

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. Throws.
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 disturbs normal flow of program is called "Exception".

Ex:- Sleeping Exception, TypedException, FileNotFoundException - 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 methodCalls, Just before Terminating The Thread JVM destroyed the Stack.

Ex:-

Class Test

↓
P.S.V.m(String args[])

↓
doStuff();

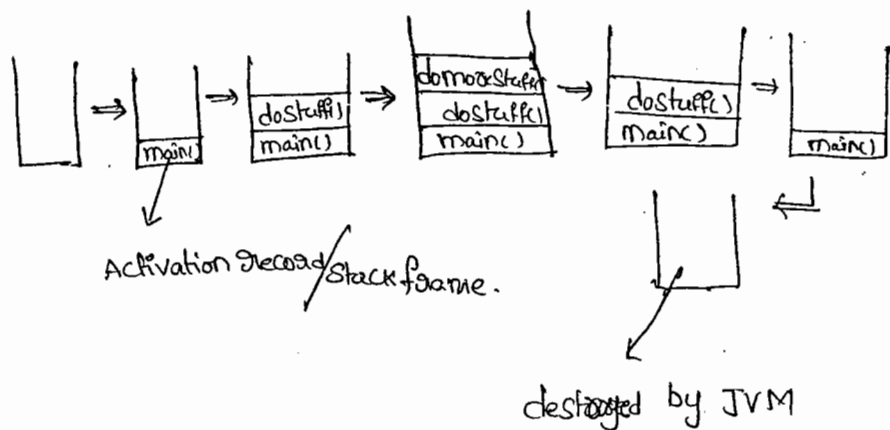
{
P.S.V.doStuff()

↓
doMoreStuff();

{
P.S.V.doMoreStuff()

↓
S.o.plnc("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. description of Exception.
 3. location of Exception (Stack trace)
- After creating Exception object, method hands over that Exception object to the JVM.
- JVM checks whether 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 whether 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 until `main()` & if the `main()` ^{also} doesn't contain handling code, JVM terminates the `main()` also abnormally & removes corresponding entry from stack.

- 253
- Just before terminating the program abnormally JVM hands over the responsibility of Exception handling to the default Exception handler.
 - Default Exception handler just prints 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();

}

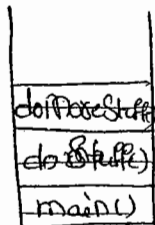
P.S.V.doMoreStuff()

{

S.o.p in (10/0);

}

}



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 hierarchy.

It has the following 2 child classes

1. Exception

2. Error

1. Exception:-

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

These are Recoverable.

2. Error:-

→ most of the cases Errors are not caused by our program

These are due to lack of system resources.

→ Errors are NON-Recoverable.

Checked vs UN-checked Exceptions?

→ The Exceptions which are checked by Compiler for smooth execution of the program at Runtime are called 'checked Exception'.

Ex:- HallTicketMissingException,
PenNotWorkingException,
FileNotFoundException.

→ The Exceptions which are not checked by Compiler are called 'un-checked Exceptions'.

Ex:- BombBlastException,
ArithmeticException, FireExidentException.

254

→ Whether Exception is checked or unchecked ~~Component~~ only it should runtime only. There is no chance of occurring at Compile time.

→ Runtime Exception and it's child classes

→ Error & 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.

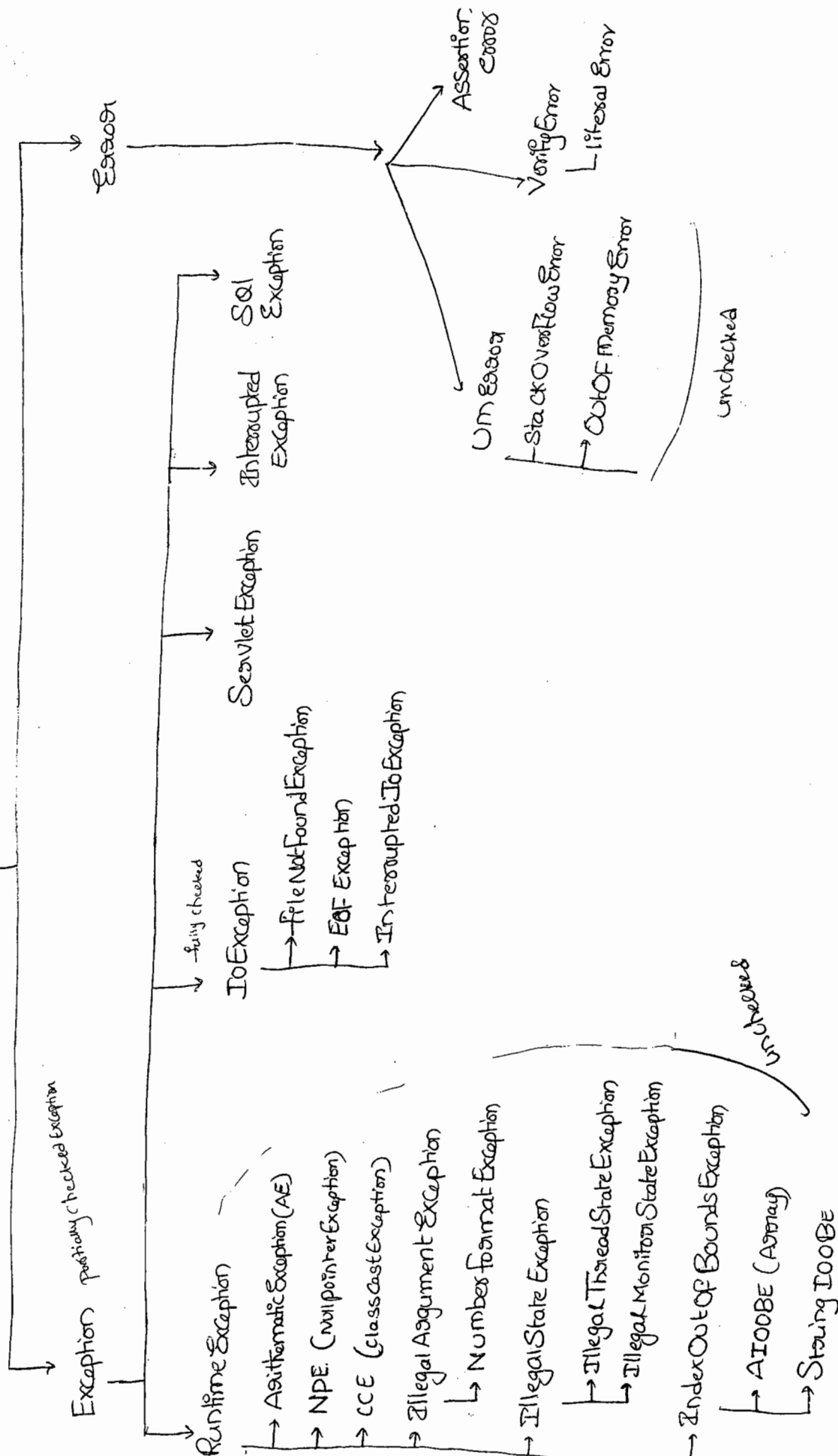
Q (a) which of the following are checked

- 1) IOException : fully checked
- 2) Error : unchecked
- 3) Throwable : partially checked
- 4) NullPointerException : unchecked
- 5) InterruptedException : fully checked
- 6) SQLException : fully checked.

Note!

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

Throwable



Customized Exception Handling by Try-Catch:-

255

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

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

Class Test

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

```
        S.o.pln("State1");
```

```
        S.o.pln(10/0);
```

```
        S.o.pln("State3");
```

```
    }
```

o/p - State1

R.E : A.E : 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("State3");
```

```
    }
```

o/p - State1

5

State3

Normal termination

Control flow in Try-catch :-

```
try
{
    Stat1;
    Stat2;
    Stat3;
}
catch(xxx e)
{
    Stat4;
}
Stat5;
```

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.N.T ^{Abnormal Termination}

Note:-

→ With in the Try block if anywhere 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 ²⁵⁶
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 : description follow by
Stack trace

(2) toString() :-

→ It prints Exception information in the following format.

Name of Exception : description

(3) getMessage() :-

→ This method prints only description of the Exception.

description

Ex:-

```
class Test
{
    p.s.v.m(String [] args)
    {
        try
        {
            S.opln(10/0);
        }
        catch(A.E e)
        {
            e.printStackTrace();
            S.op(e); (or) S.opln(e.toString());
            S.opln(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 separate catch block.

Ex:-

```
try
{
    ...
}
catch(Exception e)
{
    ...
}
```

(but not recommended.)

```

Ex(8):- try
{
    //
}
catch (Ae e)
{
    Perform these Arithmetic operations;
}
catch (FileNotFoundException e)
{
    Use local file;
}
catch (Npe e)
{
    Use Another resource
}
catch (Exception e)
{
    default Exception handle;
}
    
```

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)
{
    //
}

```

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 within the ^{try} block because there is no guarantee for the execution of every statement.
- It is never recommended to define Clean-up code within the catch-block, because it won't be executed if there is no exception.
- We required a place to maintain clean-up code which should be executed always irrespective of whether exception raised or not raised & whether handle or not handle, such type of place is nothing but finally-block.
- Hence, the main purpose of finally-block is to maintain clean-up code which should be executed always.

Ex1:-

```

try
{
    Risky Code;
}
catch (xxx e)
{
    handling Code;
}
finally
{
    Clean-up Code;
}
    
```

Ex2:-

Class Test

```

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

o/p:- try
finally

Class Test

```

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

o/p:- Try
Catch
finally

Class Test

```

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

o/p:- try
finally

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 c)
        {
            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()

(*)

```

Ex:- class Test
{
    p.s.v.m(String l1arg&)
    {
        tag
        {
            S.opln("tag");
            System.exit(0);
        }
        catch(AE e)
        {
            System.out.println("catch");
        }
        finally
        {
            S.opln("finally");
        }
    }
}
o/p:- tag

```

*) Difference b/w final, finally & finalize :-

→ final :-

- It is a modifier applicable for classes, methods & 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 ^(changing the value) reassignment 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 whether exception raised or not raised & whether handled or not handled.

finalize() :-

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

Note:-

→ When Compare with finalize(), it is highly recommended to use finally block to maintain clean-up code. Because, we can't expect exact behaviour of the Garbage Collector.

Various possible Combinations of try-catch-finally :-

① try ✓
{
}
catch(xxx e)
{
}

② try ✓
{
}
catch(xxx e) child
{
}
catch(yyy e) parent
{
}

③ try ✓
{
}
finally
{
}

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

⑤ X
catch(xxx e)
{
}
C.E:-
Catch with
out Try

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

⑦ try X
{
S.opIn("Hello");
catch(xxx e)
{
}

C.E:- Try without catch or finally
C.E:- catch without try

⑧ try ✓
{
}
catch(xxx e)
{
}
S.opIn("Hello");

X | Catch(xxx e) C.E:- catch with out try

```

(9) try
    {
    }
    catch (xx e)
    {
    }
    S.opln("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 (xx e)
    {
    }
  
```

```

(15) try
    {
    }
    catch (xx e)
    {
    }
    finally
    {
    }
    try
    {
    }
    catch (xx 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
{
    State 1;
    State 2;
    State 3;
}
catch (xx 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 Statement 5 or Statement 6, it is always abnormal termination.

Control flow in Nested try-catch-finally :-

261

```
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 Statement 5 & inner & outer Catch blocks are not matched then 1, 2, 3, 4, 8, 11, Abnormal

Case 7:-

→ If an Exception raised at Statement 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 :-

throw new ArithmeticException(" / by zero ");

Creation of A.E object Explicitly

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

class Test

class Test

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

<http://javabynataraj.blogspot.com> 267 of 401.

→ In General, we can use throw keyword for customized Exceptions 263

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

```
class 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 allowed 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, otherwise we'll get compiletime Error says "unreported Exception 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.InterruptedException 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 Convince Compiler.

→ In the case of unchecked Exceptions, it is not required to use throws keyword.

```
Eg1 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.

18/10/11

265

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

```

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

```

CE:- incompatible 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();
    }
}

```

CE:- 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();
    }
}

```

RE:- Exception in thread "main"
java.lang.Error.

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

Case 2!

→ In our program, if there is no chance of raising an Exception then, ~~it is not~~ we can't define Catch blocks 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)
{
}
//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)
{
}
//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 catches)
- 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

Sound : 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".

Ex: TooYoungException, TooOldException, InsufficientFundsException...etc

```
class TooYoungException extends RuntimeException
```

```
{
```

```
    TooYoungException(String s)
```

```
{
```

```
        super(s);
```

```
}
```

```
}
```

```
class TooOldException extends RuntimeException
```

```
{
```

```
    TooOldException(String s)
```

```
{
```

```
        super(s);
```

```
}
```

```
}
```

```
class Test
```

```
{
```

```
    public static void main(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("Or age is already crossed marriage  
age")
```



```

else
{
    S.o.println("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 divided into 2 types.

1. J.V.M Exceptions

2. programmatic Exceptions.

1. JVM Exceptions :-

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

Ex:- (i) ArrayIndexOutOfBoundsException.

(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:- 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]);` RE:- AIOOBE

② NullPointerException:-

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

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

Ex:- `String s = null;`

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

③ StackOverflowError:-

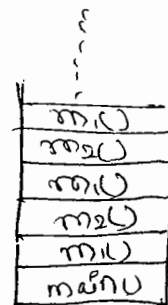
→ It is the child class of Error and hence it is unchecked.

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

Ex:-

```
Class Test
├── p.s.v.m m1()
│   ├── m2();
│   └── p.s.v.m m2()
│       └── m1();
```

```
p.s.v.m (String[] args)
├── m1();
└── }
```



(4) NoClassDefFoundError :-

269

- 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 trying to typeCast parent object to the child type.

Ex:-

String s = new String("duaga");
✓ Object o = (Object) s;

~~String~~ 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 ^{while} executing static blocks.

Ex:-

Class Test

↓

Static int i = 10/0;

↓

R.E:-

ExceptionInInitializerError

Caused by java.lang.ArithmeticException: / by zero.

Class Test

↓

Static

↓

String s = null;

S.opln(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

Ex: ✓ int i = Integer.parseInt("0");

✗ int i = Integer.parseInt("ten");

RE
↑
IAE
↑
NPE

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 inappropriate time.

Ex:-

Once Session Expires we can't call any method on that object otherwise we will get IllegalStateException.

Ex ①:-

```
HttpSession session = req.getSession();
s.o.println(session.getId()); 12345678 ✓
```

X

```
session.invalidate();
s.o.println(session.getId()); R.E:- ISE
```

Ex ②:-

```
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. AIOBE	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 Responsibility Exception handling from one method to another method by using throws keyword is called Exception propagation