14/02/10

Exception Handling

Exception :-

→ When unwanted, unExpected Event that disterbes normal flow of program is Called "Exception".

   Ex:- Sleeping Exception, Type punchased Exception, file not found - Exception.

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

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

Ex:- If our programing requirement is to read data from the file locating at 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 (file not found Exception e)
         {
            use local file and Continue rest of the program normally.
         }

Runtime Stack mechanism :-
——x——x——x——

→ 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.
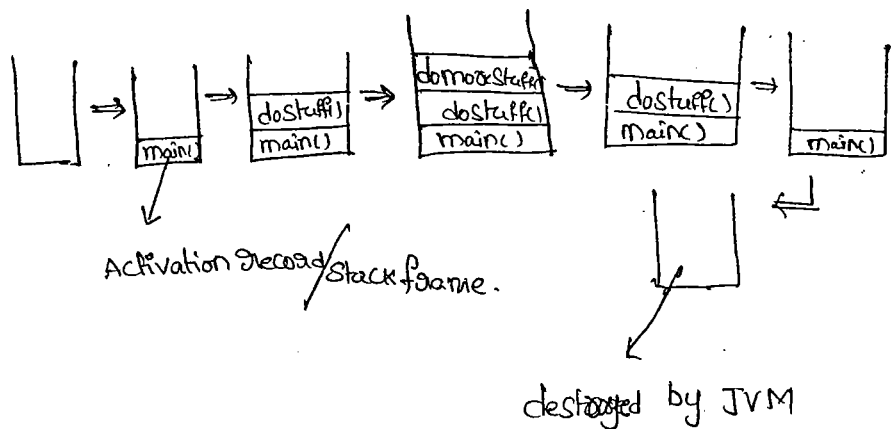
Ep!-

```
Class Test
{
P.S.V.m(String args[])
{
 dostuff();
}
P.S.V.dostuff()
{
 do moreStuff();
}
P.S.V.domorestuff()
{
S.o.pln(" don't Sleep");
}
```


Activation record/Stack frame.

destroyed by JVM

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

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

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

→ Default Exception handler Just print Exception information to the console in the following format.

```
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.p in (10/0);
}
}
```

doMoreStuff
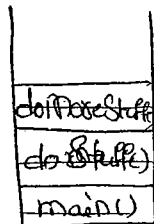doStuff()
main()

Runtime Stack

name of exception   description

Exception in thread "main" : Java.lang.AE : / by Zero
        at Test.doMoreStuff()
        at Test.doStuff()   } Stack Trace.
        at Test.main()

Exception hierarchy:-

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

    1. Exception

    2. Error

1. Exception :-

→ most of the Cases Exceptions are Caused by our program & These are Recoverable.

2. Error :-

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

→ Errors are NON-Recoverable.

Checked Vs un-checked Exceptions?

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

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

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

    Ep!- BombBlast Exception.
        Aarthematic Exception, FireExcident Exception.

→ Wheather Exception is checked or unchecked Componssorily it should runtime only. There is no chance of occouring at Compile time.

→ Runtime Exception 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.

Ex¹:-   IOException

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

Ex!-   Exception.

Q
(1) which of the following are checked
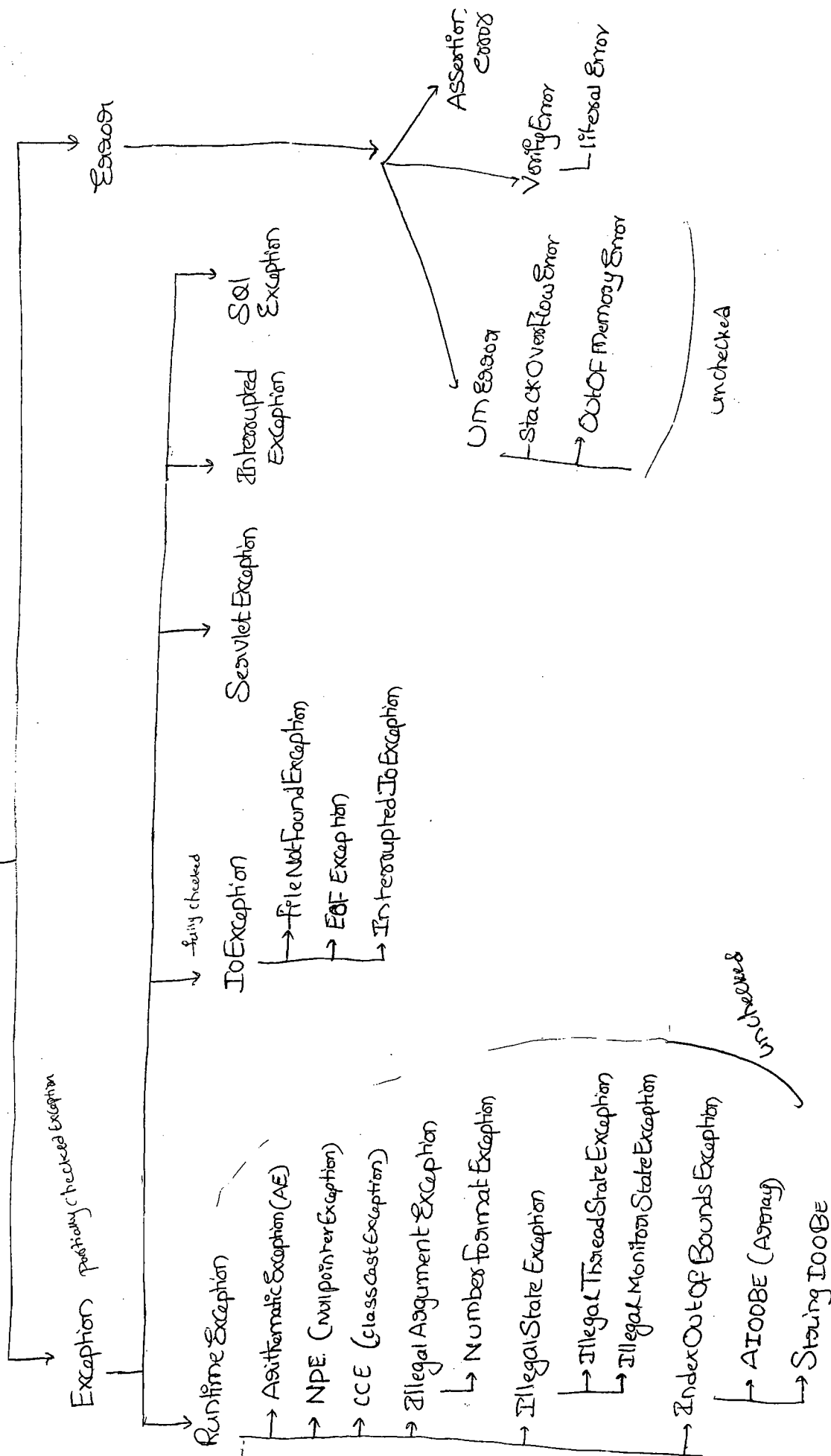
    i) IOException : fully checked

    2) Error   : unchecked

    3) Throwable   : partially checked

    4). Nullpoint Exception : unchecked

    5) InterruptedIOException : fully checked

    6) SQLException   : fully checked.

Note!

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

Throwable

```
Throwable
   │
   ├──► Error
   │      │
   │      ├──► AssertionError
   │      │
   │      ├──► VerifyError
   │      │      └─ LiteralError
   │      │
   │      └──► VM Error
   │             ├─ StackOverFlowError
   │             └─ OutOfMemoryError          } unchecked
   │
   └──► Exception  partially checked exception
          │
          ├──► RuntimeException
          │      ├─ ArithematicException (AE)
          │      ├─ NPE (NullPointerException)
          │      ├─ CCE (ClassCastException)
          │      ├─ Illegal Argument Exception
          │      │      └─ NumberFormatException
          │      ├─ IllegalState Exception
          │      │      ├─ IllegalThreadStateException
          │      │      └─ IllegalMonitorStateException
          │      └─ IndexOutOfBoundsException
          │             ├─ AIOOBE (Array)
          │             └─ String IOOBE        } unchecked
          │
          ├──► IOException   ↓ fully checked
          │      ├─ FileNotFoundException
          │      ├─ EOF Exception
          │      └─ InterruptedIOException
          │
          ├──► ServletException
          │
          ├──► InterruptedException
          │
          └──► SQL Exception
```

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 (xxxx  e)
{
    handling code.
}
```

```
Class Test
{
P.S.v.m (String [] args)
{
    S.o.pln ("State1");
    S.o.pln (10/0);
    S.o.pln ("State 3");
}
}
O/P:- State1
R.E: A.E : 1 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 ("State 3");
    }
}
O/P:-  State1
        5
      State 3
    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, I 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 point Exception Information :-   16/02/11

→ Throwable Class defines the following methods to point Exception information.

(1) PointStackTrace() :-

→ This method points Exception information in the following formatt.

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

(2) toString() :-

→ It points Exception information in the following formatt.

> Name of Exception : discription

(3) getMessage() :-

→ This method points only discription of the Exception.

> discription

Ex!-

```
Class Test
{
  P.S.V.m (String [] args)
  {
    try
    {
      S.opln(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 Exception handler 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(0):-   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 fallows   │
└─────────────────────────────────┘

```
try                          try
{                            {
  ―                            ―
  ―                            ―
}                            }
    Catch (exception e)          Catch (A.E e)      ✓
    {                            {
      ―                            ―
    }                            }
X
    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 block, because there is no gaurenty from 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):-

Column 1:
```
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

Column 2:
```
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

Column 3:
```
Class Test
{
    P.S.V.M (String [] args)
    {
        try
        {
            S.o.pln("try");
            S.o.pln(10/0);
        }
        catch (NullpointerEr 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(o)

(✱) Ex!:-  class Test
```
{
    p.s.v.m (String [] args)
    {
        tay
        {
            S.op ln ("tay");
            System. exit(0);
        }
        catch (AE e)
        {
            System. out. println ("catch");
        }
        finally
        {
            S.o.pln (" finally"),
        }
    }
}
```

o/p !:-    tay

---

✱ 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 Garbage Collector before destroying any object to perform clean-up activities.

## Note :-

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

## <u>Various possible Combinations of try-catch-finally :-</u>

① try  
{  
}  
Catch (xx e)  
{  
}  
✓

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

③ try  
{  
}  
finally  
{  
}  
✓

④ try  
{  
}  
✗  
C-E :-  
Try with out Catch  
on finally

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

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

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

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

⑨
```
try
{
}
Catch(xx e)
{
}
   S.o.pln("Hello");
x| finally
{
}
```
C.E!- finally without try

⑩
```
try
{
}
Catch(xx e)
{
}
   finally
{
}
x| 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)
   {
   }
}
```

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

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

# Control-flow in try-catch-finally :-

```
try
{
    State 1;
    State 2;
    State 3;
}
Catch (xxx e)
{
    State 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 State5 or State6, it is always abnormal termination

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.

throw :-

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

$$throw \quad new \quad ArithematicException("/ \; by \; zero");$$

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

↓ Creation of A.E object Explicitly

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

→ The Result of following two programs is Exactly Same.

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

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

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

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

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

Case1:-

→ 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;
  }
}
```
R.E: NPE

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

Case2:-

→ After throw Statement we are not allow to write any statement directly otherwise we will get 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 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 Commpulsary we should handle it, Other wise we will get Compiletime Error Says " unreported Exception xxxxx must be Caught on 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 methods 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 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 {
 X p.v.m() throws test
 }
 }
 }
	class Test extends ~~Throwable~~ Exception {
	 p.v.m,() throws Test
	 }
	 }
	 }

C·E!- inCompatable type
found : Test
Required : java.lang. throwable

Case(1) :-

class Test   (Checked)
 {
    P·S·v·m (String []args)
    {
      throw new Exception();
    }
 }
	class Test (unchecked)
	 {
	   P·S·v·m (String []args)
	   {
	   throw new Error()
	   }
	 }

Left column:

CE: unrepoated Exuption java.long.
Exception must be Caught at declared
to be thrown.

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

Right column:

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 nor~~ 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.

Ep:

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

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

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

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

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

```
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 Errors in Exception Handling :-

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

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

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

④ try without Catch or finally

⑤ finally without try

⑥ Catch without try

⑦ Unreachable Statement

⑧ Incompatable types

> found : Test
>
> Required : Java.lang.Throwable.

Customized Exceptions :

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

Ep! TooYoungException, TooOld Exception, InSufficient Founds Exceptions..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(aags[0]);
  if (age >60)
  {
  throw new TooYoungException (" plz wait Some more time" ) n
                            age is already Crossed marrage age")
  }
  else if (age <18)
  {
  throw new TooYoungException (" ur age is already crossed marrage
                        age.... no chance of getting marri")n
```

```
    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 Klass but not Exception class while defining our customized Exceptions.

Top-10 Exceptions :-   21-02-11

→ Based on the Source, who triggeres 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) ArrayIndex OutofBounds Exception.

    (ii) NullPointerException. ----

2. programmatic Exceptions :-

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

Ex:- IllegalArguement Exception, NumberformatException. ..

(1) Array.IndexOutOfBoundsException:

→ 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.

Eg:-
```
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.

Eg:-
```
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.

Eg:-
```
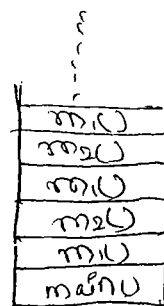Class Test
{
  p.s.v.m m1()
  {
    m2();
  }
  p.s.v.m m2()
  {
    m1();
```

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

m1()
m2()
m1()
m2()
m1()
main()

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.new

— 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;

~~Stath~~ 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.

Ex:-

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

R.E:-

ExceptionInInitializedError

Caused by java.lang.AE :/ by Zero.

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

R.E:- ExceptionInInitializedError

Caused by java.lang. NPE

---

⑦ Illegal Assignment Exception :

→ PE 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 assignment

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

⑧ NumberFormat Exception

→ PE 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

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

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.

Ex:-

ONG 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---   ✓
```

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

Ex②:-
```
Thread t = new Thread();
  t.start(); ✓
      ⋮
      ⋮

X 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) Assertion Error :

→ 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 1 assert statement fails.

Ep)-   Assert (false);

        R·E': Assertion Error.

Exception/Error	Raised by
1. AIOOBE	
2. NPE	
3. SOFE	JVM automatically
4. NoClassDefFoundError	(JVM Exception)
5. ClassCostException	
6. ExceptionInInitializerError	
7. IllegalArguementException	
8. NumberformatException	either programmer or API developer Explicitly
9. IllegalStateException	(programatic Exceptions)

Exception Propagation:-

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