Exception Handling in Java

The exception handling is one of the powerful mechanisms provided in java. It provides the mechanism to handle the **runtime errors** so that normal flow of the application can be maintained.

Exception: Dictionary Meaning: Exception is an abnormal condition. In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling: Exception Handling is a mechanism to handle runtime errors.

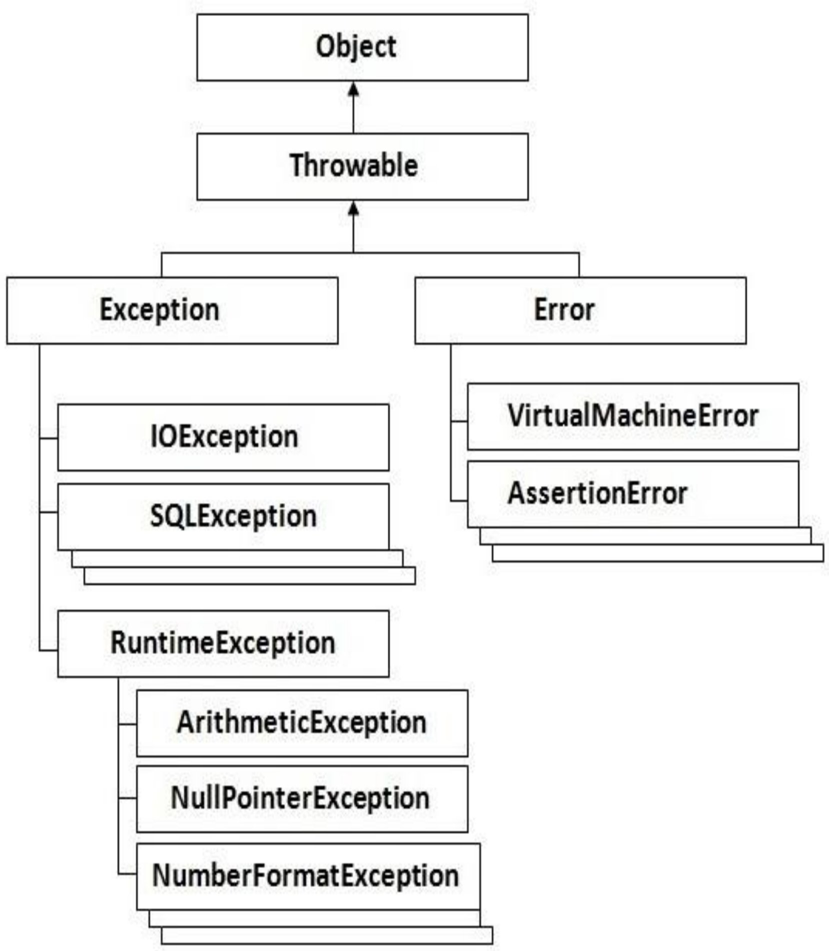

Fig: Hierarchy of Exception Class

Types of Exception:

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

- Checked Exception (Compile Time)
- Unchecked Exception (RUN Time)
- Error (can't be handled by Programmer

What is the difference between checked and unchecked exceptions ?

Checked Exception: The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at **compile-time.**

Unchecked Exception: The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are **checked at runtime.**

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

Common scenarios of Exception Handling where exceptions may occur: There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

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

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

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

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

3) 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) Scenario where ArrayIndexOutOfBoundsException occurs: If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50;              //ArrayIndexOutOfBoundsException
```

- Java defines several other types of exceptions that relate to its various class libraries.

  Following is the list of Java Unchecked RuntimeException.

Exception	**Description**
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

- **Following is the list of Java Checked Exceptions Defined in java.lang.**

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.

Five keywords used in Exception handling:

1) try /// throw the exception
2) catch // Handle the Exception
3) finally, // This execute every time either Exception is thrown or not
4) throw // to throw the exception explicit
5) throws /// not used for handling exception

Try-catch – We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.

finally – finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurred or not.

try block: Enclose the code that might throw an exception in try block. It must be used within the method and must be followed by either catch **or** finally block.

throw – We know that if any exception occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometime we might want to generate exception explicitly in our code, for example in a user authentication program we should throw exception to client if the password is null. **throw** keyword is used to throw exception to the runtime to handle it.

throws – When we are throwing any exception in a method and not handling it, then we need to use **throws** keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to it's caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with main() method also.

The 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 programmer has not checked logic of the source code before the code is being used.

1. Default throw and default catch
2. Default throw and explicit catch
3. Explicit throw and default catch
4. Explicit throw and Explicit catch

Syntax of try with catch block

```
try  { int x,y;
int c=x/y;   }
catch(Exception_class_Name reference)     {  ………….. }
```

Syntax of try with finally block

```
try
{    int c=x/y;
       …….
Sop(c);
}
Catch(CN1   Ref1)  {,,,,,,,,,,,,,,,,,,,,,,,,}
```

Catch(CN2 Ref2) {,,,,,,,,,,,,,,,,,,,,,,,}

Catch(CN 3 Ref3) {,,,,,,,,,,,,,,,,,,,,,,,}

finally

{ SOP("final block is executed"); } /// execute always

catch block: Catch block is used to handle the Exception. It must be used after the try block.

1. Default throw and default catch

```
class Exception1  {
      public static void main(String args[])   {
     int data=50/0;
     System.out.println("rest of the code...");
            }
  }
```

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

What happens behind the code int a=50/0;

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.

Solution by Exception Handling

2.Default throw and Explicit catch

```
class Exception2
 {
 public static void main (String args[])
  {
     try
       {
         int data=50/0;
        }catch(ArithmeticException e)
         {
          System.out.println(e);
         }
    System.out.println("rest of the code...");
  }
}
```

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

    rest of the code..

Example : **Default throw and Explicit catch is not Matched.**

```
class Exception3
 {
 public static void main (String args[])
  {
     try
       {
         int data=50/0;
        }catch(NullPointerException e)  //  not matched so D catch is running
         {
          System.out.println(e);
```

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

3.Explicit throw and Default catch: The throw keyword is used to **explicitly** throw an exception. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exception

Example

```
import java.util.Scanner;
class ExplicitThrow {                              //ExplicitThrow but default catch
 public static void main(String args[])
{   int balance=5000;
     int withdamt;
    Scanner sc =new Scanner(System.in);
    System.out.println("Enter Withdrawal amount");
    withdamt=sc.nextInt();
    if(balance<withdamt)
    throw new ArithmeticException("Insufficient Balance");
    balance=balance-withdamt;
   System.out.println(" Txn successful");
   System.out.println("Withdrawal amt is"+withdamt+" and Blance amt is "+balance);
     }
}
```

Example2: Explicit throw and default catch

```
          import java.util.Scanner;
          class Exception_ED
          {
            static void validate(int age)
                {
                  if(age<5)
                throw new ArithmeticException("You are not eligible for addmission");
              else
                System.out.println("Welcome !  You are eligible for addmission");
            }
             public static void main(String args[])
                {         Scanner  sc= new Scanner(System.in);
                          System.out.println("enter age :");
                          int age= sc.nextInt();
                         validate(age);
                         System.out.println("Thank you");
                }
        }
```

4.Explicit Throw Explicit catch

```
import java.util.Scanner;
class ExplicitThrowExplicitCatch
 public static void main(String args[])
{   int balance=5000;
     int withdamt;
    Scanner sc =new Scanner(System.in);
    System.out.println("Enter Withdrawl amount");
    withdamt=sc.nextInt();
  try{    if(balance<withdamt)
             throw new ArithmeticException("Insufficient Balance");
             balance=balance-withdamt;
             System.out.println(" Txn successful");
            System.out.println("W_amt is:"+withdamt+" Balance amt is "+balance);
      }
catch(ArithmeticException ae)
        {  System.out.println(" Exception::"+ae.getMessage());   }       }  }
```

Multiple catch blocks:

If you have to perform different tasks at the occurrence of different Exceptions, use multiple catch block.

Example of multiple catch block:

```
class Exception4
{
  public static void main (String args[])
{
   try
{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(ArithmeticException e)
                {     System.out.println("task1 is completed");                }
   catch(ArrayIndexOutOfBoundsException e)
                 {     System.out.println("task 2 completed");     }
   catch(Exception e)
                  {System.out.println("common task completed");
 }
   System.out.println("rest of the code...");
 }
}
```

```
Output:task1 completed
             rest of the code...
```

Rule:At a time only one Exception is occurred and at a time only one catch block is executed.

Rule:All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .**For Example ..**

```
        class Exception5
        {
          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   **// Arithmetic Exception must come before Exception**

Nested try block: try block within a try block is known as nested try block.

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)
    {
    }
    ....
```

Example of nested try block:

```
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..");
 }
}
```

finally block: The finally block is a block that is always executed. It is mainly used to perform some important tasks such as closing connection, stream etc.

Note: Before terminating the program, JVM executes finally block (if any).

Note: finally, must be followed by try or catch block.

Why use finally block?

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

case 1

Program in case exception does not occur

```
class Simple
{
  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

Program in case exception occurred but not handled

```
class Simple{
 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

Program in case exception occurred and handled

```
class Simple
{
 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).

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)

If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword. Try to understand the different in throws and throw keyword.

Syntax of throws keyword:

```
RT method_name() throws exception_class_name, Ex2,Ex3
                                        {    ... }
```

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

There are two cases:

Case1: To caught the exception i.e. handle the exception using try/catch.

Case2: To declare the exception i.e. specifying throws with the method.

```
import java.io.*;
class ThrowsDemo
{    public static void main(String args[])
   {        throw new IOException();          }
}
```

Output

```
ThrowsDemo.java:6: error: unreported exception IOException; must be caught or declared
to be thrown
        throw new IOException();
        ^
1 error
```

```
import java.io.*;
class ThrowsDemo2
{
   public static void main(String args[]) throws IOException  //declare of  exception
    {
        throw new IOException();
      }
}
```

Output:

```
Exception in thread "main" java.io.IOException
        at ThrowsDemo2.main(ThrowsDemo2.java:6)
```

```
import java.io.*;
class ThrowsDemo3
{
   public static void main(String args[]) throws IOException  //declare exception
       {
           try{ throw new IOException("Input output error");   }

          catch(IOException e)
             { System.out.println("Excetion=="+e.getMessage());}
     }
}
```

Output:

```
D:\YMS\java_program>javac ThrowsDemo3.java
```

```
D:\YMS\java_program>java ThrowsDemo3
Excetion==Input output error
```

Difference between throw and throws:

throw is used to explicitly throw an exception.	throws is used to declare an exception.
checked exception cannot be propagated without throws.	checked exception can be propagated with throws.
throw is followed by an instance.	throws keyword is followed by class.
throw is used within the method.	throws keyword is used with the method signature.

You cannot throw multiple exception.

```
class Excep13
        {       static void validate(int age)
         {        if(age<18)
 throw new ArithmeticException ("notvalid");
             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

You can declare multiple exception e.g.

```
public void method()throws
IOException,SQLException
Example: class M{
 void method()throws
IOException{
  throw new IOException("device
error");
 } }
class Test{
   public static void main(String
args[])throws IOException{
//declare exception
    Test t=new Test();
    t.method();
    System.out.println("normal flow...");
  }}
```

Output:Runtime Exception

Note:

- For each try block, there is zero or more catch block but only one finally block.
- The catch block and finally block must be in conjunction with try block.
- Try block must be followed by either at least one Catch block or one finally block.
- Catch block is the exception handler and catch block must be from specific to general.

```
Try  {}
```

```
Catch(){}
Finally  {}
```

- **Try Block Without Catch Block**

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

```
                       O/P:  Try Block Executed
                             Finally Block Executed
```

- **Try Block with Catch Block**

```
public class TryWithFinallyDemo
 {
   public static int  cubeMethod( int x) {
      try {
         System.out.println("Try Block with return type");
          return x*x*x ;
      } finally {
         System.out.println("Finally Block always execute");
      }
```

```
  }
  public static void main(String[] args)
    {
      System.out.println("Cube value =="+cubeMethod(5));
  }
}
```

```
O/P:   Try Block with return type
       Finally Block always execute
       Cube value ==125
```