Exception Handling

Topics Covered Under This Chapter:

- Introduction
- Runtime Stack Mechanism
- Exception Hierarchy and Types
- Handling Exceptions
- try with resource

1.. Introduction

- An unexpected or unwanted event that disturbs the normal flow of Program execution is known as an Exception.
- It is highly recommended to handle the exceptions in Software Applications. The main objective of Exception Handling is to terminate the application gracefully if an exception is encountered.
- Exception Handling doesn't mean repairing an exception. We must provide an alternative way to continue the rest of the program normally instead of exiting the program abnormally.
- Example – We are trying to perform read operation on a remote File which should be present in London Server. Suppose the specified file is not present in the System, then in that case we will get FileNotFoundException if an exception is not handled properly.

```
class ExceptionDemo {
    public static void main(String[] args) {
        //Try to Read the Remote File Which Doesn't Exists
        //Throws FileNotFoundException
    }
}
```

- We should handle exceptions in such a way that we provide some alternate implementation, that is if file is not present in London Server, use the Local file and continue the operation. This alternative approach is nothing but an exception handling.

```
class ExceptionDemo {
    public static void main(String[] args) {
        try {
            //Try to Read the Remote File Which Doesn't Exists
        }
        catch(FileNotFoundException e)
        {
           //Use Local File for Read Operation
        }
    }
}
```

2.. Runtime Stack Mechanism

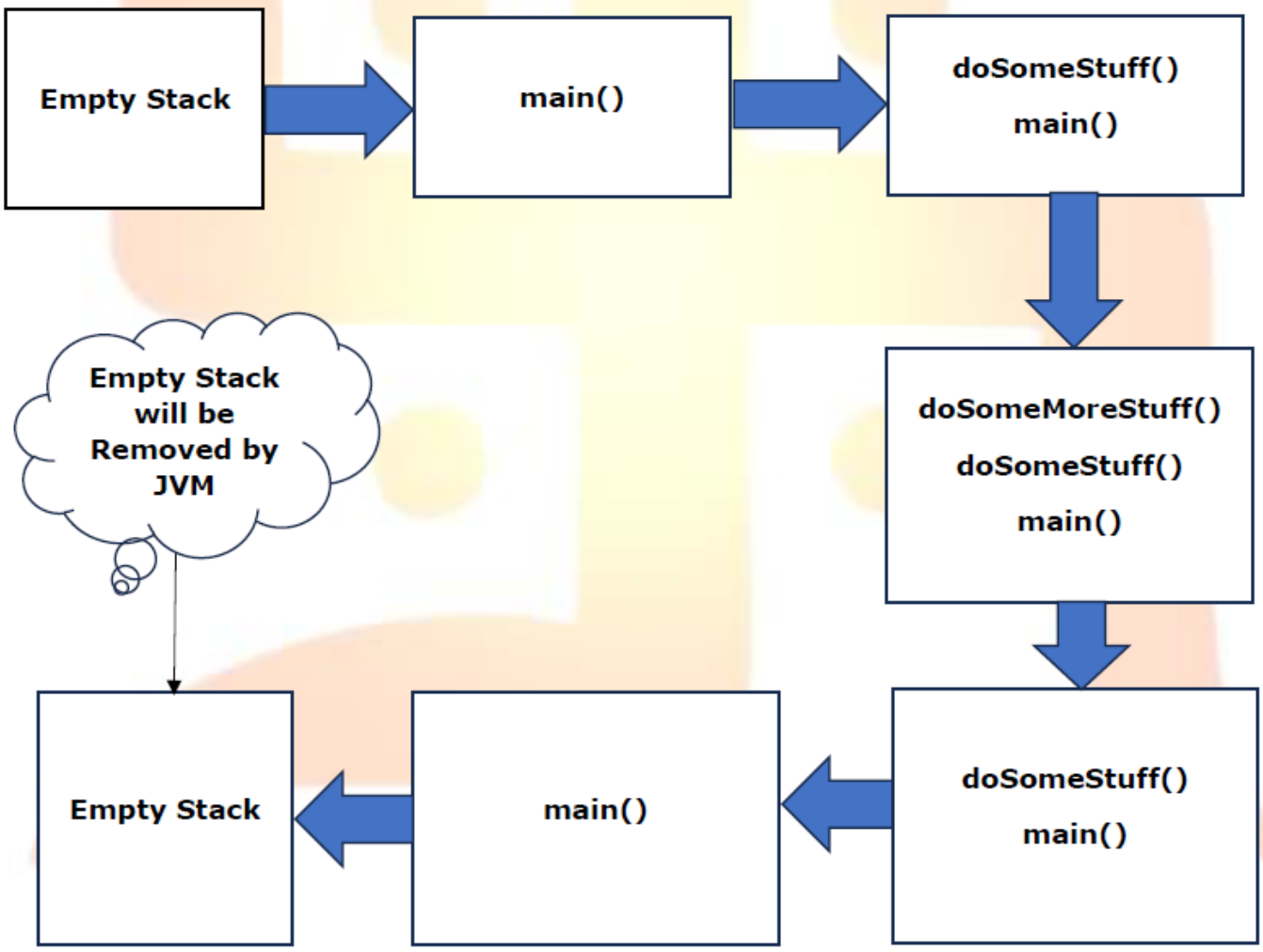
- For every thread JVM will create an empty Run time Stack.
- Each method performed by the thread will be stored in that Stack.
- Each entry in the stack is called Activation Record or Stack Frame.
- After completing every method call, the corresponding entry from the stack will be removed.
- After completing all method Calls, the stack will become empty, and that empty stack will be destroyed by JVM just before terminating the Thread.

Consider the below Program Example.

```
1  class RunTimeStackDemo {
2      public static void main(String[] args)
3      {
4          doSomeStuff();
5      }
6      public static void doSomeStuff()
7      {
8          doSomeMoreStuff();
9      }
10     public static void doSomeMoreStuff()
11     {
12         System.out.println("Hello");
13     }
14 }
```

```
java -cp /tmp/ZLvs1ZdfHD RunTimeStackDemo
Hello
```

As per Runtime Stack Mechanism, Method Calls Entries in Stack will appear as:



This will occur only when each of the execution will be completed normally and no any exceptions was encountered. Consider a Situation when any exception Occurs in Between, and this is called Default Exception Handling.

2.1 Default Exception Handling

Consider the Same Example as above Now where instead of Hello, we are trying to print 10/0 which will definitely throw ArithmeticException.

```
1  class RunTimeStackDemo {
2      public static void main(String[] args)
3      {
4          doSomeStuff();
5      }
6      public static void doSomeStuff()
7      {
8          doSomeMoreStuff();
9      }
10     public static void doSomeMoreStuff()
11     {
12         System.out.println(10/0);
13     }
14 }
```

```
java -cp /tmp/ZLvs1ZdfHD RunTimeStackDemo
Exception in thread "main" java.lang.ArithmeticException: / by zero
at RunTimeStackDemo.doSomeMoreStuff(RunTimeStackDemo.java:12)
    at RunTimeStackDemo.doSomeStuff(RunTimeStackDemo.java:8)
at RunTimeStackDemo.main(RunTimeStackDemo.java:4)
```

Now in this Case this is how the runtime stack mechanism Works:

- Inside a method if any exception occurs then the method in which exception is raised is responsible for creating the exception object by including the following properties and provide this created Object to JVM:
    a) Name of Exception
    b) Description of Exception
    c) Location at which Exception Occurs (Stack Trace).
- JVM will check whether the method contains any exception handling code or not.
- If the method doesn't contain any exception handling code, then JVM will terminate that method abnormally and remove the corresponding entry from the stack.
- JVM will check for the exception handling code in the caller method from where the method is called. If the caller method also doesn't contain any exception handling code, then JVM will terminate that method abnormally and removes the corresponding entry from the stack. This method will be repeated until main() method.
- If main() method also doesn't contains any exception handler code then JVM will terminate the main() also abnormally and remove the corresponding entry from the stack.
- Now JVM is responsible for handling such exceptions and for that JVM will call **Default Exception Handler** which will be responsible for printing the message on the Console and terminates the program abnormally.

The message will be in below Format:

Exception in thread "<Thread_Name>" <Exception_Name>: <Description>

<Stack Trace>

Stack Trace will change depending on the method where exception Occurs. Let's change the code which throws exception among different methods and see how the stack trace is getting printed by **Default Exception Handler**.

```java
class RunTimeStackDemo {
    public static void main(String[] args)
    {
        doSomeStuff();
    }
    public static void doSomeStuff()
    {
        doSomeMoreStuff();
        System.out.println(10/0);
    }
    public static void doSomeMoreStuff()
    {
        System.out.println("Hi");
    }
}
```

```
java -cp /tmp/ZLvs1ZdfHD RunTimeStackDemo
Hi
Exception in thread "main" java.lang.ArithmeticException: / by zero
at RunTimeStackDemo.doSomeStuff(RunTimeStackDemo.java:9)
at RunTimeStackDemo.main(RunTimeStackDemo.java:4)
```

```java
class RunTimeStackDemo {
    public static void main(String[] args)
    {
        doSomeStuff();
        System.out.println(10/0);
    }
    public static void doSomeStuff()
    {
        doSomeMoreStuff();
        System.out.println("Hello");
    }
    public static void doSomeMoreStuff()
    {
        System.out.println("Hi");
    }
}
```

```
java -cp /tmp/ZLvs1ZdfHD RunTimeStackDemo
Hi
Hello
Exception in thread "main" java.lang.ArithmeticException: / by zero
at RunTimeStackDemo.main(RunTimeStackDemo.java:5)
```

Note: If in a program any of the method terminates abnormally then that complete program is said to be terminated abnormally while if all the methods are terminated normally then that complete program is said to be terminated normally.

3.. Exception Hierarchy and Types

- **Throwable** Class is the Superclass of every Exceptions Class in Java.
- Throwable Class consists of two Sub classes:
    1. Exceptions
    2. Errors

3.1 Differences between Exceptions and Errors

- Most of the times Exceptions are generated by our program itself and it can be recoverable in Nature while Errors are something which is not caused by our program and these are due to system resources, and they are non-recoverable in nature. Here recoverable means we can handle exceptions by using catch block to have some alternate implementations but if any error occurs we as a programmer we can't do anything.
- Example – FileNotFoundException is an Exception which will occur when specified file on which we are operating doesn't exist while **OutOfMemory** is an error which will occur because of lack of System resources not because of Programs.

Exception Class Can be Further Classified as:

- RunTimeException
    1. NullPointerException
    2. ArithmeticException
    3. ClassCastException
    4. IndexOutOfBoundsException
    5. ArrayIndexOutOfBoundsException
    6. StringIndexOutOfBoundsException
    7. IllegalArgumentException
    8. NumberFormatException
    9. etc.
- IOException
    1. FileNotFoundException
    2. InterruptedIOException
    3. EOFException
- ServletException
- InterruptedException
- RemoteException etc.

Error Class can be Further Classified as:

- VMError
    1. StackOverFlowError
    2. OutOfMemoryError
- AssertionError
    1. ExceptionInInitializerError

3.2 Checked and Unchecked Exceptions

- The exception which is checked by the Compiler so that the program can be run smoothly during runtime such exceptions are called **Checked Exceptions**. For Example, When We are writing any File Handling Code Then Compiler will check weather FileNotFoundException is handled properly or not by the Programmer, if it is not handled, compiler will throw a compile time error saying Unreported Exception.

Consider the below Example Program for Checked Exceptions.

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

```
ERROR!
javac /tmp/j2roYjtRSN/CheckedExceptionDemo.java
/tmp/j2roYjtRSN/CheckedExceptionDemo.java:4: error: unreported exception FileNotFoundException; must be
    caught or declared to be thrown
PrintWriter pw = new PrintWriter("abc.txt");
                 ^
1 error
```

- In our program, if there is any chance of Checked Exceptions to be raised, it is highly recommended to handle it by using **try-catch** block or by using **throws** keyword. Because, if we don't handle it then compiler will throw a compile time error to let the programmer know that there is a chance of exception to be raised at runtime so handle it properly.
- The exceptions which are not checked by the compiler are called **Unchecked Exceptions**. The best example of Unchecked Exception is **Arithmetic Exception**. Let's handle the **FileNotFoundException** in above code and introduce a code which will create **ArithmeticException** at the Runtime. Now our code will compile without any issues and we will get **ArithmeticException** at the runtime.

```
import java.io.*;
class CheckedExceptionDemo {
    public static void main(String[] args) throws FileNotFoundException {
        PrintWriter pw = new PrintWriter("abc.txt");
        pw.println("Hello");
        System.out.println(10/0);
    }
}

```

```
java -cp /tmp/j2roYjtRSN CheckedExceptionDemo
Exception in thread "main" java.lang.ArithmeticException: / by zero
at CheckedExceptionDemo.main(CheckedExceptionDemo.java:6)
```

- Weather it is Checked or Unchecked, every exception occurs at Run time only, there is no chance of occurring any exception at compile time.
- Runtime Exception and its Child Classes and also Error and its Child classes are Unchecked Exceptions. Example: ArithmeticException, NullPointerException etc. Except this remaining all are Checked Exceptions.
- Exception is Said to be fully checked if both parent and Its child are Checked e.g. IOException and InterruptedException.
- Exception is Said to be partially checked if and only if some of its child classes are checked while others are unchecked. Example – Exception, Throwable class. Child of Exception class IOException which is Fully Checked while RuntimeException is partially Checked.

4.. Handling Exceptions

- If we don't want JVM to terminate the program abnormally in case of any exception is raised and get it handled by the Default Exception Handler of Java then we should go for Customized Exception Handling.
- We do Customized Exception Handling either by try-catch block or by throwing exception by using throws keyword.
- Its highly recommended to handle the exception by either of the two mechanisms mentioned above.

Code Without try-catch Will led to Abnormal Termination. Consider a Below Code and Its Output.

```
class CodeWithoutTryCatch {
    public static void main(String[] args) {
        System.out.println("Hi!!");
        System.out.println(10/0);
        System.out.println("Hello!!");
    }
}
```

```
java -cp /tmp/gKTfmwJ7fa CodeWithoutTryCatch
Hi!!
Exception in thread "main" java.lang.ArithmeticException: / by zero
at CodeWithoutTryCatch.main(CodeWithoutTryCatch.java:4)
```

Let's handle the exception using try-catch block now. The risky code which might raise an exception should be written in try block i.e., in between try{} while the exception handler code should always be written in catch block. Let's rewrite the above program using try-catch.

```
class CodeWithoutTryCatch {
    public static void main(String[] args) {
        System.out.println("Hi!!");
        try{
            System.out.println(10/0);
        }catch(ArithmeticException e) {
            System.out.println("Arithmetic Exception Occurred !! So Changing 0 to 2");
            System.out.println(10/2);
        }
        System.out.println("Hello!!");
    }
}
```

```
java -cp /tmp/gKTfmwJ7fa CodeWithoutTryCatch
Hi!!
Arithmetic Exception Occurred !! So Changing 0 to 2
5
Hello!!
```

4.1 Control Flow in Try Catch

Consider the below Programming snippet.

```
class TryCatchFlow {
    public static void main(String[] args) {
        try {
            stmt1;
            stmt2;
            stmt3;
        }catch (X e)
        {
            stmt4;
        }
        stmt5;
    }
}
```

From this snippet, let's have the below Conclusions.

- If there is no exception raised within a try block then the statements within a catch block won't be executed and program will be terminated Normally. So, in this Case the Statements will be executed in sequence – stmt1 -> stmt2 -> stmt3 -> stmt5.
- If an exception is raised within a try block say at stmt2 then all the remaining statements will be skipped from execution after stmt2 and control will directly go to corresponding matching catch block and won't come back to try block. Hence, it is highly recommended to put only those statements which can raise exception in try

block rest of the statements should be outside the try block. So, in this Case the Statements will be executed in sequence – stmt1 -> stmt4 -> stmt5 and program will be terminated Normally.

- If an exception is raised within a try block say at stmt2 and corresponding matching catch block is not present then in that case program will be terminated abnormally by just executing stmt1.
- If an exception is raised at stmt4 or stmt5 then it is always an abnormal termination. In addition to try block, exception can be raised in catch block or finally block also which will lead to abnormal termination.

4.2 Methods to print Exception Information

- **e.printStackTrace()**
  Print the Complete Information about Exception which Includes Exception Name, Description and Location at which It occurs.

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

```
java -cp /tmp/pcaL1sniXv ExceptionMethod
java.lang.ArithmeticException: / by zero
at ExceptionMethod.main(ExceptionMethod.java:5)
```

- e.getMessage()
  Print the Description message of Exception.

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

```
java -cp /tmp/pcaL1sniXv ExceptionMethod
/ by zero
```

- e.toString() or System.out.println(e)
  Print the Name and Description of Exception but doesn't print Stack Trace Info.

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

```
java -cp /tmp/pcaL1sniXv ExceptionMethod
java.lang.ArithmeticException: / by zero
```

Note: Internally default Exception handler of JVM uses printStackTrace() method to print Exception on the Console.

4.3 try with multiple catch blocks

- Multiple exceptions can be raised within a Single try block but it can be only one exception at a time. So, it is highly recommended to catch all the Exceptions separately in a separate catch block instead of catching Exception class only one time. Because we don't know at a time which exception is going to raised.

So below is the Invalid programming practice.

```
class InvalidExceptionHandling {
    public static void main(String[] args) {
        try{
            //Code Raising SQLException, IOException, InterruptedException,ArithmeticException etc
        }
        catch(Exception e)
        {
            //Handling Code
        }
    }
}
```

Below is the valid Programming Practice for Above Invalid Code.

```
class ValidExceptionHandling {
    public static void main(String[] args) {
        try{
            //Code Raising SQLException, IOException, InterruptedException,ArithmeticException etc
        }
        catch(SQLException e){
            //SQL Handling Code
        }
        catch(IOException e){
            //IO Handling Code
        }
        catch(InterruptedException e){
            //Interruption Handling Code
        }
        catch(ArithmeticException e){
            //Arithmetic Exception Handling Code
        }
        catch(Exception e){
            //All other Exceptions Handling Code
        }
    }
}
```

If try with multiple catch blocks are there then the order of catch block is also very important. First we have to catch always Child Exception then Only we should go for Parent Exception. If we Catch first Parent Exception and Then Child Exception then we will get compile time error saying "**exception XXX has already been caught**".

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

```
ERROR!
javac /tmp/Ww1ARKL27s/ExceptionHandling.java
/tmp/Ww1ARKL27s/ExceptionHandling.java:9: error: exception ArithmeticException has already been caught
catch(ArithmeticException e){
      ^
1 error
```

Above issue can be resolved by first catching **ArithmeticException** and then the Parent class **Exception**.

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

```
java -cp /tmp/Mw1ARKL27s ExceptionHandling
/ by zero
```

We should not catch the same exception twice for the same try block, if we do so, we will get compile time error saying "**exception XXX has already been caught**".

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

```
ERROR!
javac /tmp/Mw1ARKL27s/ExceptionHandling.java
/tmp/Mw1ARKL27s/ExceptionHandling.java:9: error: exception ArithmeticException has already been caught
        catch(ArithmeticException e){
        ^
1 error
```

4.4 finally block (final vs finally vs finalize)

final

- final is a modifier which can be applied for Class, Methods and Variables.
- If a class is declared as final that class cannot be inherited.
- If a method is declared as final that method cannot be overridden.
- If a variable is declared as final that variable cannot be reassigned to any other value. I.e., Value got fixed.

finally

- finally is a block associated with try-catch exception handling mechanism.
- The cleanup code is always written inside a finally block.
- Weather an exception is raised or not, finally block is going to execute.
- If an exception is raised then finally block will be executed after catch block and if an exception is not raised then finally block will be executed after try block.

```
class ExceptionHandling {
    public static void main(String[] args) {
        try{
            //Risky Code
        }
        catch(Exception e){
            //Exception Handling Code
        }
        finally{
            //Cleanup Code
        }
    }
}
```

finalize

- finalize is a method which is called by Garbage Collector.
- Garbage Collector always calls finalize() method just before destroying Unused Objects to perform cleanup associated with that Unused Objects.

- Once finalize() method execution completes, then only Unused Objects gets destroyed by Garbage Collector.

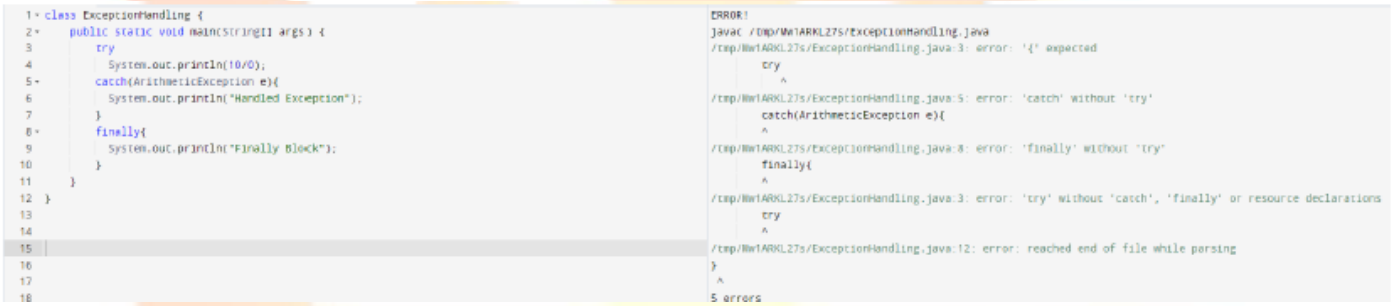
Note:

Though Both finally block and finalize() method are used to perform cleanup activities. But there is one difference between the two. finally block will perform cleanup of all those resources which got opened in a try block. Example – SQL Connection, Any File. While finalize() method will perform cleanup activities associated with the Unused Object. Example – Database associated with the Object etc.

4.5 Various Possible Combinations of try-catch-finally

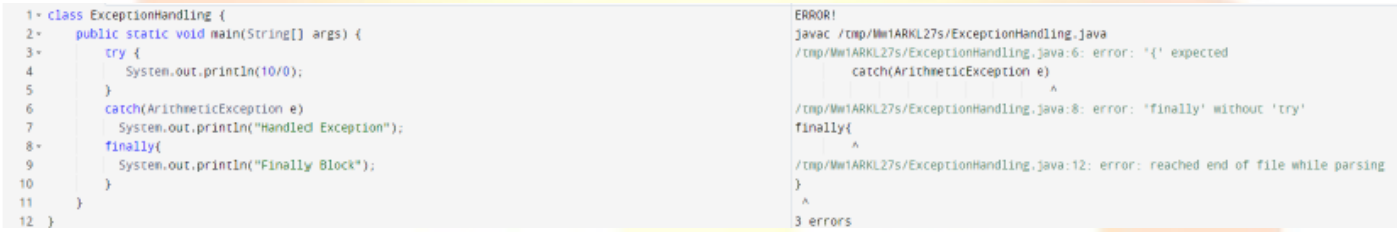
Case – 1 Mandatory Curly Braces

For try-catch-finally blocks Curly braces are mandatory. If we write any of try, catch or finally without curly braces we will get compile time error.



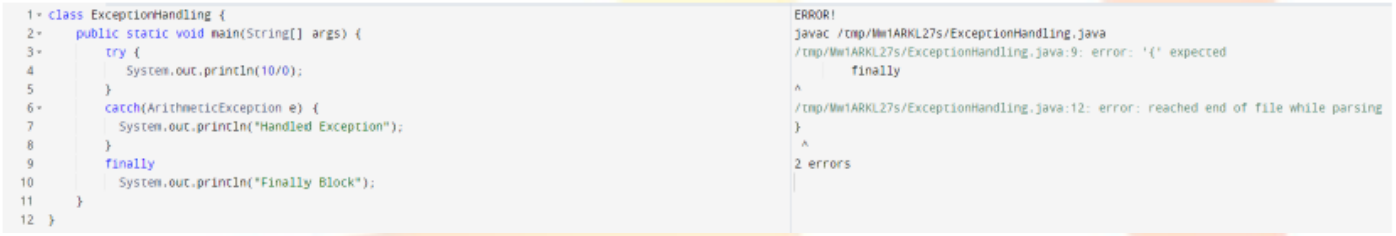
```
class ExceptionHandling {
    public static void main(String[] args) {
        try
          System.out.println(10/0);
        catch(ArithmeticException e){
          System.out.println("Handled Exception");
        }
        finally{
          System.out.println("Finally Block");
        }
    }
}
```

```
ERROR!
javac /tmp/Nw1ARKL27s/ExceptionHandling.java
/tmp/Nw1ARKL27s/ExceptionHandling.java:3: error: '{' expected
        try
          ^
/tmp/Nw1ARKL27s/ExceptionHandling.java:5: error: 'catch' without 'try'
        catch(ArithmeticException e){
        ^
/tmp/Nw1ARKL27s/ExceptionHandling.java:8: error: 'finally' without 'try'
        finally{
        ^
/tmp/Nw1ARKL27s/ExceptionHandling.java:3: error: 'try' without 'catch', 'finally' or resource declarations
        try
        ^
/tmp/Nw1ARKL27s/ExceptionHandling.java:12: error: reached end of file while parsing
}
 ^
5 errors
```



```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
          System.out.println(10/0);
        }
        catch(ArithmeticException e)
          System.out.println("Handled Exception");
        finally{
          System.out.println("Finally Block");
        }
    }
}
```

```
ERROR!
javac /tmp/Nw1ARKL27s/ExceptionHandling.java
/tmp/Nw1ARKL27s/ExceptionHandling.java:6: error: '{' expected
        catch(ArithmeticException e)
                                   ^
/tmp/Nw1ARKL27s/ExceptionHandling.java:8: error: 'finally' without 'try'
        finally{
        ^
/tmp/Nw1ARKL27s/ExceptionHandling.java:12: error: reached end of file while parsing
}
 ^
3 errors
```



```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
          System.out.println(10/0);
        }
        catch(ArithmeticException e) {
          System.out.println("Handled Exception");
        }
        finally
          System.out.println("Finally Block");
    }
}
```

```
ERROR!
javac /tmp/Nw1ARKL27s/ExceptionHandling.java
/tmp/Nw1ARKL27s/ExceptionHandling.java:9: error: '{' expected
        finally
^
/tmp/Nw1ARKL27s/ExceptionHandling.java:12: error: reached end of file while parsing
}
 ^
2 errors
```

Case – 2 Order of try-catch-finally block is important

The order of try-catch-finally block is important. We cannot have try-finally-catch or finally-try-catch combination. If we have so we will get Compile time error. Below Order of try-catch-finally is a Valid Combination.

```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
          System.out.println(10/0);
        }
        catch(ArithmeticException e) {
          System.out.println("Handled Exception");
        }
        finally {
          System.out.println("Finally Block");
        }
    }
}
```

```
java -cp /tmp/Nw1ARKL27s ExceptionHandling
Handled Exception
Finally Block
```

Below Order of try-catch-finally are invalid

```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
           System.out.println(10/0);
        }
        finally {
          System.out.println("Finally Block");
        }
        catch(ArithmeticException e) {
          System.out.println("Handled Exception");
        }

    }
}
```

```
ERROR!
javac /tmp/Ww1ARKL27s/ExceptionHandling.java
/tmp/Ww1ARKL27s/ExceptionHandling.java:9: error: 'catch' without 'try'
        catch(ArithmeticException e) {
        ^
1 error
```

```
class ExceptionHandling {
    public static void main(String[] args) {
        finally {
          System.out.println("Finally Block");
        }
        try {
           System.out.println(10/0);
        }
        catch(ArithmeticException e) {
          System.out.println("Handled Exception");
        }

    }
}
```

```
ERROR!
javac /tmp/Ww1ARKL27s/ExceptionHandling.java
/tmp/Ww1ARKL27s/ExceptionHandling.java:3: error: 'finally' without 'try'
finally {
        ^
1 error
```

```
class ExceptionHandling {
    public static void main(String[] args) {
        catch(ArithmeticException e) {
          System.out.println("Handled Exception");
        }
        try {
           System.out.println(10/0);
        }
        finally {
          System.out.println("Finally Block");
        }
    }
}
```

```
ERROR!
javac /tmp/Ww1ARKL27s/ExceptionHandling.java
/tmp/Ww1ARKL27s/ExceptionHandling.java:3: error: 'catch' without 'try'
catch(ArithmeticException e) {
        ^
1 error
```

Case – 3 Single try, catch and finally block gives compile time error.

Single try block

Single try block will give us compile time error. Whenever we are writing single try block then it is compulsory that we write either catch block or finally block.

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

```
ERROR!
javac /tmp/Ww1ARKL27s/ExceptionHandling.java
/tmp/Ww1ARKL27s/ExceptionHandling.java:3: error: 'try' without 'catch', 'finally' or resource declarations
try {
    ^
1 error
```

If we write try with only catch block then the Program will be terminated Normally with handled Exception.

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

```
java -cp /tmp/Ww1ARKL27s ExceptionHandling
Handled Exception
```

If we just write try with finally, exception will be handled by JVM by using Default exception Handler and program will be terminated Abnormally.

```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
           System.out.println(10/0);
        }
        finally {
          System.out.println("Finally Block");
        }
    }
}
```

```
java -cp /tmp/Nw1ARKL27s ExceptionHandling
Finally Block
Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExceptionHandling.main(ExceptionHandling.java:4)
```

Single catch block

Whenever we are writing catch block then compulsory associated try block is required.

```
class ExceptionHandling {
    public static void main(String[] args) {
        catch(ArithmeticException e) {
          System.out.println("Handled Exception");
        }
    }
}
```

```
ERROR!
javac /tmp/Nw1ARKL27s/ExceptionHandling.java
/tmp/Nw1ARKL27s/ExceptionHandling.java:3: error: 'catch' without 'try'
        catch(ArithmeticException e) {
        ^
1 error
```

Single finally block

Whenever we are writing finally block then compulsory associated try block is required.

```
class ExceptionHandling {
    public static void main(String[] args) {
        finally {
          System.out.println("Finally Block");
        }
    }
}
```

```
ERROR!
javac /tmp/Nw1ARKL27s/ExceptionHandling.java
/tmp/Nw1ARKL27s/ExceptionHandling.java:3: error: 'finally' without 'try'
        finally {
^
1 error
```

Case – 4 Nested try-catch-finally is Possible

Inside try, catch and finally block, we can have nested try-catch-finally blocks without any issue.

```
class ExceptionHandling {
    public static void main(String[] args) {
        try
        {
           System.out.println(10/2);
           try {
              System.out.println(20/0);
           }catch(ArithmeticException e) {
                System.out.println("Nested Handled Exception");
           }finally {
               System.out.println("Nested Finally Block");
           }
        }
        catch(ArithmeticException e) {
          System.out.println("Handled Exception");
        }
        finally {
          System.out.println("Finally Block");
        }
    }
}
```

```
java -cp /tmp/Nw1ARKL27s ExceptionHandling
5
Nested Handled Exception
Nested Finally Block
Finally Block
```

```
class ExceptionHandling {
    public static void main(String[] args) {
        try
        {
           System.out.println(10/0);
        }
        catch(ArithmeticException e)
        {
          System.out.println("Handled Exception");
          try {
              System.out.println(20/0);
           }catch(ArithmeticException e1) {
                System.out.println("Nested Handled Exception");
           }finally {
               System.out.println("Nested Finally Block");
           }
        }
        finally {
          System.out.println("Finally Block");
        }
    }
}
```

```
java -cp /tmp/Nw1ARKL27s ExceptionHandling
Handled Exception
Nested Handled Exception
Nested Finally Block
Finally Block
```

```java
class ExceptionHandling {
    public static void main(String[] args) {
        try
        {
           System.out.println(10/0);
        }
        catch(ArithmeticException e)
        {
          System.out.println("Handled Exception");
        }
        finally {
          System.out.println("Finally Block");
          try {
              System.out.println(20/0);
           }catch(ArithmeticException e1) {
                System.out.println("Nested Handled Exception");
           }finally {
               System.out.println("Nested Finally Block");
           }
        }
    }
}
```

```
java -cp /tmp/Ww1ARKL27s ExceptionHandling
Handled ExceptionFinally Block
Nested Handled Exception
Nested Finally Block
```

Case – 5 Introducing Statement between try-catch-finally block

If any statement is written between try and catch or between catch and finally will give us Compile time error. Because if we write any statement between try and catch it will make the compiler to think that we have written try without catch or finally and catch without try block and will give us error. Similarly, if we put any statement between catch and finally it will treat finally without try.

```java
class ExceptionHandling {
    public static void main(String[] args) {
        try
        {
           System.out.println(10/0);
        }
        catch(ArithmeticException e)
        {
          System.out.println("Handled Exception");
        }
        System.out.println("Hello Exception");
        finally {
          System.out.println("Finally Block");
        }
    }
}
```

```
ERROR!
javac /tmp/Ww1ARKL27s/ExceptionHandling.java
/tmp/Ww1ARKL27s/ExceptionHandling.java:12: error: 'finally' without 'try'
        finally {
        ^
1 error
```

```java
class ExceptionHandling {
    public static void main(String[] args) {
        try
        {
           System.out.println(10/0);
        }
        System.out.println("Hello Exception");
        catch(ArithmeticException e)
        {
          System.out.println("Handled Exception");
        }
        finally {
          System.out.println("Finally Block");
        }
    }
}
```

```
ERROR!
javac /tmp/Ww1ARKL27s/ExceptionHandling.java
/tmp/Ww1ARKL27s/ExceptionHandling.java:3: error: 'try' without 'catch', 'finally' or resource declarations
        try
        ^
/tmp/Ww1ARKL27s/ExceptionHandling.java:8: error: 'catch' without 'try'
        catch(ArithmeticException e)
        ^
/tmp/Ww1ARKL27s/ExceptionHandling.java:12: error: 'finally' without 'try'
        finally {
        ^
3 errors
```

4.6 throw Keyword

Sometimes we can create an Exception Object explicitly and we can hand over that Exception Object to JVM Manually by using **throw** keyword.

Example: throw new **ArithmeticException**("/ by zero")

Default Exception Handler of JVM Handles Exception in below Way:

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

```
java -cp /tmp/Ww1ARKL27s ExceptionHandling
Exception in thread "main" java.lang.ArithmeticException: / by zeroat ExceptionHandling.main
    (ExceptionHandling.java:3)
```

Same we can do by using **throw** keyword manually.

```
class ExceptionHandling {
    public static void main(String[] args) {
        throw new ArithmeticException("/ by zero");
    }
}
```

```
java -cp /tmp/Ww1ARKL27s ExceptionHandling
Exception in thread "main" java.lang.ArithmeticException: / by zeroat ExceptionHandling.main
    (ExceptionHandling.java:3)
```

Note:

Best Use of throw keyword is for User defined Exception or Customized Exceptions But not for predefined Exceptions.

Use Cases related to throw keyword

Case 1:

In statement throw e, if e refers to any Exception Object, then we will get that Exception in the Console.

```
class ExceptionHandling {
    static ArithmeticException e = new ArithmeticException();
    public static void main(String[] args) {
        throw e;
    }
}
```

```
java -cp /tmp/Ww1ARKL27s ExceptionHandling
Exception in thread "main" java.lang.ArithmeticException
at ExceptionHandling.<clinit>(ExceptionHandling.java:2)
```

In statement throw e, if e refers null then we will get null pointer exception on the Console.

```
class ExceptionHandling {
    static ArithmeticException e;
    public static void main(String[] args) {
        throw e;
    }
}
```

```
java -cp /tmp/Ww1ARKL27s ExceptionHandling
Exception in thread "main" java.lang.NullPointerException: Cannot throw exception because "ExceptionHandling
    .e" is null
    at ExceptionHandling.main(ExceptionHandling.java:4)
```

Case 2:

After throw statement we cannot write any other statement in the same block if we do so we will get compile time error.

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

```
ERROR!
javac /tmp/Ww1ARKL27s/ExceptionHandling.java
/tmp/Ww1ARKL27s/ExceptionHandling.java:4: error: unreachable statement
        System.out.println("Hello");
        ^
1 error
```

Case 3:

We can throw only throwable objects, exceptions and errors. We cannot throw any normal Objects if we do so we will get compile time error.

```
class ExceptionHandling {
    public static void main(String[] args) {
        throw new ExceptionHandling();
    }
}
```

```
ERROR!
javac /tmp/LEeuPKTWST/ExceptionHandling.java
/tmp/LEeuPKTWST/ExceptionHandling.java:3: error: incompatible types: ExceptionHandling cannot be converted to
    Throwable
        throw new ExceptionHandling();
^
1 error
```

If we make any class throwable then we can throw it by using throw keyword.

```
class Test extends RuntimeException {
    public static void main(String[] args) {
        throw new Test();
    }
}
```

```
java -cp /tmp/vRS1CLyYQe Test
Exception in thread "main" Test
at Test.main(Test.java:3)
```

4.7 throws Keyword

As we discussed earlier, we can handle any exception either by using try-catch block or by throws keyword. We have already seen how to handle exception using try-catch. Let's now see how to handle any exceptions using throws keyword.

We can use throws keyword for exception handling if we don't want to handle exception by our own by using try-catch but rather wants our caller method to handle the exception. Here caller method can be any method or the JVM itself.

```
class Test extends RuntimeException {
    public static void main(String[] args) throws InterruptedException {
        Thread.sleep(10000);
        System.out.println("Hello World");
    }
}
```

```
java -cp /tmp/vRSiCLyYQe Test
Hello World
```

The **throws** keyword requires only for checked Exceptions and usage of **throws** keyword for unchecked Exceptions there is no use or impact. The use of **throws** keyword is just to convince compiler and avoid compile time error, **throws** keyword is not going to prevent the abnormal termination of the program. So, it is highly recommended to use only try-catch for exception handling and try to avoid the use of throws keyword.

```
class ThrowsDemo {
    public static void main(String[] args) {
        doStuff();
    }
    public static void doStuff() {
        doMoreStuff();
    }
    public static void doMoreStuff() {
        Thread.sleep(5000);
        System.out.println("Hey!!");
    }
}
```

```
ERROR!
javac /tmp/3OtvCw9bX6/ThrowsDemo.java
/tmp/3OtvCw9bX6/ThrowsDemo.java:9: error: unreported exception InterruptedException; must be caught or
    declared to be thrown
        Thread.sleep(5000);
                     ^
1 error
```

```
class ThrowsDemo {
    public static void main(String[] args) {
        doStuff();
    }
    public static void doStuff() {
        doMoreStuff();
    }
    public static void doMoreStuff() throws InterruptedException {
        Thread.sleep(5000);
        System.out.println("Hey!!");
    }
}
```

```
ERROR!
javac /tmp/3OtvCw9bX6/ThrowsDemo.java
/tmp/3OtvCw9bX6/ThrowsDemo.java:6: error: unreported exception InterruptedException; must be caught or
    declared to be thrown
doMoreStuff();
             ^
1 error
```

```
class ThrowsDemo {
    public static void main(String[] args) {
        doStuff();
    }
    public static void doStuff() throws InterruptedException {
        doMoreStuff();
    }
    public static void doMoreStuff() throws InterruptedException {
        Thread.sleep(5000);
        System.out.println("Hey!!");
    }
}
```

```
ERROR!
javac /tmp/3OtvCw9bX6/ThrowsDemo.java
/tmp/3OtvCw9bX6/ThrowsDemo.java:3: error: unreported exception InterruptedException; must be caught or
    declared to be thrown
        doStuff();
                 ^
1 error
```

```
class ThrowsDemo {
    public static void main(String[] args) throws InterruptedException {
        doStuff();
    }
    public static void doStuff() throws InterruptedException {
        doMoreStuff();
    }
    public static void doMoreStuff() throws InterruptedException {
        Thread.sleep(5000);
        System.out.println("Hey!!");
    }
}
```

```
java -cp /tmp/3OtvCw9bX6 ThrowsDemo
Hey!!
```

In above program, if we remove any of the **throws** keyword then program won't compile and will get the same error.

Note: We can use throws keyword for Constructors and Methods but not for class. If we do so we will get Compile Time Error.

```java
class Test
{
    Test() throws Exception
    {

    }

    public void m1() throws Exception
    {

    }
}

class ThrowsDemo {
    public static void main(String[] args)
    {

    }
}
```

```
java -cp /tmp/3OtvCw9bX6 ThrowsDemo
```

```java
class Test throws Exception
{
    Test() throws Exception
    {

    }

    public void m1() throws Exception
    {

    }
}

class ThrowsDemo {
    public static void main(String[] args)
    {

    }
}
```

```
ERROR!
javac /tmp/3OtvCw9bX6/ThrowsDemo.java
/tmp/3OtvCw9bX6/ThrowsDemo.java:1: error: '{' expected
class Test throws Exception
           ^
1 error
```

Note: Just like **throw**, **throws** also can be used to throw only **Throwable** Objects.

4.8 Customization Exceptions

Sometimes to meet programming requirements we can define our own Exceptions and Such type of Exceptions are Called **Customized** or **User-Defined Exceptions**. Example – InsufficientBankBalanceException.

Every Exception is a Java Class and to create User-defined Exception Just we need to Create a new User Defined Exception class which should extends to **RuntimeException**.

```java
class InsufficientBalanceException extends RuntimeException
{
    InsufficientBalanceException(String s) {
        super(s);
    }
}
class CustomizedExceptionDemo {
    public static void main(String[] args) {
        double balance = 1000;
        double amountToWithdraw = 2000;
        if(amountToWithdraw > balance) {
            throw new InsufficientBalanceException("Insufficient Balance!! Please deposit Money!!");
        }
        else {
            System.out.println("Money Withdrawn Successfully : " + amountToWithdraw);
        }
    }
}
```

```
java -cp /tmp/3OtvCw9bX6 CustomizedExceptionDemo
Exception in thread "main" InsufficientBalanceException: Insufficient Balance!! Please deposit Money!!
at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:12)
```

```java
class InsufficientBalanceException extends RuntimeException
{
    InsufficientBalanceException(String s) {
        super(s);
    }
}
class CustomizedExceptionDemo {
    public static void main(String[] args) {
        double balance = 1000;
        double amountToWithdraw = 500;
        if(amountToWithdraw > balance) {
            throw new InsufficientBalanceException("Insufficient Balance!! Please deposit Money!!");
        }
        else {
            System.out.println("Money Withdrawn Successfully : " + amountToWithdraw);
        }
    }
}
```

```
java -cp /tmp/3OtvCw9bX6 CustomizedExceptionDemo
Money Withdrawn Successfully : 500.0
```

It is highly recommended to Create Customized Exceptions as Unchecked Exception.

Note: We write super(s) inside the constructor to make our Exception description available to Default Exception Handler of Throwable Class which can be printed on the console.

5.. try with resource and Multi Catch Block

- try with resource and Multi Catch Block was introduced in Java 1.7 Version.

5.1 try with resource

Prior to Java Version 1.7 whatever resource we opened in a Try block was mandatory to close in finally block to avoid wastage of resource. Consider a below Example:

```
1  import java.io.*;
2  class TryWithoutResource
3  {
4      public static void main(String[] args)
5      {
6          BufferedReader br = null;
7          try
8          {
9              br = new BufferedReader(new FileReader("Demo.txt"));
10             System.out.println("Hello, World!");
11         }
12         catch(IOException e)
13         {
14             e.printStackTrace();
15         }
16         finally
17         {
18             if(br != null)
19             {
20                 br.close();
21             }
22         }
23     }
24 }
```

But from Java version 1.7 try with resource feature was introduced. With this new feature we are not required to close any resource in the finally block. The above example can be written with try with resource as:

```
1  import java.io.*;
2  class TryWithResource
3  {
4      public static void main(String[] args)
5      {
6          try(BufferedReader br = new BufferedReader(new FileReader("Demo.txt")))
7          {
8              System.out.println("Hello, World!");
9          }
10         catch(IOException e)
11         {
12             e.printStackTrace();
13         }
14     }
15 }
```

- In try with resource, resource will be closed once the control reaches the end of the try block either normally or abnormally.
- In try with resource, we can declare any number of Resources separated by ;.
  try(R1; R2; R3; R4)
  {}

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

- Whatever resources we takes inside try() should implements **AutoCloseable** Interface. All IO related resources, Database Related Resources, Network Related Resources already implements AutoCloseable Interface. Being a programmer, we are not required to close AutoCloseable Implemented resources. **AutoCloseable** Interface was also introduced in **Java Version 1.7** and contains only One Method which is **close**() method and it is present in java.lang package.

  public void close();

- Whatever Resource we declare in try with Resource is Implicitly final, we cannot reassign them to point to any other Resource in the Later part of the Program.

```
import java.io.*;
class TryWithResource
{
    public static void main(String[] args)
    {
        try(BufferedReader br = new BufferedReader(new FileReader("Demo.txt")))
        {
            br = new BufferedReader(new FileReader("DemoNew.txt"));
            System.out.println("Hello, World!");
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

```
ERROR!
javac /tmp/HpyCKYEKjB/TryWithResource.java
/tmp/HpyCKYEKjB/TryWithResource.java:8: error: auto-closeable resource br may not be assigned
            br = new BufferedReader(new FileReader("DemoNew.txt"));
            ^
1 error
```

- We cannot write try{} statement without finally or catch but we can write try(R) without finally and catch.

5.2 Multi Catch Blocks

Prior to Java 1.7 Version, we can have try block followed by Multiple Catch Block. But there was a catch. If multiple catch Blocks are having the same exception handler code we have to write them separately. We cannot club Them Together. Consider the Below Example:

```
import java.io.*;
class MultiCatchDemo
{
    public static void main(String[] args) {
        try{
            BufferedReader br = null;
            System.out.println(10/0);
        }catch(ArithmeticException e) {
            e.printStackTrace();
        }catch(NullPointerException e) {
            e.printStackTrace();
        }catch(IOException e) {
            System.out.println("Something Went Wrong");
        }catch(InterruptedException e) {
            System.out.println("Something Went Wrong");
        }
    }
}
```

From Java version 1.7 Onwards, we can club two catch blocks into One if both the catch Blocks are having the same handling code as per the requirement.

The above code can be written as:

```java
import java.io.*;
class MultiCatchDemo
{
    public static void main(String[] args) {
        try{
            BufferedReader br = null;
            System.out.println(10/0);
        }catch(ArithmeticException|NullPointerException e) {
            e.printStackTrace();
        }catch(IOException|InterruptedException e) {
            System.out.println("Something Went Wrong");
        }
    }
}
```

These two enhancements try with resource and multi catch block mainly reduce the size of the code programmer is going to write and also increases readability of the code.

```java
import java.io.*;
class MultiCatchDemo
{
    public static void main(String[] args) {
        try{
            System.out.println(10/0);
            String s = null;
            System.out.println(s.length());
        }catch(ArithmeticException|NullPointerException e) {
            e.printStackTrace();
        }
    }
}
```

```
java -cp /tmp/HpyCKYEKjB MultiCatchDemo
java.lang.ArithmeticException: / by zero
at MultiCatchDemo.main(MultiCatchDemo.java:6)
```

```java
import java.io.*;
class MultiCatchDemo
{
    public static void main(String[] args) {
        try{
            //System.out.println(10/0);
            String s = null;
            System.out.println(s.length());
        }catch(ArithmeticException|NullPointerException e) {
            e.printStackTrace();
        }
    }
}
```

```
java -cp /tmp/HpyCKYEKjB MultiCatchDemo
java.lang.NullPointerException: Cannot invoke "String.length()" because "<local1>" is nullat MultiCatchDemo
    .main(MultiCatchDemo.java:8)
```

In Multi catch Block, If we are taking Parent class Exception then Child Exception shouldn't be there else a compile time error will be raised.

```java
import java.io.*;
class MultiCatchDemo
{
    public static void main(String[] args) {
        try{
            System.out.println(10/0);
        }catch(ArithmeticException | Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
ERROR!
javac /tmp/HpyCKYEKjB/MultiCatchDemo.java
/tmp/HpyCKYEKjB/MultiCatchDemo.java:7: error: Alternatives in a multi-catch statement cannot be related by
    subclassing
        }catch(ArithmeticException | Exception e) {
                                     ^
Alternative ArithmeticException is a subclass of alternative Exception
1 error
```