPE

(3) StackOverFlowError :

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

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

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



R.E: SOFE

(4) NoClassDefFoundError :-

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

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

Ex: Java Srinu ↵

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

(5) ClassCastException :-

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

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

Ex):-

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

~~Stren~~: Object o = new Object();  | X
           String s = (String) o;    |

           R.E :- CCE
```

(6) ExceptionInInitializerError :-

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

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

Eg:-

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

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

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

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

⑦ Illegal Argument Exception :-

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

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

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

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

⑨ IllegalStateException :-

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

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

Ex:-

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

Ex①:-

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

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

Ex②:-

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

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

10) AssertionError:-

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

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

Ex:- Assert(false);

R.E:- AssertionError.

Exception/error	Raised by
1. AIOOBE	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. NumberformattException	
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