Lesson:

Exception Handling

PW SKILLS

List of Concepts Involved:

- Different types of Errors in Java
- What is an Exception?
- try catch
- Multiple catch block
- Handling vs Ducking an Exception
- Rethrowing an Exception(throw, throws,finally) and Custom Exception
- Hierarchy of an Exception class
- Control flow of Exception Handling concept
- try with Resources

Different types of Errors in Java
In any programming language we categorise errors into 2 types

1. Syntax Error/CompileTime Mistakes
2. Logical Error/RunTimeMistakes

Syntax error/CompileTime Mistakes
- It refers to the mistakes done by the programmer with respect to syntax.
- These mistakes are identified by the compiler, so we say it as“CompileTimeMistake”.

Logical Error/RunTimeMistakes
- It refers to the mistakes done by the programmer in terms of writing a logic
- These mistakes are identified by jvm during the execution of a program, so we say it as “RunTimeMistake”.

What is an Exception?
- An unwanted/expected event that disturbs the normal flow of execution of a program is called "Exception handling".
- The main objective of Exception handling is to handle the exception.
- It is available for graceful termination of program.

What is the meaning of Exception handling?
- Exception handling means not repairing the exception.
- We have to define alternative ways to continue the rest of the program normally. This way of defining an alternative is nothing but **"Exception handling".**

Example
Suppose our programming requirement is to read data from a file located at one location,At run time if the file is not available then our program should terminate successfully.

Solution: Provide the local file to terminate the program successfully,This way of defining alternatives is nothing but **"Exception handling".**

Example

```
try{
    read data from London file
 } catch(FileNotFoundException e){
     use local file and continue rest of the program normally
}
```

RunTimeStackMechansim

- For every thread in the Java language, jvm creates a separate stack at the time of Thread creation.
- All method calls performed by this thread will be stored in the stack. Every entry in the stack is called **"StackFrame/Activation Record".**
- main() => doStuff() => doMoreStuff()

Example

```
class Demo{
 public static void main(String[] args){
  doStuff();
 }
 public static void doStuff(){
  doMoreStuff();
 }
 public static void doMoreStuff(){
  System.out.println("hello");
 }
}
```

output:: Hello

Syntax of Exception handling

```
try{
 //risky code
}catch(Exception e){
 //handling logic
}
```

Default Exception handling

```
class Demo{
 public static void main(String[] args){
  System.out.println("Entering main");
  doStuff();
  System.out.println("Exiting main");
 }
 public static void doStuff(){
  System.out.println("Entering doStuff");
  doMoreStuff();
                System.out.println("Exiting doStuff");
 }
 public static void doMoreStuff(){
  System.out.println("Entering doMoreStuff");
  System.out.println(10/0);
  System.out.println("Exiting doMoreStuff");
 }
}
```

Output::

Entering main
Entering doStuff
Entering doMoreStuff
Exception in thread "main" java.lang.ArithmeticException: / by zero
at TestApp.doMoreStuff(TestApp.java:14)
at TestApp.doStuff(TestApp.java:9)
at TestApp.main(TestApp.java:4)

As noticed in the above example in the method called doMoreStuff(), an exception is raised.
When exception is raised inside any method, that method is responsible for creating the Exception object will the following details

Name of the exception::java.lang.ArithmeticException
Description of exception::/ by zero
location/stacktrace::

1. This Exception object will be handed over to jvm, now jvm will check whether the method has the handling code or not,if it is not available then that method will be abnormally terminated.
2. Since it is a method, it will propogate the exception object to caller method.
3. Now jvm will check whether the caller method is having the code of the caller method or not.
4. If it is not available, then that method will be abnormally terminated.
5. Similar way if the exception object is propagated to main(), jvm will check whether the main() is having a code for handling or not, if not then the exception object will be propagated to JVM by terminating the main().
6. JVM now will handover the exception object to "Default exception handler", the duty of "default exception handler" is to just print the exception object details in the following way

Exception in thread "main" java.lang.ArithmeticException:/ by zero
at TestApp.doMoreStuff
at TestApp.doStuff
at TestApp.main

Example

```
class Demo{
 public static void main(String[] args){
  System.out.println("Entering main");
  doStuff();
  System.out.println("Exiting main");
 }
 public static void doStuff(){
  System.out.println("Entering doStuff");
  doMoreStuff();
  System.out.println(10/0");
                System.out.println("Exiting doStuff");
 }
 public static void doMoreStuff(){
  System.out.println("Entering doMoreStuff");
  System.out.println("hello");
  System.out.println("Exiting doMoreStuff");
 }
}
```

Output::

Entering main
Entering doStuff
Entering doMoreStuff
Hello
Exiting doMoreStuff
Exception in thread "main" java.lang.ArithmeticException : / by zero
at TestApp.doStuff()
TestApp.main()

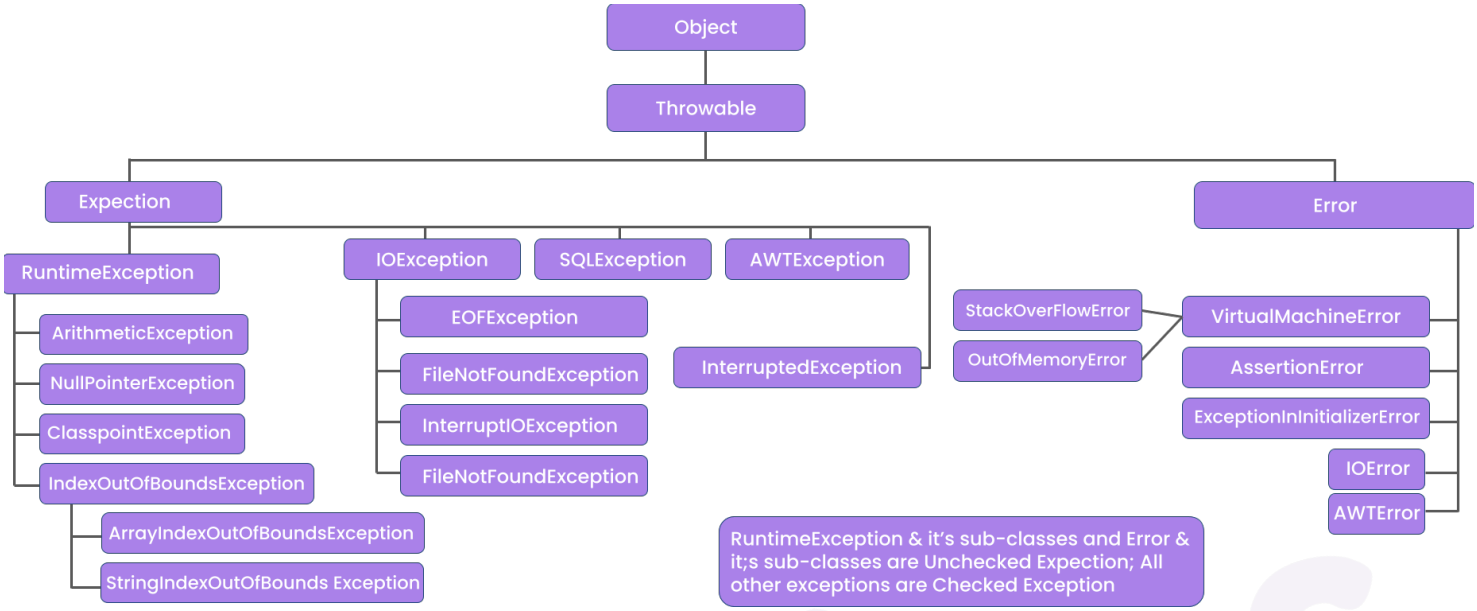
Example

```
class TestApp{
 public static void main(String[] args){
  System.out.println("Entering main");
  doStuff();
  System.out.println(10/0);
  System.out.println("Exiting main");
 }
 public static void doStuff(){
  System.out.println("Entering doStuff");
  doMoreStuff();
  System.out.println("hiee");
                System.out.println("Exiting doStuff");
 }
 public static void doMoreStuff(){
  System.out.println("Entering doMoreStuff");
  System.out.println("hello");
  System.out.println("Exiting doMoreStuff");
 }
}
```

Output::

Entering main
Entering doStuff
Entering doMoreStuff
Hello
Exiting doMoreStuff
hiee
ExitingdoStuff
Exception in thread "main" java.lang.ArithmeticException : / by zero
at TestApp.main().

Exception hierarchy



Exception

- Most of the cases exceptions are caused by our program and these are recoverable

Example::

If FileNotFoundException occurs then we can use a local file and we can continue the rest of the program execution normally.

Error

- Most of the cases errors are not caused by our program these are due to lack of system resources and these are non-recoverable.

Example::

If OutOfMemoryError occurs being a programmer we can't do anything the program will be terminated abnormally.

Checked vs UnCheckedException

- The exceptions which are checked by the compiler whether programmer handling or not, for smooth execution of the program at the runtime are called “CheckedException”.

Example::

FileNotFoundException,IOException,SQLException...

- The exceptions which are not checked by the compiler whether the programmer is handling or not such type of exceptions are called "UnCheckedExceptions".

Example::

NullPointerException,ArithmeticException

Note

- RunTimeException and its child classes,Error and its child classes are called as "UncheckedException", remaining all exceptions are considered as "CheckedExceptions".
- Whether the exception is checked or unchecked compulsorily it should occur at runtime only and there is no chance of Occuring any exception at compile time.

Fully Checked Exception

A checked exception is said to be a fully checked exception if and only if all its child classes are also checked.

Example:

IOException, InterruptedException

Partially Checked Exception

A checked exception is said to be partially checked if and only if some of its child classes are unchecked.

Example:

Throwable, Exception

Describe the behaviour of following exceptions?

1. RunTimeException **Ans.UncheckedException**
2. Error **Ans.UncheckedException**
3. IOException **Ans.fullychecked**
4. Exception **Ans.partiallychecked**
5. InterruptedException **Ans.fullychecked**
6. Throwable **Ans.partially checked**
7. ArithmeticException **Ans.unchecked**
8. NullPointerException **Ans.unchecked**
9. FileNotFoundException **Ans.fullychecked**

Control flow in try catch

```
try{
 Statement-1;
 Statement-2;
 Statement-3;
}catch( X e){
 Statement-4;
}
Statement5;
```

Case1

If there is no exception
1,2,3,5 normal termination

Case2

if an exception raised at statement2 and corresponding catch block matched
1,4,5 normal termination

Case 3
if any exception raised at statement2 but the corresponding catch block not matched, followed by abnormal termination

Case 4
if an exception raised at statement 4 or statement 5 then its always abnormal termination of the program.

Note

1. Within the try block if anywhere an exception is raised then the rest of the try block won't be executed even though we handled that exception.Hence we have to place/take only risk code inside try block and length of the try block should be as less as possible.
2. If any statement which raises an exception and it is not part of any try block then it is always an abnormal termination of the program.

Various methods to print exception information
Throwable class defines the following methods to print exception information to the console

printStackTrace()

- This method prints exception information in the following format.
- Name of the exception:description of exception stack trace

toString()

- This method prints exception information in the following format
- Name of the exception : description of exception

getMessage()

- This method returns only the description of the exception Description.

Example:

```
public class Test{
       public static void main(String[] args){
  try{
   System.out.println(10/0);
  }catch(ArithmeticException e){
   e.printStackTrace();
   System.out.println(e);
   System.out.println(e.getMessage());
  }
 }
}
```

Note:
Default exception handler internally uses **printStackTrace()** method to print exception information to the console.

Try with multiple catch Blocks

The way of handling the exception is varied from exception to exception,hence for every exception type it is recommended to take a separate catch block. That is try with multiple catch blocks is possible and recommended to use.

Example 1

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

This approach is not recommended because for any type of Exception we are using the same catch block.

Example 2

```
try{
  ....
  ....
  ....
}catch(FileNotFoundException fe){

}catch(ArithmeticException ae){

}catch(SQLException se){

}catch(Exception e){

}
```

- This approach is highly recommended because for any exception raise we are defining a separate catch block.
- If try with multiple catch blocks present then order of catch blocks is very important, it should be from child to parent.
- By mistake if we are taking from parent to child then we will get **"CompileTimeError" saying "exception XXXX has already been caught".**

Example 1

```
class Test{
 public static void main(String[] args){
  try{
   System.out.println(10/0);
  }catch(Exception e){
   e.printStackTrace();
  }catch(ArithmeticException ae){
   ae.printStackTrace();
  }
 }
}
```

CE: exception java.lang.ArithmeticException has already been caught

Example 2

```
class Test{
 public static void main(String[] args){
  try{
   System.out.println(10/0);
  }catch(ArithmeticException ae){
   ae.printStackTrace();
  }catch(Exception e){
   e.printStackTrace();
  }
 }
}
```

Output: Compile successfully

finally block

- It is not recommended to clean up code inside a try block because there is no guarantee for the execution of every statement inside a try block.
- It is not recommended to place clean up code inside the catch block becoz if there is no exception then the catch block won't be executed.
- we require some place to maintain clean up code which should be executed always irrespective of whether exceptions are raised or not raised and whether or not handled.
- Such type of best place is nothing but finally block.
- Hence the main objective of finally block is to maintain cleanup code.

Example

```
try{
 risky code
}catch( X e){
 handling code
}finally{
 cleanup code
}
```

The speciality of finally block is it will be executed always irrespective of whether the exception is raised or not raised and whether handled or not handled.

Case-1: If there is no Exception

```
class Test{
 public static void main(String... args){
  try{
   System.out.println("try block gets executed");
  }catch(ArithmeticException e){
   System.out.println("catch block gets executed");
  }finally{
   System.out.println("finally block gets executed");
  }
 }
}
```

Output:
try block gets executed
finally block gets executed

Case-2: If an Exception is raised,but the corresponding catch block matched

```
class Test{
 public static void main(String... args){
  try{
   System.out.println("try block gets executed");
   System.out.println(10/0);
  }catch(ArithmeticException e){
   System.out.println("catch block gets executed");
  }finally{
   System.out.println("finally block gets executed");
  }
 }
}
```

Output:
try block gets executed
catch block gets executed
finally block gets executed

Case-3: If an Exception is raised,but the corresponding catch block not matched

```
class Test{
 public static void main(String... args){
  try{
   System.out.println("try block gets executed");
   System.out.println(10/0);
  }catch(NullPointerException e){
   System.out.println("catch block gets executed");
  }finally{
   System.out.println("finally block gets executed");
  }
 }
}
```

Output:
Try block gets executed
finally block gets executed
Exception in thread "main" java.lang.ArithemeticException :/by Zero atTest.main(Test.java:8)

return vs finally

- Even though the return statement present in try or catch blocks first finally will be executed and after that only return statement will be considered
- finally block dominates return statement.

Example

```
class Test{

 public static void main(String... args){
  try{
   System.out.println("try block executed");
   return;
  }catch(ArithmeticException e){
   System.out.println("catch block executed");
  }finally{
   System.out.println("finally block executed");
  }
 }
}
```

Output:
try block executed
finally block executed

Example
If the return statement present try,catch and finally blocks then finally block return statement will be considered.

```
class Test{
 public static void main(String... args){
  System.out.println(m1());
 }
 public static int m1(){
  try{
   System.out.println(10/0);
   return 777;
  }catch(ArithmeticException e){
   return 888;
  }finally{
   return 999;
  }
        }
}
```

inally vs System.exit(0)

- There is only one situation where the finally block won't be executed whenever we are using System.exit(0) method.
- Whenever we are using System.exit(0) then the JVM itself will be shutdown, in this case the finally block won't be executed.
  - ie,.. System.exit(0) dominates finally block

System.exit(0)

- This argument acts as status code, Instead of Zero, we can't take any integer value
- Zero means normal termination, non zero means abnormal termination
- This status code internally used by JVM,whether it is zero or non-zero there is no change in the result and effect is the same w.r.t program

Difference b/w final,finally and finalize

final

- final is the modifier applicable for classes, methods and variables
- If a class is declared as the final then child class creation is not possible.
- If a method is declared as the final then overriding of that method is not possible.
- If a variable is declared as the final then reassignment is not possible.

finally

- It is a final block associated with try-catch to maintain clean up code, which should be executed always irrespective of whether exceptions are raised or not raised and whether handled or not handled.

finalize

- It is a method, always invoked by Garbage Collector just before destroying an object to perform cleanup activities.

Note

- finally block meant for cleanup activities related to try block whereas finalize() method for cleanup activities related to object.
- To maintain cleanup code finally block is recommended over finalize() method because we can't expect exact behaviour of GC.

Handling vs Ducking an Exception

- It is highly recommended to handle exceptions
- In our program the code which may rise exception is called "risky code"
- We have to place our risky code inside the try block and corresponding handling code inside the catch block.

Syntax

```
try{
      ...
      ... risky code
      ...
 }catch(XXXX e){
     ...
     ... handling code
     ...
}
```

Code without using try catch

```
class Test{
 public static void main(String... args){
  System.out.println("statement1");
  System.out.println(10/0);
  System.out.println("statement2");
 }
}
```

Output:
statement1
RE: AE:/by zero at Test.main()
Abnormal termination

with using try catch

```
public class Test{
 public static void main(String... args){
  System.out.println("statement1");
  try{
      System.out.println(10/0);
  }catch(ArithmeticException e){
      System.out.println(10/2);
  }
  System.out.println("statement2");
 }
}
```

Output:
Statement1
5

Statement2
Rethrowing an Exception(throw, throws) and Custom Exception

throw keyword in java

- This keyword is used in java to throw the exception object manually and inform jvm to handle the exception.

Syntax:: throw new ArithmeticException("/ by zero");

Eg#1.

```
class Test{
 public static void main(String... args){
  System.out.println(10/0);
 }
}
```

- Here the jvm will generate an Exception called "ArithmeticException", since main() is not handling it will handover the control to jvm, jvm will handover to DEH to dump the exception object details through printStackTrace().

vs

```
class Test{
 public static void main(String... args){
  throw new ArithmeticException("/by Zero");
 }
}
```

- Here the programmer will generate ArithmeticException,and this exception object will be delegated to JVM, jvm will handover the control to DefaultExceptionHandler to dump the exception information details through printStackTrace().
- throw keyword is mainly used to throw an customised exception not for predefined exception.

Eg#2

```
class Test{
 static ArithmeticException e =new ArithmeticException();
 public static void main(String... args){
  throw e;
 }
}
```

Output:
Exception in thread "main" java.lang.ArithmeticException

Eg#3

```
class Test{
 static ArithmeticException e;
 public static void main(String... args){
  throw e;
 }
}
```

Output:
Exception in thread "main" java.lang.NullPointerException.

Case2

- After throw statement we can't take any statement directly otherwise we will get a compile time error saying an unreachable statement.

Eg#1

```
class Test{
 public static void main(String... args){
  System.out.println(10/0);
  System.out.println("hello");
 }
}
```

Output:
Exception in thread "main" java.lang.ArithmeticException

Eg#2

```
class Test{
  public static void main(String... args){
  throw new ArithmeticException("/ by zero");
  System.out.println("hello");
  }
}
```

Output:
CompileTime error
Unreachable statement
System.out.println("hello");

Case3

- We can use throw keyword only for Throwable types otherwise we will get a compile time error saying incompatible type.

Eg#1

```
class Test3{
 public static void main(String... args){
  throw new Test3();
 }
}
```

Output:
Compile time error.
found::Test3
required:: java.lang.Throwable

Eg#2

```
public class Test3 extends RunTimeException{
 public static void main(String... args){
  throw new Test3();
 }
}
```

Output:
RunTimeError: Exception in thread "main" Test3

Customized Exceptions (User defined Exceptions)

- Sometimes we can create our own exception to meet our programming requirements.
- Such type of exceptions are called customised exceptions (user defined exceptions).

Example

- InSufficientFundsException
- TooYoungException
- TooOldException

Eg#1

```
class TooYoungException extends RuntimeException{
 TooYoungException(String s){
  super(s);
 }
}
class TooOldException extends RuntimeException{
 TooOldException(String s){
  super(s);
 }
}

public class CustomizedExceptionDemo{
 public static void main(String[] args){
 int age=Integer.parseInt(args[0]);
 if(age>60){
     throw new TooYoungException("please wait some more time.... u
will get best match");
        }
 else if(age<18){
  throw new TooOldException("u r age already crossed....no chance of
getting married");
 }
 else{
  System.out.println("you will get match details soon by email");
 }
  }
}
```

Output:

java CustomizedExceptionDemo 61

Exception in thread "main" TooYoungException: please wait some more time....
u will get best match at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:21)

java CustomizedExceptionDemo 27

You will get match details soon by email

java CustomizedExceptionDemo 9

Exception in thread "main" TooOldException: u r age already crossed....no chance of getting married at CustomizedExceptionDemo.main (CustomizedExceptionDemo.java:25)

Note:

It is highly recommended to maintain our customized exceptions as unchecked by extending **RuntimeException.**

throws statement

In our program if there is a chance of raising a checked exception then we should handle either by **try catch** or by **throws** keyword otherwise the code won't compile.

Eg#1

```
import java.io.*;
class Test3{
 public static void main(String... args){
  PrintWriter pw=new PrintWriter("abc.txt");
  pw.println("Hello world");
 }
}
```

CE: unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown

Eg#2

```
class Test3{
 public static void main(String... args){
  Thread.sleep(3000);
 }
}
```

CE: unreported exception java.lang.InterruptedException; must be caught or declared to be thrown

We can handle this compile time error by using the following 2 ways

1. using try catch
2. using throws keyword

1. using try catch

```
class Test3{
 public static void main(String... args){
  try{
   Thread.sleep(5000);
  }catch(InterruptedException ie){}
 }
}
```

Output:: compiles and successfully runs.

2. using throws keyword

```
class Test{
 public static void main(String... args) throws InterruptedException{
  Thread.sleep(5000);
 }
}
```

Output:: **compiles and successfully runs.**

- we can use throws keyword to delegate the responsibility of exception handling to the caller method.
- The caller method is responsible for handling the exception.

Note

- Hence the main objective of the "throws" keyword is to delegate the responsibility of exception handling to the caller method.
- throws keyword required only for checked exception. usage of throws keyword for unchecked exceptions there is no use.
- "throws" keyword required only to convince compiler.Usage of throws keyword does not prevent abnormal termination of the program.
- Hence recommended to use try-catch over throws keyword.

Eg#1

```
class Test{
 public static void main(String... args) throws InterruptedException{
  doWork();
 }
 public static void doWork() throws InterruptedException{
  doMoreWork();
 }
 public static void doMoreWork() throws InterruptedException{
  Thread.sleep(5000);
 }
}
```

In the above code, if we remove any of the throws keyword it would result in "CompileTimeError".

Case studies of Throwable

Case 1:

- we can use throws keyword only for Throwable types otherwise we will get a compile time error.

```
class Test3{
 public static void main(String... args)throws Test3{
 }
}
```

Output:: **Compile Time Error,Test3 cannot be Throwable**

```
class Test3 extends RuntimeException{
 public static void main(String... args)throws Test3{
 }
}
```

Output:: **Compiles and run successfully**

Case 2:

```
public class Test3 {
 public static void main(String... args) {
  throw new Exception();
 }
}
```

Output::
Compile Time Error
unreported Exception must be caught or declared to be thrown

```
public class Test3 {
 public static void main(String... args) {
  throw new Error();
 }
}
```

Output::
RunTimeException
Exception in thread "main" java.lang.Error at Test3.main(Test3.java:4)

Case 3:

- In our program within the try block,if there is no chance of raising an exception then we can't write a catch block for that exception, otherwise we will get a Compile Time Error saying "exception XXX is never thrown in the body of the corresponding try statement".
- But this rule is applicable only for fully checked exceptions only.

Eg#1

```
public class Test3
{
 public static void main(String... args) {
  try
  {
   System.out.println("hiee");
  }
  catch (Exception e)
  {
  }
 }
}
```

Output:: hiee

Eg#2

```
public class Test3
{
 public static void main(String... args) {
  try
  {
   System.out.println("hiee");
  }
  catch (ArithmeticException e)
  {
  }
 }
}
```

Output:: **hiee**

Eg#3

```
public class Test3
{
 public static void main(String... args) {
  try
  {
   System.out.println("hiee");
  }
  catch (java.io.FileNotFoundException e)
  {
  }
 }
}
```

Output:: **Compile time error(fully checked Exception)**

Eg#4

```
public class Test3
{
 public static void main(String... args) {
  try
  {
   System.out.println("hiee");
  }
  catch (InterruptedException e)
  {
  }
 }
}
```

Output:: Compile time error(fully checked Exception)
exception InterruptedException is never thrown in the body of the corresponding try statement.

Eg#5

```
public class Test3
{
 public static void main(String... args) {
  try
  {
   System.out.println("hiee");
  }
  catch (Error e)
  {
  }
 }
}
```

Output:: hiee

Case 4:
we can use throws keyword only for constructors and methods but not for classes.

Eg#1

```
class Test throws Exception    //invalid
{
 Test() throws Exception{ //valid

 }
 methodOne() throws Exception{ //valid

 }
}
```

Note:
Exception handling keywords summary

- try => maintain risky code
- catch=> maintain handling code
- finally=> maintain cleanup code
- throw => To hanover the created exception object to JVM manually
- throws=> To delegate the Exception object from called method to caller method.

Various compile time errors in ExceptionHandling

1. Exception XXXX is already caught
2. Unreported Exception XXXX must be caught or declared to be thrown.
3. Exception XXXX is never thrown in the body of the corresponding try statement.
4. try without catch,finally
5. catch without try
6. finally without try
7. incompatible types : found : xxxx / required : Throwable
8. unreachable code

Control flow of Exception Handling concept

Control flow in try catch finally

```
try{
 statement-1
 statement-2
 statement-3

}catch(Exception e){
 statement-4

}finally{
 statement-5
}
 statement-6
```

Case1: If there is no exception.

Case2: If an exception is raised at statement2 and the corresponding catch block is matched.

Case3: If an exception is raised at statement2 and corresponding catch block is not matched

Case4 : If an exception is raised at statement4

Case5 : If an exception is raised at statement5

Control flow in Nested try-catch-finally

```
try{
 stmt-1
 stmt-2
 stmt-3

 try{
     stmt-4;
     stmt-5;
     stmt-6;
 }catch(X e){
  stmt-7;
 }finally{
     stmt-8;
 }
 stmt-9;
}catch(Y e){
 stmt-10;
}finally{
 stmt-11;
}
    stmt-12;
```

Case1 : If there is no exception

Case2 : If an exception is raised at statement2 and the corresponding catch block is matched.

Case3 : If an exception is raised at statement2 and the corresponding catch block is not matched.

Case4 : If an exception is raised at statement5 and corresponding inner catch block is matched

Case5 : If an exception is raised at statement5 and the inner catch has not matched but the outer catch block is matched.

Case6 : If an exception is raised at statement5 and both inner catch and outer catch block is not matched.

Case7 : If an exception is raised at statement7 and corresponding catch block is matched

Case8 : If an exception is raised at statement7 and corresponding catch block is not matched

Case9 : If an exception is raised at statement8 and corresponding catch block is matched.

Case10 : If an exception is raised at statement8 and the corresponding catch block is not matched.

Case11 : If an exception is raised at statement 9 and the corresponding catch block is matched.

Case12 : If an exception is raised at statement 9 and the corresponding catch block is not matched.

Case13 : If an exception is raised at statement 10

Case14 : If an exception is raised at statement 11 or 12

- If we are not entering into a try block then finally the block won't be executed.
- If we are entering into a try block without executing the final block we can't come out.
- We can write try inside try, nested try-catch is possible.
- Specific exceptions can be handled using inner try catch and generalised exceptions can be handled using outer try catch.

Note:

```
public class TestApp{
 public static void main(String... args){
  try{
   System.out.println(10/0);
  }catch(ArithmeticException ae){
   System.out.println(10/0);
  }finally{
   String s=null;
   System.out.println(s.length());
  }
 }
}
```

Default exception handler handles the most recent exception and it can handle only one exception.
RE: java.lang.NullPointerException

Various possible cases of Exception

```
1. try{
   }catch(X e){     //valid
   }

2. try{
   }catch(X e){
                   //valid
   }catch(Y e){
   }

3. try{
   }catch( X e){
   //invalid
   }catch( X e){
   }

4. try{
    }finally{      //valid

   }

5. try{
   }catch(X e){
         //valid
   }finally{
   }

6. try{}  //invalid

7. catch(){}    //invalid

8. finally{}   //invalid

9. try{}
   System.out.println("Hello");   //invalid
   catch(){}

10. try{}
    catch(X e){}
    System.out.println("hello");  //invalid
    catch(Y e){}
```

11.
```
try{}
catch(X e){}
System.out.println("hello");        //invalid
finally{}
```

12.
```
try{}
finally{}
catch(X e){}      // invalid
```

13.
```
try{}
catch(X e){}
try{}
finally{}
```

14.
```
try{}
catch(X e){}
finally{}
finally{}     //invalid
```

15.
```
try{}
catch(X e){
   try{}
     catch(Y e1){}                  //valid
}
```

16.
```
try{}
catch(X e){}
finally{
  try{}
    catch(Y e1){}                   //valid
    finally{}
}
```

17.
```
try{
 try{}                              //invalid
}
```

18.
```
try
 System.out.println("hello");       //invalid
catch(X e){}
```

19.
```
try{}
catch( X e1)
  System.out.println("hello");      //invalid
```

20.
```
try{}
catch( NullPointerException e1){}  //invalid
finally
 System.out.println("Hello");
```

Rules associated with Exception handling

- Whenever we are writing try block compulsorily we should write either catch block or finally try without catch and finally is invalid.
- Whenever we are writing a catch block, compulsorily try block is required.
- Whenever we are writing a finally block, compulsorily try block is required.
- try catch and finally order is important.
- With in try catch finally blocks, we can take try catch finally.
- For try catch finally blocks curly braces are mandatory.

1.7 version Enhancements

- try with resource
- try with multi catch block

until jdk1.6, it is compulsorily required to write a finally block to close all the resources which are open as a part of try block.

Example:

```
BufferReader br=null
      try{
  br=new BufferedReader(new FileReader("abc.txt"));
      }catch(IOException ie){
          ie.printStackTrace();
      }finally{
  try{
    if(br≠null){
  br.close();
    }
          }catch(IOException ie){
  ie.printStackTrace();
   }
}
```

Problems in the approach

- Compulsorily the programmer is required to close all opened resources which increases the complexity of the program
- Compulsorily we should write finally block explicitly, which increases the length of the code and reviews readability.
- To Overcome this problem SUN MS introduced try with resources in "1.7" version of jdk.

try with resources

- In this approach, the resources which are opened as a part of try block will be closed automatically once the control reaches to the end of
- try block normally or abnormally,so it is not required to close explicitly so the complexity of the program would be reduced.
- It is not required to write a finally block explicitly,so length of the code would be reduced and readability is improved.

```
try(BufferedReader br=new BufferedReader(new FileReader("abc.txt")){
 //use br and perform the necessary operation
        //once the control reaches the end of try automatically br will be closed
}catch(IOException ie){
     //handling code
}
```

Rules of using try with resource

1. we can declare any no of resources, but all these resources should be separated with ;
eg#1.

```
try(R1;R2;R3;){
         //use the resources
   }
```

2. All resources are said to be AutoCloseable resources iff the class implements an interface called "java.lang.AutoCloseable" either directly or indirectly

eg:: `java.io package classes, java.sql.package classes`

3. All resource references by default are treated as implicitly final and hence we can't perform reassignment within the try block.

```
try(BufferedReader br=new BufferedReader(new FileWriter("abc.txt")){
  br=new BufferedReader(new FileWriter("abc.txt"));
}
```

output:: CE: can't reassign a value

4. Until the 1.6 version try should compulsorily be followed by either catch or finally, but from 1.7 version we can only take try with resources without catch or finally.

```
try(R){
  //valid
}
```

5. Advantage of try with resources concept is finally block will become dummy because we are not required to close resources explicitly.

MultiCatchBlock

- Till jdk1.6, even though we have multiple exceptions having the same handling code we have to write a seperate catch block for every exception, it increases the length of the code and reviews readability.

Eg#1

```
try{
    ....
    ....
    ....
    ....
}catch(ArithmeticException ae){
 ae.printStackTrace();
}catch(NullPointerException ne){
 ne.printStackTrace();
}catch(ClassCastException ce){
 System.out.println(ce.getMessage());
}catch(IOException ie){
 System.out.println(ie.getMessage());
}
```

To overcome this problem SUMS has introduced "Multi catch block" concept in 1.7 version

```
try{
    ....
    ....
    ....
    ....
}catch(ArithmeticException |NullPointerException e){
 e.printStackTrace();
}catch(ClassCastException |IOException e){
 e.printStackTrace();
}
```

- In multi catch blocks,there should not be any relation b/w exception types(either child to parent or parent to child or same type) it would result in compile time error.

Eg#1

```
try{

     }catch( ArithmeticException | Exception e){
  e.printStackTrace();
}
```

output:: CompileTime Error