ExceptionHandling

1. Introduction
2. Runtime stack mechanism
3. Default exception handling in java
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 between final, finally, finalize
11. Control flow in try catch finally
12. Control flow in nested try catch finally
13. Various possible combinations of try catch finally
14. throw keyword
15. throws keyword
16. Exception handling keywords summary
17. Various possible compile time errors in exception handling
18. Customized exceptions
19. Top-10 exceptions
20. 1.7 Version Enhancements
    1. try with resources
    2. multi catch block
21. Exception Propagation
22. Rethrowing an Exception

Exception

=> An unwanted/expected event that disturbs the normal flow of execution of 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 way to continue rest of the program normally.
 This way of defining an alternative is nothing but "Exception handling".

example
   Suppose our programming requirement is to read a data from a file locating at one location,
   At run time if the file is not available then our programm should terminate succesfully.
   Solution:: Provide the local file to terminate the program succesfully,This way of defining alternative is nothing but "Exception handling".

eg#1.

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

RunTimeStackMechansim
=====================
For every thread in java language, jvm create a seperate 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()

eg::
```
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{

}catch(Exception e){

}
```

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(), 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.

since it is a method, it will propogate the exception object to caller method.

2.Now jvm will check whether the caller method is having the code of caller method or not

if it is not available, then that method will be abnormally terminated.

3.Simiar way if the exception object is propogated to main(), jvm will check whether the main()

is having a code for handling or not, if not then the exception object will be propogated to

JVM by terminating the main().

4.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
```

eg#2.

```
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()


eg#3.
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
===================
   refer Exception.png

Exception:: Most of the cases exceptions are caused by our program and
these are recoverable
   ex:: If FileNotFoundException occurs then we can use local file and we
can continue 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.
   ex:: If OutOfMemoryError occurs being a programmer we can't do
anything the program will be
        terminated abnormally.

Checked vs UnCheckedExceptions
==============================
 => 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.
```

eg::FileNotFoundException,IOException,SQLException...

=> The exceptions which are not checked by the compiler whether programmer is handling or not
such type of exceptions are called as "UnCheckedExceptions".
eg::NullPointerException,ArithmeticException

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

A checked exception is said to be fully checked exception if and only if all its child classes
are also checked.
1. IOException
2. InterruptedException

A checked exception is said to be partially checked if and only if some of its child classes
are unchecked.
eg:: Throwable, Exception

Describe the behaviour of following exceptions?
A. RunTimeException => UncheckedException
B. Error => UncheckedException
C. IOException => fully checked
D. Exception => partially checked
E. InterruptedException => fully checked
F. Throwable => partially checked
G. ArithemeticException => unchecked
H. NullPointerException => unchecked
I. FileNotFoundException => fully checked

Customized Exception handling
==========
1. It is highly recomended to handle exceptions
2. In our program the code which may rise exception is called "risky code"
3. We have to place our risky code inside try block and corresponding handling code inside
catch block.

Example::
```
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(ArithemticException e){
                System.out.println(10/2);
            }
            System.out.println("statement2");
      }
 }
output:
  Statement1
    5
  Statement2

=========================
Control flow in try catch
=========================
try{
      Statement-1;
      Statement-2;
      Statement-3;
}catch( X e){
      Statement-4;
}
Statement5;

Case 1:: If there is not exception
         1,2,3,5 normal termination

Case 2:: 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
          1 follwed 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 raised then rest of the try block wont
     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 abnormal termination of the program.

  3. There may be a chance of raising an exception inside catch and finally blocks also in
     addition to try block.

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
                 stacktrace

toString()   => This method prints exception information in the following format
                   Name of the exception : description of exception

getMessage()   => This method returns only description of the exception Description.

eg::
```
public class Test{
       public static void main(String[] args){
            try{
                  System.out.println(10/0);
            }catch(ArithmeticException e){
                  e.printStackTrace(); => java.lang.ArithmeticException :/by Zero
                                                    at Test.main(Test.java:6)
                  System.out.println(e); => java.lang.ArithmeticException:/by Zero
                  System.out.println(e.getMessage());=> /by Zero
            }
      }
}
```
Default exception handler internally uses printStatckTrace() method to print exception
information to the console.

Try with mulitple catch Blocks
==========
The way of handling the exception is varied from exception to exception,hence for every exception

type it is recomended to take a separate catch block. That is try with multiple catch blocks is
possible and recomended to use.

Example#1
=========
```
try{
      ...
      ...
      ...
}
catch(Exception e){
      default handler
}
```
This approach is not recomended because for any type of Exception we are using
same catch block.
==========
Example#2
==========
```
try{
      ....
      ....
      ....
}catch(FileNotFoundException fe){

}catch(ArithmeticException ae){

}catch(SQLException se){

}catch(Exception e){

}
```
This approach is highly recomended because for any exception raise we are defining a seperate
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 succesfully
=====================================================================
finally
  . It is not recomended to take clean up code inside try block becoz
there is no gurantee for
    the execution of every statement inside a try block.
  . It is not recomended to place clean up code inside catch block becoz
if there is no exception
    then catch block wont be executed.
  . we require some place to maintain clean up code which should be
executed always irrespective
    of whether exception 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#1.
==========
try{
      risky code
}catch( X e){
      handling code
}finally{
      cleanup code
}

The speciality of finally block is it will be executed always
irrespecitve 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 return statement present in try or catch blocks first finally will be executed and
after that only return statement will be considered.ie 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 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;
            }
        }
}
```

finally vs System.exit(0)
==========
   There is only one situation where the finally block wont be executed is whenever we are using
System.exit(0) method.
   When ever we are using System.exit(0) then JVM itself will be shutdown, in this case finally
block wont be executed.

ie,.. System.exit(0) dominates finally block

```
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;
            }
        }
}
```

Output::
try

Note:: System.exit(0);
1. This argument acts as status code, Instead of Zero, we cant take any integer value
2. Zero means normal termination, non zero means abnormal termination
3. This status code internally used by JVM,whether it is zero or non-zero there is no change

in the result and effect is 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 declared as the fianl then child class creation is not possible.
=> If a method declared as the final then overriding of that method is not possible.
=> If a variable declared as the final then reassignement 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 whethere exception 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 activites.

Note::
1. finally block meant for cleanup activites related to try block where as finalize() method
for cleanup activites related to object.
2. To maintain cleanup code finally block is recomended over finalize() method because we can't
expect exact behaviour of GC.

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 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 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 statement5 and corresponding inner catch block is matched
Case5: If an exception is raised at statement5 and inner catch has not matched but 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 corresponding catch block is not matched.
Case11: If an exception is raised at statement 9 and corresponding catch block is matched.
Case12: If an exception is raised at statement 9 and 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

Note: If we are not entering into try block then finally block wont be executed.
If we are entering into try block without executing finally 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 generalized 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){}
}
```

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

}
```

17.
```
try{
  try{}
}
```

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

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

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

Rules::
1. Whenever we are writing try block compulsorily we should write either catch block or finally

try without catch and finally is invalid.
2. Whenever we are writing catch block, compulsorily try block is required.
3. Whenever we are writing finally block, compulsorily try block is required.
4. try catch and finally order is important.
5. With in try catch finally blocks, we can take try catch finally.
6. For try catch finally blocks curly braces are mandatory.

throw keyword in java
==========
This keyword is used in java to throw the exception object manually and informing 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 DEH to dump the excpetion information details through
printStackTrace().

Note:: throw keyword is mainly used to throw an customized exception not for predefined exception.

eg::
```
  class Test{
      static ArithmeticException e =new ArithmeticException();
      public static void main(String... args){
            throw e;
      }
  }
```
Output::
```
   Exception in thread "main" java.lang.ArithmeticException
```
eg::
```
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 compile time
error saying 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
                    VS
eg#2.
class Test{
  public static void main(String... args){
            throw new ArithmeticExeption("/ 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
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 customized exceptions (user defined exceptions).
Example:
1. InSufficientFundsException
2. TooYoungException
3. TooOldException

Program:
```
class TooYoungException extends RuntimeException{
      TooYoungException(String s){
            super(s);
      }
}
class TooOldException extends RuntimeException{
      TooOldException(String s){
            super(s);
      }
}
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 e-
mail");
      }
  }
}
```
Output
======
```
1)E:\corejava>java CustomizedExceptionDemo 61
      Exception in thread "main" TooYoungException: please wait some more
time.... u will get                      best match at
CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:21)
2)E:\corejava>java CustomizedExceptionDemo 27
      You will get match details soon by e-mail

3)E:\corejava>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 checked exception then compulsory 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){}
      }
}
```
ouput:: compiles and succesfully running

2. using throws keyword
```
class Test{
      pubilc static void main(String... args) throws InterruptedException{
            Thread.sleep(5000);
      }
}
```
output:: compiles and succesfully running.
=> we can use throws keyword to delegate the responsibility of exception handling to the caller
    method. Then caller method is responsible to handle the exception.

Note::
  . Hence the main objective of "throws" keyword is to delegate the responsiblity of exception
    handling to the caller method.
  . throws keyword required only for checked exception. usage of throws keyword for unchecked
    exception there is no use.
  . "throws" keyword required only to convince compiler.Usage of throws keyword does not prevent

abnormal termination of the program.

Hence recomended to use try-catch over throws keyword.

```
eg#1.
class Test{
      public static void main(String... args) throws InteruptedException{
            doWork();
      }
      pubilc static void doWork() throws InteruptedException{
            doMoreWork();
      }
      public static void doMoreWork() throws InteruptedException{
            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 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 succesfully

Case2::

```
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)
```

Case3::

In our program with in try block,if there is no chance of rising an exception then we can't
write catch block for that exception, otherwise we will get Compile Time Error saying
"exception XXX is never thrown in the body of 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)
            {

            }
      }
}
```

Ouput::Compile time error(fully checked Exception)

eg#4.

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

            try
            {
                  System.out.println("hiee");
            }
            catch (InteruptedException e)
            {

            }
      }
}
```
Ouput::Compile time error(fully checked Exception)
exception InterruptedException is never thrown in the body of correspoding try statement.

eg#5.
```
public class Test3
{
      public static void main(String... args) {

            try
            {
                  System.out.println("hiee");
            }
            catch (Error e)
            {

            }
      }
}
```
Output:hiee

Case4:: 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

      }
}
```

Exception handling keywords summary
==========
1. try => maintain risky code
2. catch=> maintain handling code
3. finally=> maintain cleanup code
4. throw => To hanover the created exception object to JVM manually
5. throws=> To delegate the Exception object from called method to caller method.

Various compile time errors in ExceptionHandling
==========

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

Exception which are normally occured in java coding
==========
1. Based on the events occured exceptions are classified into 2 types
    a. JVM Exceptions
    b. Programtic Exceptions

JVM Exceptions
  => The exceptions which are raised automatically by the jvm whenever a particular event occurs
     are called JVM Exceptions
     eg:: ArrayIndexOutOfBoundsException
          NullPointerException

ProgramaticExceptions
  => The exceptions which are raised explicitly by the programmer or by API developers is called
     as "Programatic Exceptions".
      eg:: IllegalArgumentException,NumberFormatException

ArrayIndexOutOfBoundsException
   =>This exception is raised automatically whenever we are trying to access array elements which
     is out of the range.

Top10JavaExceptions
==========
1. ArithmeticException
2. NullPointerException
3. StackOverFlowError
4. IllegalArgumentException
      eg:: Thread t=new Thread();
         t.setPriority(10);
         t.setPriority(100);//invalid
5. NumberFormatException
6. ExceptionInInitializerError
7. ArrayIndexOutOfBoundsException
8. NoClassDefFoundError
9. ClassCastException
10. IllegalStateException(learn in servlet programming)
11. AssertionError(learn in Junit)

JVMException
==========
 a. ArithmeticException
 b. NullPointerException
 c. ArrayIndexOutOfBoundsException
 d. StackOverFlowError

```
 e. ClassCastException
 f. ExceptionInInitalizerError

ProgrammaticException
=====================
 a. IllegalArgumentException
 b. NumberFormatException
 c. IllegalStateException
 d. AssertionError


Rules of Overriding when exception is involved
==============================================
While Overriding if the child class method throws any checked exception
compulsorily the parent class method should throw the same checked
exception or its parent otherwise we will get
Compile Time Error.
  There are no restrictions on UncheckedException.

eg#1.
class Parent{
      public void methodOne();
}
class Child extends Parent{
      public void methodOne() throws Exception{}
}
error: methodOne() in Child cannot override methodOne() in Parent
       public void methodOne() throws Exception{}
       overridden method does not throw Exception

examples
parent: public void methodOne() throws Exception{}
child : public void methodOne()
Output:: valid

parent: public void methodOne(){}
child : public void methodOne() throws Exception{}
output:: invalid

parent: public void methodOne()throws Exception{}
child : public void methodOne()throws Exception{}
Output:: valid

parent: public void methodOne()throws IOException{}
child : public void methodOne()throws IOException{}
Output:: valid

parent: public void methodOne()throws IOException{}
child : public void methodOne()throws
FileNotFoundException,EOFException{}
Output:: valid

parent: public void methodOne()throws IOException{}
child : public void methodOne()throws
FileNotFoundException,InterruptedException{}
Output:: invalid

parent: public void methodOne()throws IOException{}
child : public void methodOne()throws
FileNotFoundException,ArithmeticException{}
```

Output:: valid

parent: public void methodOne()
child : public void methodOne()throws
ArithmeticException,NullPointerException,RuntimeException{}
Output:: valid

Rules w.r.t constructor
=======================
eg#1
```
class Parent{
      Parent() throws java.io.IOException{

      }
}
class Child extends Parent{

}
```
output::CompileTime Error

eg#2.
```
class Parent{
      Parent() throws java.io.IOException{}
}
class Child extends Parent{
      Child() throws Exception{super();}
}
```

If parent class constructor throws some checked exception compulsorily child class constructor should throw the same checked exception or its parent exception.

1.7 version Enhancements
========================
 1. try with resource
 2. try with multicatch block

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

eg::
```
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 apporach
========================
1. Compulsorily the programmer is required to close all opened resources which increases the complexity of the program

2. Compulsorily we should write finally block explicitly, which increases the length of the code and reviews readablity.
   To Overcome this problem SUN MS introduced try with resources in "1.7" version of jdk.

try with resources
==========
In this apporach, 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 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 seperated 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 reference by default are treated as implicitly final and hence we can't perform reassignment with in 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. untill 1.6 version try should compulsorily be followed by either catch or finally, but from
   1.7 version we can take only take try with resources without cath 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, eventhough we have multiple exception having same handling code we have to write a

seperate catch block for every exceptions, it increases the length of the code and reviews
readability.

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

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

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

In multicatch block,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::

```
try{

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

Output:CompileTime Error

Exception Propogation
=====================

With in a method, if an exception is raised and if that method does not handle that exception
then Exception object will be propogated to the caller method then caller method is responsible
to handle that exceptions, This process is called as "Exception Propogation".

ReThrowing an Exception
=======================

To convert one exception type to another exception type, we can use rethrowing exception concept.

eg::

```
public class TestApp
{
```

```
      public static void main(String[] args)
      {
        try{
            System.out.println(10/0);
           }catch( ArithmeticException  e){
            throw new NullPointerException();
           }
      }
}
Output::
Exception in thread "main" java.lang.NullPointerException
        at TestApp.main(TestApp.java:10)


new vs newInstance()
====================
 => new is an operator to create an objects, if we know classname at the
begining then we can
    create an object using new operator.
 => newInstance() is a method present in class called "Class", which can
be used to create
    Object.
 => If we dont' know the class name at the begining and its available
dynamically RunTime then
    we should go for newInstance() method.

eg#1.
 class Demo{
      Demo(){
            System.out.println("Zero argument constructor");
      }
 }
 public class Test{
      public static void main(String... args){
            Object o=Class.forName(arg[0]).newInstance();
            System.out.println(o.getClass().getName());
      }
  }
 => If dynamically provide class name is not available then we will get
the RuntimeException
    saying "ClassNotFoundException".
 => To use newInstance() method compulsorily correspoding class should
contains no argument
    constructor, otherwise we will get the RuntimeException saying
"InstantiationException".
 => If the constructor exists with Zero argument and if the access
modifier is private then
    it would reslut in IllegalAccessException.

Difference b/w new and newInstance()
new => It is an operator which is used to create Object
       If we know the name of the class at the begining then we should
use new operator.
                  Test t=new Test();
       If the corresponding .class is not available at the RunTime then
we will get
       RuntimeException saying "NoClassDefFoundError", it is
"uncheckedException".
       To use new Operator the correspoding class not required to contain
no argument constructor.
```

```
newInstance() => It is an method present in java.lang.Class which is used
to create an Object.
                 If we don't know the className at the begining and
available dynamically at
                 the runtime.
                     Object o=Class.forName("Test").newInstance();
                 If the corresponding .class is not available at the
RunTime then we will get
                 RuntimeException saying "ClassNotFoundException", it is
"checkedException".
               To use newInstance() the corresponding class should
compulsorily contain no
               argument constructor, otherwise we will get
RuntimeException saying
                "InstantiationException".


Difference b/w ClassNotFoundException and NoClassDefFoundError
 ClassNotFoundException =>It is a checkedException
                          For dynamically provided class name at the
Runtime, if the Correspoding
                          .class files is not available then we will get
RuntimeException saying
                          "ClassNotFoundException".
                       eg:: Object o
=Class.forName(args[0]).newInstance();

 NoClassDefFoundError   => It is a uncheckedException.
                           For HardCoded class names, if the Correspoding
.class files is not                                  available then we
will get RuntimeException saying
                           "NoClassDefFoundError".
                        eg:: Test t=new Test();


Difference b/w instanceof vs isInstance()
instanceof => It is an operator which can be used to check whether the
given object is of
              particular type or not.
              We know at the type of begining it is available.
           eg:: String s=new String("sachin");
                System.out.println(s instanceof Object);//true

isInstance() =>It is an method present in java.lang.Class, we can use
isInstance() method to check
               whether the given object is of particular type or not
where type at begining is
               available Dynamically at RunTime.

eg#1.
class Demo{

}
public class TestApp {
      public static void main(String[] args) throws Exception{
        Demo d=new Demo();
        System.out.println(Class.forName(args[0]).isInstance(d));
      }
}
```

```
java TestApp Demo //true
java TestApp java.lang.String //false
java TestApp java.lang.Object //true
```

Describe the behaviour of following exceptions?

A. RunTimeException
B. Error
C. IOException
D. Exception
E. InterruptedException
F. Throwable
G. ArithemeticException
H. NullPointerException
I. FileNotFoundException

==========
Control flow in try catch
==========

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

Case 1:: If there is not exception.

Case 2:: if an exception raised at statement2 and corresponding catch block matched.

Case 3:: if any exception raised at statement2 but the corresponding catch block
not matched.

Case 4:: if an exception raised at statement 4 or statement 5.

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 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 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 statement5 and corresponding inner catch block is matched
Case5: If an exception is raised at statement5 and inner catch has not matched but 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 corresponding catch block is not matched.
Case11: If an exception is raised at statement 9 and corresponding catch block is matched.
Case12: If an exception is raised at statement 9 and 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

Various possible cases of Exception
==========

```
1.try{

  }catch(X e){

  }
```

```
2. try{
```

```
   }catch(X e){

   }catch(Y e){

   }
```

3.
```
   try{

   }catch( X e){

   }catch( X e){

   }
```

4.
```
   try{

   }finally{

   }
```

5.
```
   try{

   }catch(X e){

   }finally{

   }
```

6.
```
try{}
```

7.
```
catch(){}
```

8.
```
finally{}
```

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

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

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

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

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

14.
```
try{}
 catch(X e){}
```

```
    finally{}
    finally{}

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

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

    }

17. try{
      try{}
    }

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

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

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

throw keyword
=============
Case 1:
 class Test{
      public static void main(String... args){
            System.out.println(10/0);
      }
 }
            VS
class Test{
      public static void main(String... args){
            throw new ArithmeticException("/by Zero");
      }
}

Case2:
  class Test{
      static ArithmeticException e =new ArithmeticException();
      public static void main(String... args){
            throw e;
      }
  }
               VS
 class Test{
      static ArithmeticException e;
```

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

Case3:
  class Test{
      public static void main(String... args){
            System.out.println(10/0);
            System.out.println("hello");
      }
 }
            VS
 class Test{
  public static void main(String... args){
            throw new ArithmeticExeption("/ by zero");
            System.out.println("hello");
  }
}

Case4:
 class Test3{
      public static void main(String... args){
            throw new Test3();
      }
 }
                  VS
 public class Test3 extends RunTimeException{
      public static void main(String... args){
            throw new Test3();
      }
 }



throws
======
Case1:
import java.io.*;
class Test3{
      public static void main(String... args){
            PrintWriter pw=new PrintWriter("abc.txt");
            pw.println("Hello world");
      }
}

Case2:
class Test3{
      public static void main(String... args){
            Thread.sleep(3000);
      }
}
Solution
 1A.
  class Test3{
      public static void main(String... args){
            try{
                  Thread.sleep(5000);
            }catch(InterruptedException ie){}
      }
```

```
 }

1B.
 class Test{
      pubilc static void main(String... args) throws
InterruptedException{
            Thread.sleep(5000);
      }
 }

Case2:
 class Test{
      public static void main(String... args) throws InteruptedException{
            doWork();
      }
      pubilc static void doWork() throws InteruptedException{
            doMoreWork();
      }
      public static void doMoreWork() throws InteruptedException{
            Thread.sleep(5000);
      }
 }

Case studies of Throwable
=========================
Case 1::
class Test3{
      public static void main(String... args)throws Test3{

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

      }
}


Case2::
public class Test3 {
      public static void main(String... args) {
            throw new Exception();
      }
}
                  VS

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

Case3::

eg#1.
 public class Test3 {
      public static void main(String... args) {
            try
            {
```

```
                System.out.println("hiee");
            }
            catch (Exception e)
            {

            }
      }
}
```

eg#2.

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

            }
      }
}
```

eg#3.

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

            try
            {
                System.out.println("hiee");
            }
            catch (java.io.FileNotFoundException e)
            {

            }
      }
}
```

eg#4.

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

            try
            {
                System.out.println("hiee");
            }
            catch (InteruptedException e)
            {

            }
      }
}
```

eg#5.

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

		try
		{
			System.out.println("hiee");
		}
		catch (Error e)
		{

		}
	}
}

Case4::
class Test throws Exception//invalid
{
	Test() throws Exception{//valid

	}
	methodOne() throws Exception{//valid

	}
}
```

Exception handling keywords summary
==========
1. try => maintain risky code
2. catch=> maintain handling code
3. finally=> maintain cleanup code
4. throw => To hanover the created exception object to JVM manually
5. throws=> To delegate the Exception object from called method to caller method.

Various compile time errors in ExceptionHandling
==========
1. Exception XXX is already caught
2. Unreported Exception XXX must be caught or declared to be thrown.
3. Exception XXX is never thrown in the body of corresponding try statement.
4. try without catch,finally
5. catch without try
6. finally without try
7. incompatible types :found xxx required:Throwable
8. unreachable code.

Rules w.r.t Overriding
==========
parent: public void methodOne() throws Exception{}
child : public void methodOne()

parent: public void methodOne(){}
child : public void methodOne() throws Exception{}

parent: public void methodOne()throws Exception{}

```
child : public void methodOne()throws Exception{}


parent: public void methodOne()throws IOException{}
child : public void methodOne()throws IOException{}


parent: public void methodOne()throws IOException{}
child : public void methodOne()throws
FileNotFoundException,EOFException{}


parent: public void methodOne()throws IOException{}
child : public void methodOne()throws
FileNotFoundException,InterruptedException{}


parent: public void methodOne()throws IOException{}
child : public void methodOne()throws
FileNotFoundException,ArithmeticException{}


parent: public void methodOne()
child : public void methodOne()throws
ArithmeticException,NullPointerException,RuntimeException{}
```