Exception handling

Exception handling is one of the most important feature of java programming that allows us to handle the runtime errors caused by exceptions.

What is an exception?

Dictionary meaning of the exception is abnormal termination. An exception is a problem occurred during execution time of the program.

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.

Exception is a class present in java.lang package.

All the exceptions are nothing but objects of called classes.

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.

In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome.

Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

What is the meaning of exception handling?

Exception handling doesn't mean repairing an exception.We have to define alternative way to continue rest of the program normally this way of "defining alternative way to continue rest of the program"

It is highly recommended to handle exceptions.The main objective of exception handling is grace ful(normal) termination of the program.

When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment

Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.

Examples :

Uncaught Exception : We not using **try….catch** block in our program.

```
class ExceptionExample1
 {
 public static void main(String[] args)
 {
   System.out.println("Prestige ");
   System.out.println("Point");
   System.out.println(10/0);
   System.out.println("Institutes");
  }
}
```

```
Output :
Prestige
Point
Exception in Thread "main" :
java.lang.ArithmeticException: / by zero
```

Note:- if we are not using try-catch it is always abnormal termination if an Exception raised.

Default exception handling in java:
•If an exception raised inside any method then the method is responsible to create Exception object with the following information.
•Nameoftheexception.
•Descriptionoftheexception.
•Locationoftheexception.
• After creating that Exception object the method hand overs that object to the JVM.
• JVM checks whether the method contains any exception handling code or not. If method won't contain any handling code then JVM terminates that method abnormally and removes corresponding entry form thestack.
• JVM identifies the caller method and checks whether the caller method contain any handling code or not. If the caller method also does not contain handling code then JVM terminates that caller also abnormally and removes corresponding entry from the stack.
• This process will be continued until main() method and if the main() method also doesn't contain any exception handling code then JVM terminates main() method and removes corresponding entry from the stack.
• Then JVM hand overs the responsibility of exception handling to the default exception handler.
• Default exception handler justprint exception information to the console in the following formats and terminates the program abnormally.

```
Nameofexception:description
Locationofexception(stacktrace)
```

```
class ExceptionExample2
 {
 static void m()
   {
    System.out.println("Prestige ");
    System.out.println("Point");
    System.out.println(10/0);
    System.out.println("Institutes");
   }
 static void m1() { m(); }
 public static void main(String[] args)
 {
    m1();
    }
}
```

```
Output :
Prestige
Point
Exception in Thread "main" : java.lang.ArithmeticException: / by zero
          at  ExceptionExample2.m(ExceptionExample2.java:7)
          at  ExceptionExample2.m1(ExceptionExample2.java:10)
          at  ExceptionExample2.main(ExceptionExample2.java:13)
```

Runtime stack mechanism: Fore very thread JVM will create a separate stack all method calls performed by the thread will be stored in that stack. Each entry in the stack is called "oneactivationrecord" (or) "stackframe". After completing every method call JVM removes the corresponding entry from the stack. After completing all method calls JVM destroys the empty stack and terminates the program normally.

```
class Account
{
        void deposit(int accno)
                { System.out.println("money is deposited into  "+accno);  }
        void withdraw(float amount)
                {   System.out.println("money  withdraw is over amount "+amount);  }
        public static void main(String[] args)
        {
        Account  t=new Account();
         t.deposit(111);
        t.withdraw(10000);
         }
}
```

Run Time Stack :

Step 1:- One empty stack is created

Step 2:- Whenever the JVM calls the main method the main method entry is stored in the stack memory. The stored entry is deleted whenever the main method is completed.

Step 3:- Whenever the JVM is calling the deposit () the method entry is stored in the stack and local variables are store in the stack. The local variables are deleted whenever the JVM completes the deposit () method execution. Hence the local variables scope is only within the method.

Step 4 :- The deposit method is completed Then deposit () method is deleted from the stack.

Step 5 :- Only main method call present in the stack.

Step 6:- Whenever the JVM calls the withdraw() method the entry is stored in the stack.

Step 7:- Whenever the withdraw() method is completed the entry is deleted from the stack.

Step 8:- Whenever the main method is completed the main method is deleted from the stack.

Customized exception handling by try catch :

•It is highly recommended to handle exceptions.

•In our program the code which may cause an exception is called risky code we have to place risky code inside try block and the corresponding handling code inside catch block

```
try {
 // block of code to monitor for errors
    }
    catch (ExceptionType1 exOb)
   {
  // exception handler for ExceptionType1
   }
   catch (ExceptionType2 exOb)
 {
  // exception handler for ExceptionType2
  }
    // ...
   finally
    {
        // block of code to be executed after try block ends
   }
```

Example :

caught Exception : We using **try….catch** block in our program.

```
class ExceptionExample3
 {
 public static void main(String[] args)
 {
   System.out.println("Prestige ");
   System.out.println("Point");
try {
   System.out.println(10/0);
       }
 catch (ArithmeticException e)
         {   System.out.println("you are getting AE "+e);   }
  System.out.println("Institute");
  }
}
```

```
Output :
Prestige
Point
You are getting java.lang.ArithmeticException: /
by zero
Institute
```

Note:

•Within the try block if any where an exception raised then rest of the try block won't be executed even though we handled that exception. Hence we have to place/take only risk code inside try and length of the try block should be as less as possible.

• If any statement which raises an exception and it is not part of any try block then it is always abnormal termination of the program.

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

Case 1 : Exception raised in try block the JVM will search for corresponding catch block if the catch block is matched, corresponding catch block will be executed and rest of the code is executed normally.

```
class ExceptionExample4
      {
            public static void main(String[] args)
            {
             System.out.println("program starts");
             try   {    int[] a={10,20,30};
                        System.out.println(a[0]);
                        System.out.println(a[1]);
                        System.out.println(a[2]);
                         System.out.println(a[3]);
                   }
            catch(ArrayIndexOutOfBoundsException ae)
             {    System.out.println("we are getting exception");   }

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

Case 2 : Exception raised in try block the JVM will search for corresponding catch block if the catch block is matched, corresponding catch block will be executed and rest of the code is executed normally. If the catch block is not matched the program is terminated abnormally.

```
 class ExceptionExample5
      {
            public static void main(String[] args)
            {
             System.out.println("program starts");
             try   {    int[] a={10,20,30};
                        System.out.println(a[0]);
                        System.out.println(a[1]);
                        System.out.println(a[2]);
                         System.out.println(a[3]);
                   }
             catch(AithmeticException ae)
                         {    System.out.println("we are getting exception");    }
            System.out.println("rest of the code");
```

```
} }
```

Case 3 : if there is no exception in try block the catch blocks won't be executed.

```
          class ExceptionExample5
           {
           public static void main(String[] args)
            {  System.out.println("program starts");
                     try  {   System.out.println("Hi");
                              System.out.println("how r u");  }
                   catch(ArithmeticException ae)
                   {   System.out.println("we are getting exception");  }
               System.out.println("rest of the code");
            }
          }
```

Case 4 : independent try blocks are not allowed in java language.(compilation error)

```
          class ExceptionExample6
          {
           public static void main(String[] args)
           {
            System.out.println("program starts");
             try  {   int a=10/0;  }
          System.out.println("rest of the code is avilable");
           }
          }
```

Case 5 : In between try and catch independent statements are not allowed. If we are providing independent statements the compiler will raise compilation error.

```
class ExceptionExample6
{
public static void main(String[] args)
   {
     System.out.println("program starts");
       try  {   int a=10/0;  }
    System.out.println("in between try and catch");
   catch(ArithmeticException e)
   {   int a=10/5;
       System.out.println(a);  }
   System.out.println("rest of the code is avilable");
  }
}
```

Case 6: Inside the try block once we are getting exception the JVM search for the corresponding catch block . If the catch block is matched then it will executed that catch block the program is terminated normally the control never goes try block once again. Once the control is out of the try block the control never entered into try block once again .

```
class ExceptionExample7
  {  public static void main(String[] args)
          {
         System.out.println("program starts");
          try   {
                        System.out.println("PrestigePoint");
                        int a=10/0;
                       System.out.println("Hello");
                       System.out.println("how are you boys");
                 }
         catch(ArithmeticException e)
                    {   int a=10/5;
                         System.out.println(a);   }
                       System.out.println("rest of the code ");
          }
 }
```

Case 7 : The way of handling the exception is varied from exception to the exception so it is recommended to provide try with number of catch blocks.

```
import java.util.*;
class ExceptionExample8 {
        public static void main(String[] args)
         {
          Scanner s=new Scanner(System.in);
         System.out.println("provide the division value");
         int n=s.nextInt();
         try   {    System.out.println(10/n);
                   String str=null;
                   System.out.println("u r name is :"+str);
                   System.out.println("u r name length is--->"+str.length());
               }
         catch (ArithmeticException ae)
               {
           System.out.println("zero not allowed geting Exception"+ae);
           }
         catch (NullPointerException ne)
           {   System.out.println(" Exception"+ne);
           }
       System.out.println("rest of the code");
     }
}
```

Exception Hierarchy

$$\begin{tikzcd}
 & \text{Throwable} \arrow[dl] \arrow[dr] & \\
\text{Exception} \arrow[d] & & \text{Error} \\
\text{RuntimeException} & &
\end{tikzcd}$$

Throwable is at the top of the exception class hierarchy.
Exception class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.
RuntimeException are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Case 8 : By using root class (Exception) we are able to hold any type of exceptions.

```
import  java.util.*;
class ExceptionExample8  {
            public static void main(String[] args)
            {   Scanner s=new Scanner(System.in);
               System.out.println("provide the division value");
               int n=s.nextInt();
             try   {    System.out.println(10/n);
                          String str=null;
                        System.out.println("u r name is :"+str);
                        System.out.println("u r name length is--->"+str.length());
                    }
             catch (Exception e)   {    System.out.println("getting  Exception"+e);   }

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

Case 9 : In java class if we are declaring multiple catch blocks at that situation the catch block order should be child to parent shouldn't be parent to the child.

```
import java.util.*;
 class ExceptionExample9  {
        public static void main(String[] args)
        {
         Scanner s=new Scanner(System.in);
          System.out.println("provide the division val");
          int n=s.nextInt();
          try   {  System.out.println(10/n);
                   String str=null;
                   System.out.println("u r name is :"+str);
                   System.out.println("u r name length is-->"+str.length());
                }
         catch (ArithmeticException ae)
                   {  System.out.println("Exception"+ae);   }

         catch (Exception ne)   {    System.out.println("Exception"+ne);   }

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

If we write above code(red marked in above example) as

```
catch (Exception ne)   {    System.out.println("Exception"+ne);   }
catch (ArithmeticException ae)
   {  System.out.println("Exception"+ae);   }
```

Then we will get compilation error

Case 10: The exception raised in catch block it is always abnormal termination.

```
class ExceptionExample10
   {  public static void main(String[] args)
      {
        System.out.println("program starts");
            try   {    int a=10/0;   }
          catch(ArithmeticException e)
            {    int a=10/0;
                 System.out.println(a);
            }
  System.out.println("rest of the code is avilable");
   }
}
```

Case 11 : Try with multiple catch blocks: The way of handling an exception is varied from exception to exception hence for every exception raise a separate catch block is required

```
class  MultipleTryCatchBlock
{  static String str;//instance variable default value is null
```

```
   public static void main(String[] args)
    {
       try
         {     System.out.println(10/0);   }
      catch (ArithmeticException ae)
         {   System.out.println("we are getting ArithematicException"+ae);   }

      try      {   System.out.println(str.length());   }
      catch (NullPointerException ne)
              {   System.out.println("we are getting nullpointerException"+ne);   }

     System.out.println("Multiple times we provide try-catch");
    System.out.println("the rest of the code is executed");
 }
}
```

Case 12 : Nested try-catch

```
          class NestedTryCatch  {
             public static void main(String[] args)
              {
               try  {
                    try   {   System.out.println("first try block");
                               int a=10/0;
                              System.out.println(a);
                          }
                    catch (ArithmeticException ae)
                         {   System.out.println("first catch block"+ae);   }
                    try   {   System.out.println("second try block");
                              int[] a={10,20,30};
                             System.out.println(a[5]);
                          }
                    catch (ArrayIndexOutOfBoundsException aaa)
                        {   System.out.println("second catch block"+aaa);   }
                   }
             catch (Exception e)  {   System.out.println("main  catch"+e);  }
      System.out.println("normal flow of execution");
       } }
```

Case 13 : catch with try-catch

```
          class CatchWithTryCatch {

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

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

Finally Block

• It is never recommended to take clean up code inside try block because there is no guarantee for the execution of every statement inside a try.

• It is never recommended to place clean up code inside catch block because if there is no exception then catch block won't be executed.

• We require some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled such type of place is nothing but finally block.

• Hence the main objective of finally block is to maintain clean up code.

try { riskycode }
catch (x e) { handlingcode }
finally { cleanupcode }

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

Case 1: If No Exception Occured in try block, catch block is not exceuted but finlly block is exectued. There is a normal termination of program.

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

```
Output:

try block executed
finally block executed
```

Case 2: If a Exception Occured in try block, matched corresponding catch block will be exceuted and also finly block is exectued. There is a normal termination of program.

```
class FinallyBlock1
   {
   public static void main (String[] args)
           {
                     try  {
                               System.out.println("try block executed ");
                               System.out.println(10/0);
                               System.out.println("End of Try Block ");
                               }
                     catch(ArithmeticException e)
                         {
                        System.out.println("catch block executed");
                        }
                      finally
                        {  System.out.println("finally block executed");
                        }
                     System.out.println("Rest of the Code");
           }
   }
```

```
Output:

try block executed
catch block executed
finally block executed
Rest of the Code
```

Case 3: If a Exception Occured in try block and corresponding matched catch block is not avaible, there is abnormal termination of program but finly block is exectued.

```
class FinallyBlock2
   {
   public static void main (String[] args)
           {
                     try  {
                               System.out.println("try block executed ");
                               System.out.println(10/0);
                               System.out.println("End of Try Block ");
                               }
                     catch(NullPointerException e)
                         {
                        System.out.println("catch block executed");
                        }
                      finally
                        {  System.out.println("finally block executed");
                        }
                     System.out.println("Rest of the Code");

           }
   }
```

```
Output:
try block executed
finally block executed
Exception in Thread "main"  :java.lang.ArithmeticException:   / by zero
```

Return Vs Finally:
Even though return present in try or catch blocks first finally will be executed and after that only return statement will be considered that is finally block dominates return statement.

Example:

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

```
Output:
try block executed
finally block executed
```

•If return statement present in try catch and finally blocks then finally block return statement will be considered.

Example:

```
class FinallyReturn
 {
   public static void main(String[]args)
            {
            System.out.println(methodOne());
            }
 public static int methodOne()
            {
             try {
                        System.out.println(10/0);
                         return777;
                        }
            catch(ArithmeticException  e)
              { return   888; }
            finally {    return  999;   }
            }//methodOne
}//class
```