Exception Handling in Java

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

What is Exception in Java

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Why an exception occurs?

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

What is Exception Handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc. If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user.

Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

statement 8;

statement 9;

statement 10;

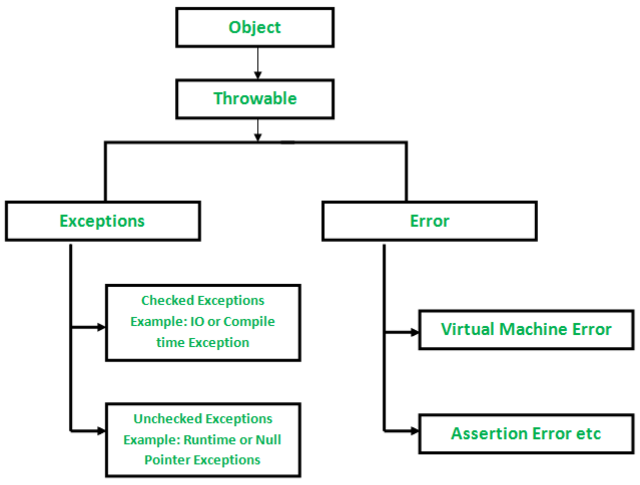
Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Error vs Exception

Error: An Error indicates serious problem that a reasonable application should not try to catch. Exception: Exception indicates conditions that a reasonable application might try to catch.

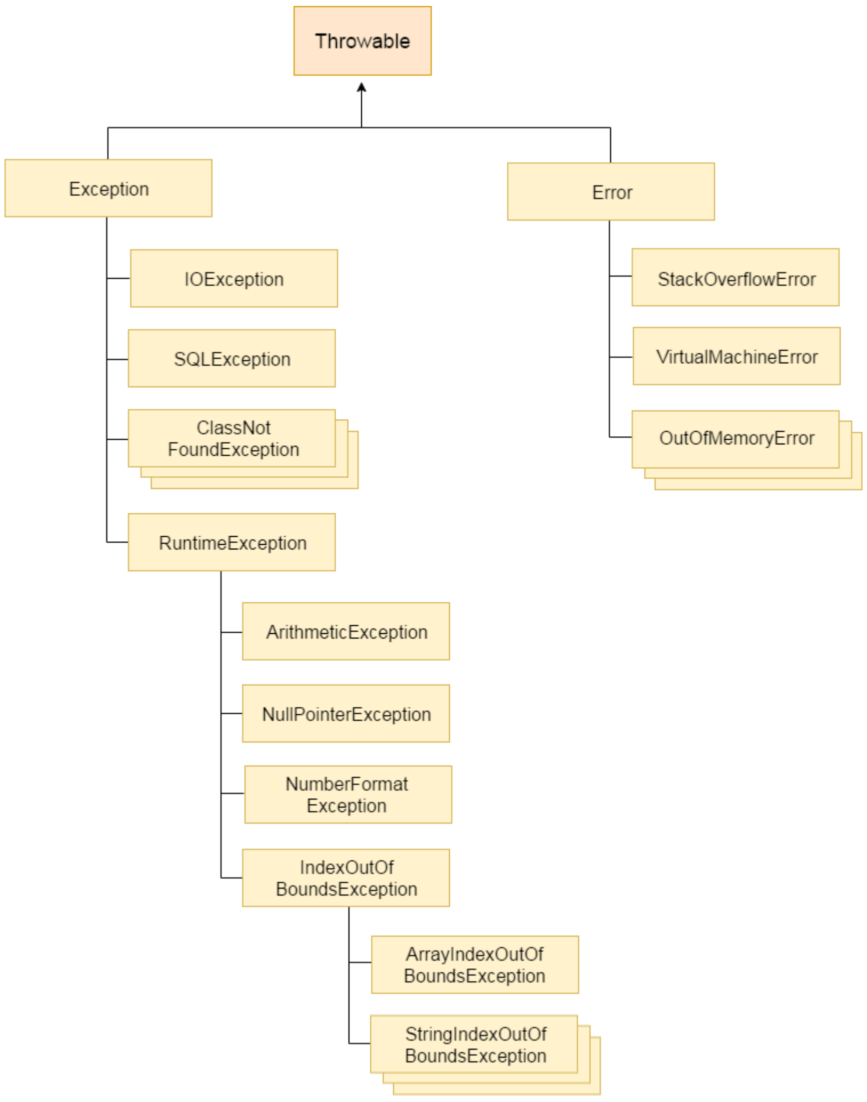
Exception Hierarchy

All exception and errors types are sub classes of class Throwable, which is base class of hierarchy.One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception.Another branch,Error are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

- Checked Exception
- Unchecked Exception
- Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc. The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.

Example

In this example we are reading the file myfile.txt and displaying its content on the screen. In this program there are three places where a checked exception is thrown as mentioned in the comments below. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException.

```
import java.io.*;
class Example {
  public static void main(String args[])
  {
          FileInputStream fis = null;
          /*This constructor FileInputStream(File filename)
           * throws FileNotFoundException which is a checked
           * exception
       */
      fis = new FileInputStream("B:/myfile.txt");
          int k;

          /* Method read() of FileInputStream class also throws
           * a checked exception: IOException
       */
          while(( k = fis.read() ) != -1)
          {
                  System.out.print((char)k);
          }
```

```
        /*The method close() closes the file input stream
         * It throws IOException*/
        fis.close();
  }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:

Unhandled exception type FileNotFoundException

Unhandled exception type IOException

Unhandled exception type IOException
```

Why this compilation error? As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn’t handled/declared the exceptions, our program gave the compilation error.

How to resolve the error? There are two ways to avoid this error. We will see both the ways one by one.

Method 1: Declare the exception using throws keyword.

As we know that all three occurrences of checked exceptions are inside main() method so one way to avoid the compilation error is: Declare the exception in the method using throws keyword. You may be thinking that our code is throwing FileNotFoundException and IOException both then why we are declaring the IOException alone. The reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare them like this public static void main(String args[]) throws IOException, FileNotFoundException.

```
import java.io.*;
class Example {
  public static void main(String args[]) throws IOException
  {
    FileInputStream fis = null;
    fis = new FileInputStream("B:/myfile.txt");
    int k;
```

```
      while(( k = fis.read() ) != -1)
      {
              System.out.print((char)k);
      }
      fis.close();
   }
}
```

Output:

File content is displayed on the screen.

Method 2: Handle them using try-catch blocks.

The approach we have used above is not good at all. It is not the best exception handling practice. You should give meaningful message for each exception type so that it would be easy for someone to understand the error. The code should be like this:

```
import java.io.*;
class Example {
   public static void main(String args[])
   {
         FileInputStream fis = null;
         try{
            fis = new FileInputStream("B:/myfile.txt");
         }catch(FileNotFoundException fnfe){
        System.out.println("The specified file is not " +
                        "present at the given path");
          }
         int k;
         try{
            while(( k = fis.read() ) != -1)
            {
                    System.out.print((char)k);
            }
            fis.close();
            }catch(IOException ioe){
               System.out.println("I/O error occurred: "+ioe);
            }
    }
}
```

This code will run fine and will display the file content.

Here are the few other Checked Exceptions –

- SQLException
- IOException
- ClassNotFoundException
- InvocationTargetException

2) Unchecked Exception

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it’s the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Note : The main difference between checked and unchecked exception is that the checked exceptions are checked at compile-time while unchecked exceptions are checked at runtime.

Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of RuntimeException class.

Example

```
class Example {
  public static void main(String args[])
  {
          int num1=10;
          int num2=0;
          /*Since I'm dividing an integer with 0
           * it should throw ArithmeticException
      */
          int res=num1/num2;
          System.out.println(res);
  }
}
```

If you compile this code, it would compile successfully however when you will run it, it would throw ArithmeticException. That clearly shows that unchecked exceptions are not checked at compile-time, they occurs at runtime. Lets see another example.

```
class Example {
  public static void main(String args[])
  {
          int arr[] ={1,2,3,4,5};
          /* My array has only 5 elements but we are trying to
      * display the value of 8th element. It should throw
           * ArrayIndexOutOfBoundsException
      */
          System.out.println(arr[7]);
  }
}
```

This code would also compile successfully since ArrayIndexOutOfBoundsException is also an unchecked exception.
Note: It doesn’t mean that compiler is not checking these exceptions so we shouldn’t handle them. In fact we should handle them more carefully. For e.g. In the above example there should be a

exception message to user that they are trying to display a value which doesn't exist in array so that user would be able to correct the issue.

```
class Example {
  public static void main(String args[]) {
         try{
           int arr[] ={1,2,3,4,5};
           System.out.println(arr[7]);
         }
     catch(ArrayIndexOutOfBoundsException e){
           System.out.println("The specified index does not exist " +
                  "in array. Please correct the error.");
         }
  }
}
```

Output:

```
The specified index does not exist in array. Please correct the error.
```

Here are the few unchecked exception classes:

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException
- IllegalArgumentException
- NumberFormatException

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description

try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

rest of the code...

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;

System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";

int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException
```

Java try-catch block

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

try{

//code that may throw an exception

}catch(Exception_class_Name ref){}

Syntax of try-finally block

try{

//code that may throw an exception

}finally{}

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

Example 1

```
public class TryCatchExample1 {

  public static void main(String[] args) {

    int data=50/0; //may throw exception
```

```
        System.out.println("rest of the code");

    }

}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

As displayed in the above example, the rest of the code is not executed (in such case, the rest of the code statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Example 2

```
public class TryCatchExample2 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            //handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

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

```
rest of the code
```

Now, as displayed in the above example, the rest of the code is executed, i.e., the rest of the code statement is printed.

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

```
public class TryCatchExample3 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
                    // if exception occurs, the remaining statement will not exceute
      System.out.println("rest of the code");
      }
          // handling the exception
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }

   }

}
```

Output:

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

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Example 4

Here, we handle the exception using the parent class exception.

```
public class TryCatchExample4 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
         // handling the exception by using Exception class
      catch(Exception e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

Output:

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

rest of the code
```

Example 5

```
public class TryCatchExample5 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
          // handling the exception
      catch(Exception e)
```

```
        {
                // displaying the custom message
            System.out.println("Can't divided by zero");
        }
    }

}
```

Output:

```
Can't divided by zero
```

Example 6

```
public class TryCatchExample6 {

    public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try
        {
        data=i/j; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
             // resolving the exception in catch block
            System.out.println(i/(j+2));
        }
    }
}
```

Output:

```
25
```

Example 7

In this example, along with try block, we also enclose exception code in a catch block.

```
public class TryCatchExample7 {

   public static void main(String[] args) {

      try
      {
      int data1=50/0; //may throw exception

      }
          // handling the exception
      catch(Exception e)
      {
         // generating the exception in catch block
      int data2=50/0; //may throw exception

      }
   System.out.println("rest of the code");
   }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

Example 8

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

```
public class TryCatchExample8 {

   public static void main(String[] args) {
      try
```

```
    {
    int data=50/0; //may throw exception

    }
      // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
    catch(ArrayIndexOutOfBoundsException e)
    {
      System.out.println(e);
    }
    System.out.println("rest of the code");
  }

}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Example 9

```
public class TryCatchExample9 {

  public static void main(String[] args) {
    try
    {
    int arr[]= {1,3,5,7};
    System.out.println(arr[10]); //may throw exception
    }
      // handling the array exception
    catch(ArrayIndexOutOfBoundsException e)
    {
      System.out.println(e);
    }
    System.out.println("rest of the code");
  }
}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10

rest of the code
```

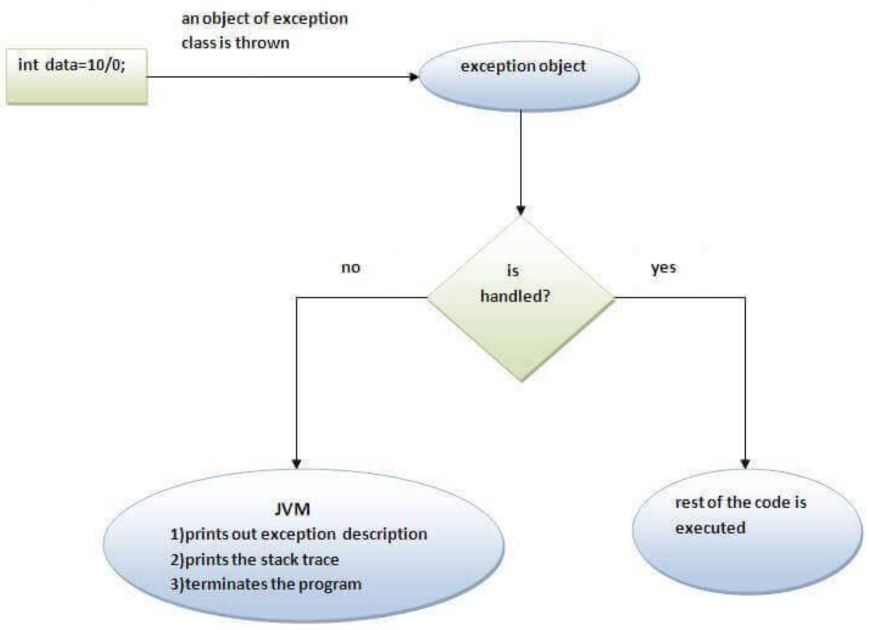
Example 10

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class TryCatchExample10 {

   public static void main(String[] args) {



      PrintWriter pw;
      try {
         pw = new PrintWriter("jtp.txt"); //may throw exception
         pw.println("saved");
      }
// providing the checked exception handler
 catch (FileNotFoundException e) {

         System.out.println(e);
      }
   System.out.println("File saved successfully");
   }
}
```

Output:

```
File saved successfully
```

Internal working of java try-catch block


int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
JVM
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Java catch multiple exceptions

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example 1

```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {

         try{
             int a[]=new int[5];
             a[5]=30/0;
            }
            catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
            catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
            System.out.println("rest of the code");
   }
}
```

Output:

```
Arithmetic Exception occurs

rest of the code
```

Example 2

```
public class MultipleCatchBlock2 {

   public static void main(String[] args) {
        try{
            int a[]=new int[5];

            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
   }
}
```

Output:

```
ArrayIndexOutOfBounds Exception occurs

rest of the code
```

Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.

```
public class MultipleCatchBlock3 {

   public static void main(String[] args) {

        try{
             int a[]=new int[5];
             a[5]=30/0;
             System.out.println(a[10]);
            }
            catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
            catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
            System.out.println("rest of the code");
   }
}
```

Output:

```
Arithmetic Exception occurs

rest of the code
```

Example 4

In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class Exception will invoked.

```
public class MultipleCatchBlock4 {

   public static void main(String[] args) {

         try{
             String s=null;
             System.out.println(s.length());
            }
            catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
            catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
            System.out.println("rest of the code");
   }
}
```

Output:

```
Parent Exception occurs

rest of the code
```

Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
class MultipleCatchBlock5{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(Exception e){System.out.println("common task completed");}
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   System.out.println("rest of the code...");
 }
}
```

Output:

```
Compile-time error
```

Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....
```

Java nested try example

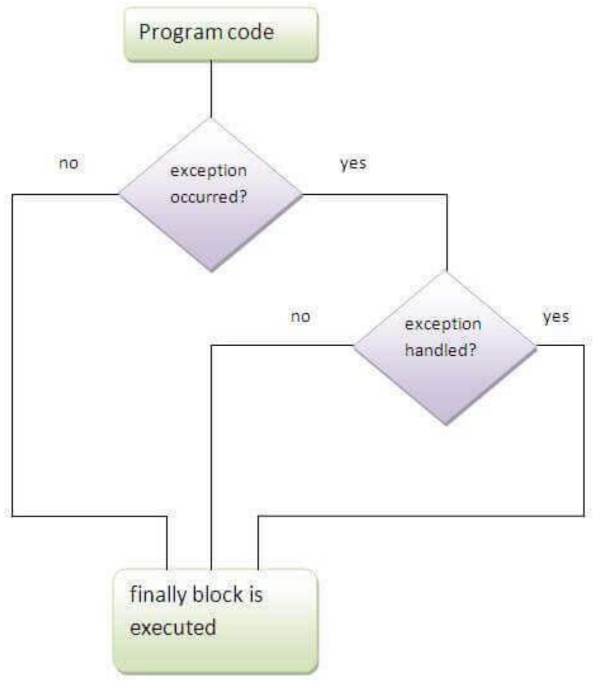
```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
    int a[]=new int[5];
    a[5]=4;
    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

    System.out.println("other statement);
  }catch(Exception e){System.out.println("handeled");}

  System.out.println("normal flow..");
 }
}
```

Java finally block

Java finally block is a block that is used to execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where exception doesn't occur.

```
class TestFinallyBlock{
 public static void main(String args[]){
 try{
  int data=25/5;
  System.out.println(data);
 }
 catch(NullPointerException e){System.out.println(e);}
 finally{System.out.println("finally block is always executed");}
```

```
  System.out.println("rest of the code...");
  }
}
```

Output:5

```
     finally block is always executed

     rest of the code...
```

Case 2

Let's see the java finally example where exception occurs and not handled.

```
class TestFinallyBlock1{

  public static void main(String args[]){

  try{

   int data=25/0;

   System.out.println(data);

  }

  catch(NullPointerException e){System.out.println(e);}

  finally{System.out.println("finally block is always executed");}

  System.out.println("rest of the code...");

  }

}
```

Output:

```
finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero
```

Case 3

Let's see the java finally example where exception occurs and handled.

```
public class TestFinallyBlock2{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(ArithmeticException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
       finally block is always executed
       rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw exception

Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

throw exception;

Let's see the example of throw IOException.

throw new IOException("sorry device error);

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
     validate(13);
     System.out.println("rest of the code...");
 }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

Java Exception propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation.

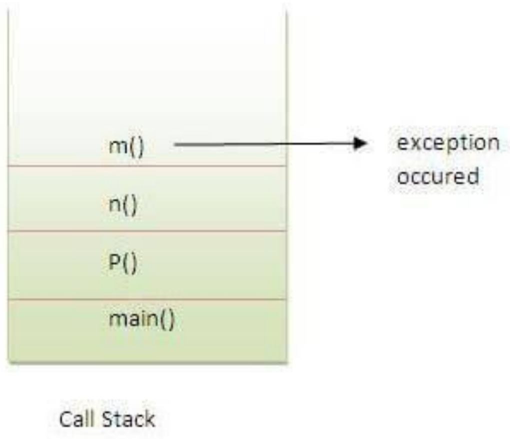
Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Program of Exception Propagation

```
class TestExceptionPropagation1{
 void m(){
   int data=50/0;
 }
 void n(){
   m();
 }
 void p(){
  try{
   n();
  }catch(Exception e){System.out.println("exception handled");}
 }
 public static void main(String args[]){
  TestExceptionPropagation1 obj=new TestExceptionPropagation1();
  obj.p();
  System.out.println("normal flow...");
 }
}
```

Output:

```
exception handled

normal flow...
```



In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Program which describes that checked exceptions are not propagated

```
class TestExceptionPropagation2{
 void m(){
  throw new java.io.IOException("device error");//checked exception
 }
 void n(){
  m();
 }
 void p(){
  try{
   n();
  }catch(Exception e){System.out.println("exception handeled");}
 }
 public static void main(String args[]){
  TestExceptionPropagation2 obj=new TestExceptionPropagation2();
  obj.p();
  System.out.println("normal flow");
 }
}
```

Output:Compile Time Error

Java throws keyword

The Java throws keyword is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

```
return_type method_name() throws exception_class_name{

//method code

}
```

Which exception should be declared

Ans) checked exception only, because:

- **unchecked** Exception: under your control so correct your code.
- **error**: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
 void m()throws IOException{
  throw new IOException("device error");//checked exception
 }
 void n()throws IOException{
  m();
 }
 void p(){
  try{
   n();
  }catch(Exception e){System.out.println("exception handled");}
 }
 public static void main(String args[]){
  Testthrows1 obj=new Testthrows1();
  obj.p();
  System.out.println("normal flow...");
 }
}
```

Output:

```
exception handled

normal flow...
```

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

- **Case1**:You caught the exception i.e. handle the exception using try/catch.
- **Case2**:You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled

normal flow...
```

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

Output:

```
device operation performed

normal flow...
```

B)Program if exception occurs

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

Output:

```
Runtime Exception
```

Difference between throw and throws

Que) Can we rethrow an exception?

Yes, by throwing same exception in catch block.

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.

3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java throw example

```
void m(){

throw new ArithmeticException("sorry");

}
```

Java throws example

```
void m()throws ArithmeticException{

//method code

}
```

Java throw and throws example

```
void m()throws ArithmeticException{

throw new ArithmeticException("sorry");

}
```

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize

1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
class FinalExample{

public static void main(String[] args){

final int x=100;

x=200;//Compile Time Error

}}
```

Java finally example

```
class FinallyExample{

public static void main(String[] args){

try{

int x=300;

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

finally{System.out.println("finally block is executed");}

}}
```

Java finalize example

```
class FinalizeExample{

public void finalize(){System.out.println("finalize called");}
public static void main(String[] args){

FinalizeExample f1=new FinalizeExample();

FinalizeExample f2=new FinalizeExample();

f1=null;

f2=null;

System.gc();

}}
```

ExceptionHandling with MethodOverriding in Java

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

- If the superclass method does not declare an exception
- If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- If the superclass method declares an exception
- If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the superclass method does not declare an exception

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
import java.io.*;
class Parent{
  void msg(){System.out.println("parent");}
}
  class TestExceptionChild extends Parent{
  void msg()throws IOException{
    System.out.println("TestExceptionChild");
  }
  public static void main(String args[]){
   Parent p=new TestExceptionChild();
   p.msg();
  }
}
```

Output:

```
Compile Time Error
```

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
import java.io.*;
class Parent{
  void msg(){System.out.println("parent");}
}
class TestExceptionChild1 extends Parent{
  void msg()throws ArithmeticException{
    System.out.println("child");
  }
  public static void main(String args[]){
   Parent p=new TestExceptionChild1();
   p.msg();
  }
}
```

Output:

```
child
```

If the superclass method declares an exception

1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Example in case subclass overridden method declares parent exception

```
import java.io.*;
class Parent{
  void msg()throws ArithmeticException{System.out.println("parent");}
}
```

```
class TestExceptionChild2 extends Parent{
  void msg()throws Exception{System.out.println("child");}
    public static void main(String args[]){
   Parent p=new TestExceptionChild2();
   try{
   p.msg();
   }catch(Exception e){}
  }
}
```

Output:

```
Compile Time Error
```

Example in case subclass overridden method declares same exception

```
import java.io.*;
class Parent{
  void msg()throws Exception{System.out.println("parent");}
}
class TestExceptionChild3 extends Parent{
  void msg()throws Exception{System.out.println("child");}
  public static void main(String args[]){
   Parent p=new TestExceptionChild3();
   try{
   p.msg();
   }catch(Exception e){}
  }
}
```

Output:

```
child
```

Example in case subclass overridden method declares subclass exception

```
import java.io.*;
class Parent{
  void msg()throws Exception{System.out.println("parent");}
}
```

```
class TestExceptionChild4 extends Parent{
  void msg()throws ArithmeticException{System.out.println("child");}
  public static void main(String args[]){
   Parent p=new TestExceptionChild4();
   try{
   p.msg();
   }catch(Exception e){}
  }
}
```

Output:

```
child
```

Example in case subclass overridden method declares no exception

```
import java.io.*;
class Parent{
  void msg()throws Exception{System.out.println("parent");}
}
class TestExceptionChild5 extends Parent{
  void msg(){System.out.println("child");}
  public static void main(String args[]){
   Parent p=new TestExceptionChild5();
   try{
   p.msg();
   }catch(Exception e){}
  }
}
```

Output:

```
child
```

Java Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
class TestCustomException1{
   static void validate(int age)throws InvalidAgeException{
     if(age<18)
      throw new InvalidAgeException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]){
      try{
      validate(13);
      }catch(Exception m){System.out.println("Exception occured: "+m);}
      System.out.println("rest of the code...");
  }
}
```

Output:

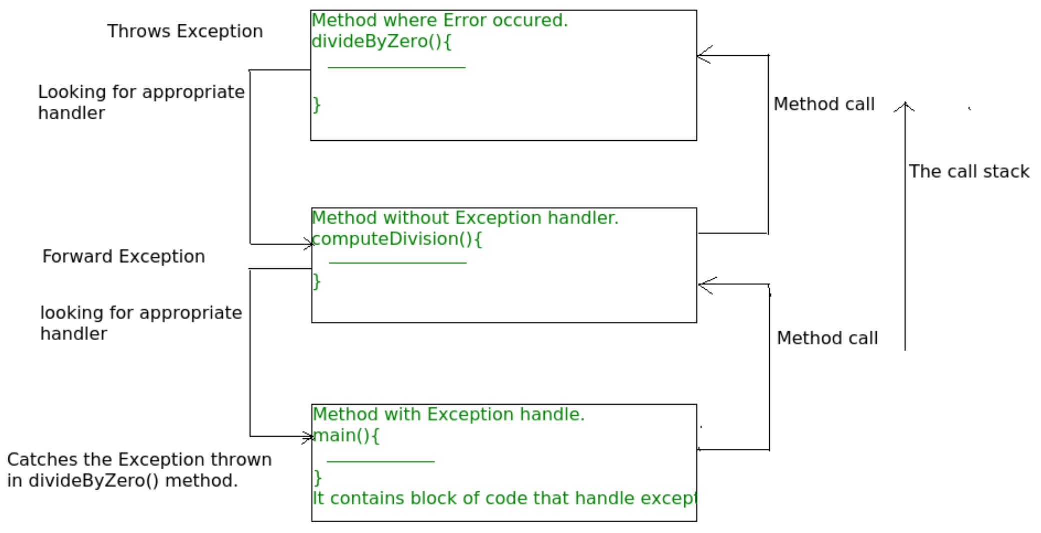
```
Exception occured: InvalidAgeException:not valid

       rest of the code...
```

How JVM handle an Exception?

Default Exception Handling : Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM). The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception. There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called Call Stack. Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called Exception handler.
- The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.
- If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to default exception handler , which is part of run-time system. This handler prints the exception information in the following format and terminates program abnormally.
- Exception in thread "xxx" Name of Exception : Description
- ... . // Call Stack

See the below diagram to understand the flow of the call stack.


Throws Exception
Looking for appropriate handler
Forward Exception
looking for appropriate handler
Catches the Exception thrown in divideByZero() method.
Method where Error occured.
divideByZero(){
}
Method without Exception handler.
computeDivision(){
}
Method with Exception handle.
main(){
}
It contains block of code that handle excep
Method call
Method call
The call stack

The call stack and searching the call stack for exception handler.

Example :

// Java program to demonstrate how exception is thrown.

class ThrowsExecp{

```
    public static void main(String args[]){


        String str = null;
        System.out.println(str.length());


    }
}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException

    at ThrowsExecp.main(File.java:8)
```

Let us see an example that illustrate how run-time system searches appropriate exception handling code on the call stack :

```
// Java program to demonstrate exception is thrown
// how the runTime system searches th call stack
// to find appropriate exception handler.
class ExceptionThrown
{
    // It throws the Exception(ArithmeticException).
    // Appropriate Exception handler is not found within this method.
    static int divideByZero(int a, int b){


        // this statement will cause ArithmeticException(/ by zero)
        int i = a/b;


        return i;
    }


    // The runTime System searches the appropriate Exception handler
    // in this method also but couldn't have found. So looking forward
    // on the call stack.
    static int computeDivision(int a, int b) {
```

```
   int res =0;

   try
   {
     res = divideByZero(a,b);
   }
   // doesn't matches with ArithmeticException
   catch(NumberFormatException ex)
   {
      System.out.println("NumberFormatException is occured");
   }
   return res;
}

// In this method found appropriate Exception handler.
// i.e. matching catch block.
public static void main(String args[]){

   int a = 1;
   int b = 0;

   try
   {
       int i = computeDivision(a,b);

   }

   // matching ArithmeticException
   catch(ArithmeticException ex)
   {
       // getMessage will print description of exception(here / by zero)
       System.out.println(ex.getMessage());
   }
```

```
    }
}
```

Output :

```
/ by zero.
```

How Programmer handles an exception?

Customized Exception Handling : Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

Consider the following java program.

```
// java program to demonstrate
// need of try-catch clause

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

        // array of size 4.
        int[] arr = new int[4];

        // this statement causes an exception
        int i = arr[4];

        // the following statement will never execute
        System.out.println("Hi, I want to execute");
    }
}
```

Output :

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at CDAC.main(CDAC.java:9)
```

Explanation : In the above example an array is defined with size i.e. you can access elements only from index 0 to 3. But you trying to access the elements at index 4(by mistake) that’s why it is throwing an exception.In this case, JVM terminates the program abnormally. The statement System.out.println(“Hi, I want to execute”); will never execute. To execute it, we must handled the exception using try-catch. Hence to continue normal flow of the program, we need try-catch clause.

How to use try-catch clause

```
try {
// block of code to monitor for errors
// the code you think can raise an exception
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// optional
finally {
// block of code to be executed after try block ends
}
```

Points to remember :

- In a method, there can be more than one statements that might throw exception, So put all these statements within its own try block and provide separate exception handler within own catch block for each of them.
- If an exception occurs within the try block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put catch block after it. There can be more than one exception handlers. Each catch block is a exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type

of the exception that it can handle and must be the name of the class that inherits from Throwable class.

- For each try block there can be zero or more catch blocks, but only one finally block.
- The finally block is optional.It always gets executed whether an exception occurred in try block or not . If exception occurs, then it will be executed after try and catch blocks. And if exception does not occur then it will be executed after the try block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.