Exception Handling

Exception Handling

Q) What is the difference between *Error* and *Exception?*

Error is a problem in java applications. it will not allow java application execution
There are two types of errors in java.

- Compile time Errors
- Runtime Errors

• Compile Time Errors:

These are problems identified by the compiler at compilation time.
In general, there are 3 types of compile time errors.

- Lexical Errors: Mistakes in keywords,..
  EX: int i=10; ----> Valid
  nit i=10; ----> Invalid

- Syntax Errors: Gramatical Mistakes or syntactical mistakes.
  EX: int i=10;---> Valid
  i int 10 ; --> Invalid

- Semantic Errors: Providing operators in between in compatible types.

  EX: int i=10;
  int j=20;
  int k=i+j; ----> Valid

  EX: int i=10;
  boolean b=true;
  char c= i+b;----> Invalid.

NOTE: Some other errors are generated by Compiler when we violate JAVA rules and regulations in java applications.

EX: Unreachable Statements, valriable might not have been initialization, possible loss of precision,....

• Runtime Errors:

These are the problems for which we are unable to provide solutions programmatically and these errors are not identified by the compilers and these errors are occurred at runtime.

EX: Insufficient Main Memory.
Unavailability of IO Components.
JVMInternal Problems.

2) Exception:

Exception is a problem, for which we are able to provide solutions programmatically.

EX: ArithmeticException
NullPointerException
ArrayIndexOutOfBoundsException

Exception: Exception is an unexpected event occurred in java applications at runtime , which may be provided by users while entering dynamic input in java applications, provided by the Database Engines while executing sql queries in Jdbc applications, provided by Network while establish connection between local machine and remote machine,.....causes abnormal termination to the java applications.

dynamic input -
user name - Rushikesh
user age - 20 but if we provide xyz

SQL Queries -
select from * emp;   // invalid syntax

Network connection
http://122.123.98.99/:1010 loginapp/login
122.123.98.99 - remote machine IP address

There are two types of terminations in java applications.

- ### Smooth Termination

Terminating program at end is called as Smooth termination.

- ### Abnormal Termination

Terminating program in the middle is called as Abnormal Termination.

In general, Exceptions are providing abnormal terminations, these abnormal terminations may crash local operating systems, these abnormal terminations may provide hanged out situations to the network,..... it may collapse database if it is datavase relation application, may server down if it is server side application

To overcome the above problems we have to handle exceptions properly, to handle exceptions properly we have to use "Exception Handling Mechanism".

Java is a Robust programming language, because,

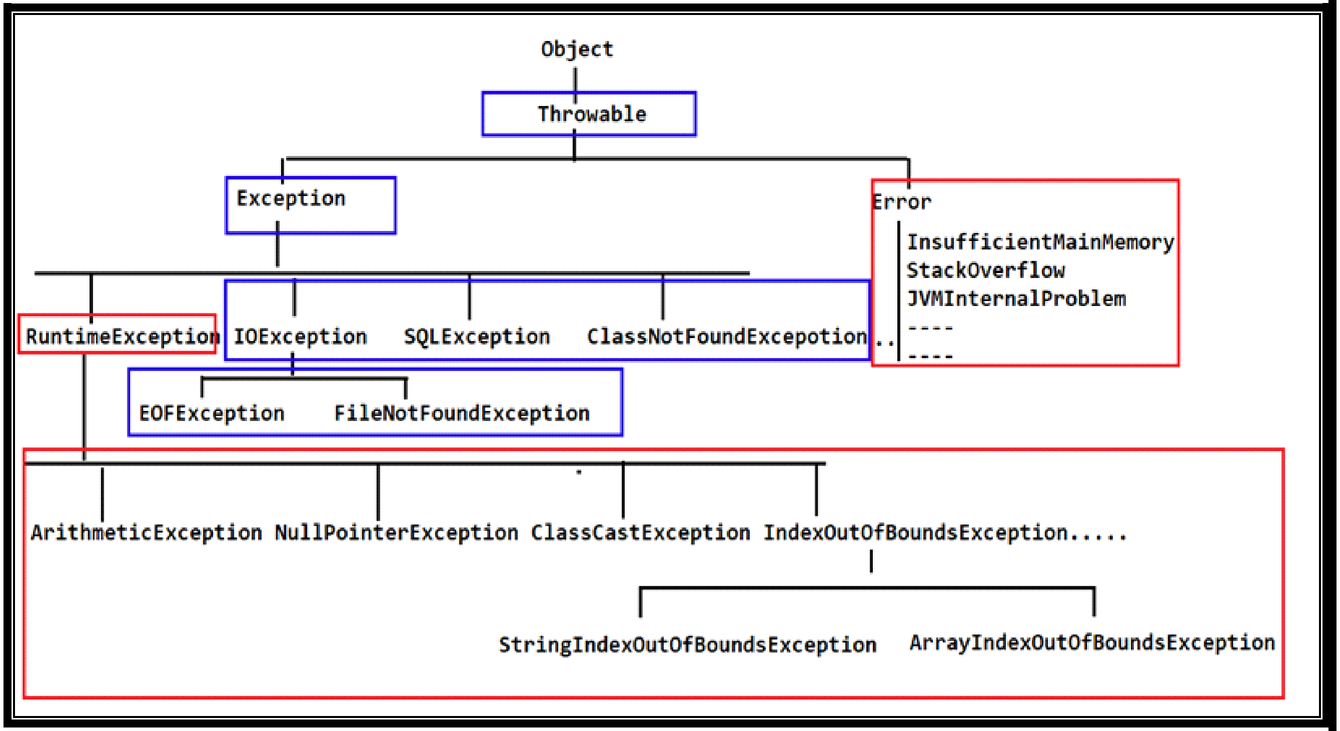
- **Java is having very good memory management system in the form of Heap Memory Management system, it is a dynamic memory management system, it allocates and deallocates memory for the objects at runtime.**
- **Java is having very good exception handling mechanisms, JAVA has provided very good predefined library to represent and handle almost all the frequently generated exceptions.**

There are two types of exceptions in Java.

- Predefined Exceptions
- User defined Exceptions

• Predefined Exceptions

These Exceptions are defined by JAVA programming language and provided along Java software.



There are two types predefined Exceptions.

- Checked Exceptions
- Unchecked Exceptions

all exception are avaiable in different packges related to topic

Q) What is the difference between *Checked Exceptions* and *Unchecked Exceptions?*

- Checked Exception is an exception recognized at compilation time, but, not occurred at compilation time.

- Unchecked Exceptions are not recognized at compilation time, these exceptions are recognized at runtime by JVM.

Note: In Exceptions Arch, RuntimeException and its sub classes and Error and its sub classes are the examples for Unchecked Exceptions and all the remaining classes are the examples for Checked Exceptions.

There are two types of Checked Exceptions
- Pure Checked Exceptions
- Partially Checked Exceptions

Q) What is the difference between *Pure Checked Exception* and *Partially Checked Exceptions?*

If any checked exception is having only checked exceptions as sub classes then this Exception is called as Pure Checked Exception.

EX: IOException

If any Checked Exception contains at least one sub class as unchecked exception then that Checked Exception is called as Partially Checked Exception.

EX: Exception, Throwable

Predefined Exceptions Overview:

• ArithmeticException

In Java applications, when we have a situation like a number is divided by zero then JVM will rise Arithmetic Exception.

EX:
```java
class Test
{
   public static void main(String[] args)
   {
      int i=10;
      int j=0;
      float f=i/j;
      System.out.println(f);
   }
}
```

If we run the above program then JVM will provide ArithmeticException with the following Exception message.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Text.main(Test.java:7)
```

The above exception message is devided into the following three parts.
- Exception Name: java.lang.ArithmeticException
- Exception Descrioptioun: / by zero
- Exception Location : Test.java:7

• NullPointerException:

In java applications, when we access any instance variable and instance methods by using a reference variable contains null value then JVM will rise NullPointerException.

Note - NullPointerException is not allowed for static variable and static methods over the references.

EX:

```
1)  class Test
2)  {
3)     public static void main(String[] args)
4)     {
5)        java.util.Date d=null;
6)        System.out.println(d.toString());
7)     }
8)  }
```

If we run the above code then JVM will provide the following exception details.

- **Exception Name: java.lang.NullPointerException**
- **Exception Description: --- No description----**
- **Exception Location: Test.java:6**

• ArrayIndexOutOfBoundsException:

In Java applications, when we insert an element to an array at a particular index value and when we are trying to access an element from an array at a particular index value and if the specified index value is in outside of the arrays size then JVM will rise an exception like "ArrayIndexOutOfBoundsException".

EX:

```
1)  class Test
2)  {
3)     public static void main(String[] args)
4)     {
5)        int [] a={1,2,3,4,5};
6)        System.out.println(a[10]);
7)     }
8)  }
```

If we run the above program then JVM will rise an exception with the following details.
Exception Name: java.lang.ArrayindexOutOfBoundsException
Exception Description: 10
Exception Location: Test.java: 6

Note - when we are trying insert an elements in an array on the basis of index value, where the provided index value is in out of the array indexes there JVM will raise ArrayIndexOutOfException

• ClassCastException:

In Java applications, we are able to keep sub class object reference value in super class reference variable, but, we are unable to keep super class object reference value in sub class reference variable. If we are trying to keep super class object reference value in sub class reference variable then JVM will rise an exception like "java.lang.ClassCastException".

EX:

```
1)  class A
2)  {
3)  }
4)  class B extends A
5)  {
6)  }
7)  class Test
8)  {
9)      public static void main(String[] args)
10)     {
11)         A a=new A();
12)         B b=(B)a;
13)     }
14) }
```

- **If we run the above code then JVM will rise an exception with the following details.**
- **Exception Name: java.lang.ClassCastException**
- **Exception Description: A cannot be cast to B**
- **Exception location: Test.java: 12**

• ClassNotFoundException:

In Java applications, if we want to load a particular class byte code to the memory without creating object then we will use the following method from java.lang.Class class.

public static Class forName(String class_Name)throws ClassNotFoundException

EX: Class c=Class.forName("A");

When JVM encounter the above instruction, JVM will search for A.class file at current location, at java predefined library and at the locations referred by "classpath" environment variable, if the required A.class file is not identified at all the locations then JVM will rise ClassNotFoundException.

EX:

```
1)  class A
2)  {
3)     static
4)     {
5)        System.out.println("Class Loading");
6)     }
7)  }
8)  class Test
9)  {
10)    public static void main(String[] args) throws Exception
11)    {
12)       Class c=Class.forName("AAA");
13)    }
14) }
```

If we run this code the JVM will provide the following exception details.

- Exception Name: java.lang.ClassNotFoundException
- Exception Description: AAA
- Exception Location: Test.java: 12
- InstantiationException 5.IllegalArgumentException

In java applications, if we load class byte code by using "Class.forName(-)" method then to create object explicitly we have to use the following method from java.lang.Class class.

public Object newInstance()

EX: Object obj = c.newInstance()

When JVM encounter the above code then JVM will search for 0-arg constructor and non-private constructor in the loaded class. if 0-arg constructor is not available, if parameterized constructor is existed then JVM will rise "java.lang.instantiationException". If non-private constructor is not available, if private constructor is available then JVM will rise "java.lang.IllegalAccessException".

EX1:

private constructor and parameterised constructor the JVM raise InstanciationException
(priority for 0 argument constructor compare with private constructor)

```
class A
{
   static
   {
      System.out.println("Class Loading");
   }
   A(int i)
```

```
8)    {
9)        System.out.println("Object Creating");
10)   }
11) }
12) class Test
13) {
14)   public static void main(String[] args) throws Exception
15)   {
16)       Class c=Class.forName("A");
17)       Object obj=c.newInstance();
18)   }
19) }
```

If run the above code then JVM will provide an exception with the following details.

- **Exception Name : java.lang.InstantiationException**
- **Exception Description: A**
- **Exception Location:Test.java: 17**

EX:

IllegalArgumentException -
Thread is a flow of execution to perform a particular task, for each and every there is a priority value , the default value - 5,
setPriorityValue(int priorityValue)
Range - 1 to 10

```
1)  class A
2)  {
3)    static
4)    {
5)        System.out.println("Class Loading");
6)    }
7)    private A()
8)    {
9)        System.out.println("Object Creating");
10)   }
11) }
12) class Test
13) {
14)   public static void main(String[] args) throws Exception
15)   {
16)       Class c=Class.forName("A");
17)       Object obj=c.newInstance();
18)   }
19) }
```

If we run the above code then JVM will rise an exception with the following details

- **Exception Name : java.lang.IllegalAccessException**
- **Exception Decrition: Test class cannot access the members of class A with the modifier "private".**
- **Exception Location" Test.java: 17**

'throw' keyword:

'throw' is a java keyword, it can be used to rise exceptions intentionally as per the developers application requirement.

In gerenral, developer will write program to implement their appliaction requirement , as part of implemeting application requirement there may be a chance to get exceptions how developers are not having any intension to raise the exception explicitly

Syntax:

throw new Exception_Name("----Exception Description-----");

EX:

```
class Test
{
   public static void main(String[] args)throws Exception
   {
      String accNo=args[0];
      String accName=args[1];
      int pin_Num=Integer.parseInt(args[2]);
      String accType=args[3];
      System.out.println("Account Details");
      System.out.println("--------------------");
      System.out.println("Account Number    :"+accNo);
      System.out.println("Account Name      :"+accName);
      System.out.println("Account Type      :"+accType);
      System.out.println("Account PIN Number:"+pin_Num);
      if(pin_Num>=1000 && pin_Num<=9999)
      {
         System.out.println("valid PIN Number");
      }
      else
      {
         throw new RuntimeException("Invalid PIN Number, enter Valid 4 digit PIN
Number");
      }
   }
}
```

```
D:\javaapps>javac Test.java
D:\javaapps>java Test abc123 AAA 1234 Ssavings
--- Account details without Exception-----

D:\javaapps>java Test abc123 AAA 123 Savings
--- Account details are displayed with  Exception-----

---Account details----
Exception in thread "main" java.lang.RuntimeException: Invalid PIN Number, enter valid 4
digit PIN number at Test.main(Test.java:17)
```

There are two ways to handle exceptions.

- By using 'throws' keyword.
- By using try-catch-finally block.

- **'throws' keyword:**
- It is a Java keyword, it can be used to bypass the generated exception from the present method or constructor to the caller method (or) constructor.

- In Java applications, 'throws' keyword will be used in method declarations, not in method body.

- In Java applications, "throws" keyword allows an exception class name, it should be either same as the generated exception or super class to the generated exception. It should not be subclass to the generated Exception.
- 'throws' keyword allows more than one exception in method prototypes.
- In Java applications, "throws" keyword will be utilized mainly for checked exceptions.

EX:
```
void m1() throws RuntimeException {
        throw new ArithmeticException();
}
```

Status:Valid

EX:
```
void m1() throws FileNotFoundException {
         throw new IOException();
}
```

Status:InValid

EX:
```
Void m1() throws NullPointerException, ClassNotFoundException {
}
```

Status:Valid

EX:
```
Void m1() throws IOException, FileNotFoundException
{
}
```

Status: Valid, Not Suggestible

If we specify any super exception class along with throws keyword, then it is not necessary to specify any of its child exception classes along with "throws" keyword.

<u>NOTE:</u> In any Java method, if we call some other method which is bypassing an exception by using "throws" keyword, then we must handle that exception either by using "throws" keyword in the present method [Caller Method] prototype or by using "try-catch-finally" in the body of the present method [Caller method].

<u>EX:</u>

```
Void m1() throws Exception {
        -----
        ------
}
Void m2() {
        try {
                m1();
        }
        catch(Exception e) {
                e.printStackTrace();
        }
}
```

<u>EX:</u>

```
void m1() throws Exception {
        ----
}
void m2() throws Exception {
        m1();
}
```

<u>EX:</u>

```
import java.io.*;
class A {
        void add() throws Exception {
                concat();
        }
        void concat() throws IOException {
                throw new IOException();
        }
}
class Test {
        public static void main(String args[]) throws Throwable {
                A a = new A();
                a.add();
        }
}
```

Internal Flow:

If we execute the above program, then JVM will recognize throw keyword in concat() method and JVM will rise an exception in concat() method, due to throws keyword in concat() method prototype, Exception will be bypassed to concat() method call that is in add(), due to throws keyword in add() method Exception will be bypassed to add() method call , that is , in main() method, Due to 'throws' keyword in main() method Exception will be bypassed to main() method call , that is, to JVM, where JVM will activate "Default Exception Handler" , where Default Exception Handler will display exception details.

Q) What are the differences between *"throw"* and *"throws"* Keywords?

- 'throw' keyword can be used to rise the exceptions intentionally as per the application requirement.
  'throws' keyword is able to bypass the exception from the present method to the caller method.
- 'throw' keyword will be utilized in method body.
  'throws' keyword will be used in method declarations or in method prototype (or) in method header part.
- 'throw' keyword allows only one exception class name.
  'throws' keyword allows more than one exception class name.

try-catch-finally:

In Java application "throws" keyword is not really an exception handler, because "throws" keyword will bypass the exception handling responsibility from present method to the caller method.
If we want to handle the exceptions, the locations where exceptions are generated then we have to use "try-catch-finally".

Syntax:

```
try {
  ----
}
catch(Exception_Name  e) {
    ----
}
finally {
   ----
}
```

Where the purpose of try block is to include a set of exceptions, which may rise an exception.

If JVM identify any exception inside "try" block then JVM will bypass flow of execution to "catch" block by skipping all the remaining instructions in try block and by passing the generated Exception object reference as parameter.

If no exception is identified in "try" block then JVM will execute completely "try" block, at the end of try block, JVM will bypass flow of execution to "finally" block directly.

The main purpose of catch block is to catch the exception from try block and to display exception details on command prompt.

To display exception details on command prompt, we have to use the following three approaches.

- e.printStackTrace()
- System.out.println(e):
- System.out.println(e.getMessage());

- **e.printStackTrace():**

It will display the exception details like Exception Name, Exception Description and Exception Location.

- **System.out.println(e):**

If we pass Exception object reference variable as parameter to System.out.println(-) method then JVM will access Exception class toString() method internally, it will display the exception details like Exception name, Exception description.

- **System.out.println(e.getMessage()):**

Where getMessage() method will return a String contains the exception details like only Description of the exception.

EX:

```
class Test {
    public static void main(String args[]) {
        try {
          throw new ArithmeticException("My Arithmetic Exception");
        }
        catch(ArithmeticException e) {
              e.printStackTrace();
              System.out.println();
              System.out.println(e);
              System.out.println();
              System.out.println(e.getMessage());
        }
        finally {
```

```
        }
    }
}
```

Output:

```
java.lang.ArithmeticException:My Arithmetic Exception
at Test.main(Test.java:7)

java.lang.ArithmeticException:My Arithmetic Exception

My Arithmetic Exception
```

Where the main purpose of finally block is to include some Java code , it will be executed irrespective of getting exception in "try" block and irrespective of executing "catch" block.

Q) What is the difference between "final","finally" and "finalize" in JAVA?

- "final" is a keyword it can be used to declare constant expressions.

There are three ways to use final keyword in java applications.

a) final variable: It will not allow modifications over its value.
b) final methods: It will not allow method overriding.
c) final class: It will not allow inheritance, that is, sub classes.

- finally block: It is part of try-catch-finally syntax, it will include some instructions, which must be executed by JVM irrespective of getting exception from try block and irrespective of executing catch block.

- finalize (): It is a method in java.lang.Object class, it will be executed just before destroying objects in order to give final notification to the user about to destroy objects.

Q) Find the output from the following programs?

```
class Test {
    public static void main(String args[]) {
        System.out.println("Before Try");
        try {
            System.out.println("Inside Try");
        }
        catch(Exception e) {
            System.out.println("Inside Catch");
        }
        finally {
```

```
            System.out.println("Inside Finally");
        }
        System.out.println("After Finally");
    }
}
```

Output:

```
Before try
Inside try
Inside finally
After finally
```

EX:

```
class Test {
    public static void main(String args[]) {
        System.out.println("Before Try");
        try {
            System.out.println("Before Exception in try");
            float f = 100/0;
            System.out.println("After Exception in try");
        }
        catch(Exception e) {
            System.out.println("Inside Catch");
        }
        finally {
                System.out.println("Inside Finally");
        }
        System.out.println("After Finally");
    }
}
```

Output:

```
Before try
Before exception in try
Inside catch
Inside finally
After finally
```

Q) Find The Output From The Following Program?

```
class A {
   int m1() {
      try {
         return 10;
      }
      catch(Exception e) {
          return 20;
      }
      finally {
          return 30;
      }
   }
}
class Test {
      public static void main(String args[]) {
          A a = new A();
          int val = a.m1();
          System.out.println(val);
      }
}
```

Output:

```
30
```

NOTE: finally block provided return statement is the finally return statement for the method

Q) Is it possible to provide "try" Block without "catch" Block?

Yes, it is possible to provide try block without catch block but by using "finally" Block.

Syntax:

```
try {
}
finally {
}
```

EX:

```
class Test {
    public static void main(String args[]) {
        System.out.println("Before try");
        try {
            System.out.println("Before Exception inside try");
            int i = 100;
            int j-0;
            float f = i/j;
            System.out.println("After Exception inside try");
        }
        finally {
            System.out.println("Inside finally");
        }
        System.out.println("After Finally");
    }
}
```

Status: No Compilation Error.

Output:

```
Before try
Before exception inside try
Inside finally
Exception in thread “main” java.lang.ArithmeticException:/by zero
  at Test.main(Test.java:11)
```

REASON: When JVM encounter exception in try Block, JVM will search for catch Block, if no catch block is identified, then JVM will terminate the program abnormally after executing finally block.

Q) Is it possible to provide "try" Block without "finally" Block?

Yes, it is possible to provide "try" block without "finally" block but by using "catch" block.

Syntax:

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

Q) Is it possible to provide try-catch-finally

a) inside try block,
b) inside catch block and
c) inside finally block

Yes, it is possible to provide try-catch-finally inside try block, inside catch block and inside finally block.

finally block -
1) it will include a set of instruction to execute irrespective of getting exception in try block and irrespective of executing catch block
2) in general in java applications we will use some resources like database connections, input output stream as per the requiremnet resource may raise some exception when we perform operations with resources
to handle this exception if we you try catch finally approach
3) then we have to open resources in side try block and we must close the resources inside the finally blocks
4) finally block is giving the gaurantee of execution

Syntax-1:
```
try {
   try {
   }
   catch(Exception e) {
   }
   finally {
   }
}
catch(Exception e) {
}
finally {
}
```

Syntax-2:
```
try {
}
catch(Exception e) {
    try {
    }
    catch(Exception e)
    {
    }
    finally {
    }
}
finally {
}
```

Syntax-3:
```
try {
}
catch(Exception e) {
}
finally {
    try {
```

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

Q) Is it possible to provide more than one catch Block for a Single try Block?

Yes, it is possible to provide more than one catch block for a single try block but with the following conditions.

- If no inheritance relation existed between exception class names which are specified along with catch blocks then it is possible to provide all the catch blocks in any order. If inheritance relation is existed between exception class names then we have to arrange all the catch blocks as per Exception classes inheritance increasing order.
- In general, specifying an exception class along with a catch block is not giving any guarantee to rise the same exception in the corresponding try block, but if we specify any pure checked exception along with any catch block then the corresponding "try" block must rise the same pure checked exception.

Ex 1:

```
try {
}
catch(ArithmeticException e) {
}
catch(ClassCastException e) {
}
catch(NullPointerException e) {
}

Status:Valid Combination
```

Ex 2:

```
try {
}
catch(NullPointerException e) {
}
catch(ArithmeticException e) {
}
catch(ClassCastException e) {
}

    status:Valid Combination
```

Ex 3:

```
try {
}
catch(ArithmeticException e) {
}
catch(RuntimeException e) {
}
catch(Exception e) {
}
```

Status:Valid

Ex 4:

```
try {
}
catch(Exception e) {
}
catch(RuntimeException e) {
}
catch(ArithmeticException e) {
}
```

status: Invalid

Ex 5:

```
try {
throws new ArithmeticException("My Exception");
}
catch(ArithmeticException e) {
}
catch(IOException e) {
}
catch(NullPointerException e) {
}
```

Status:Invalid

Ex 6:

```
try {
throw new IOException("My Exception");
}
catch(ArithmeticException e) {
}
catch(IOException e) {
}
```

```
catch(NullPointerException e) {
}
```

status:Valid

Custom Exceptions/User Defined Exceptions:

Custom Exceptions are the exceptions, which would be defined by the developers as per their application requirements.
If we want to define user defined exceptions then we have to use the following steps:

- **Define User defined Exception Class:**

  To declare user-defined Exception class, we have to take an user-defined class, which must be extended from java.lang.Exception class.

```
Class MyException extends Exception
{
}
```

- **Declare a String Parametrized Constructor in User-Defined Exception Class and Access String Parametrized Super Class Constructor by using "super" Keyword:**

```
class MyException extends Exception {
        MyException(String err_Msg) {
                super(err_Msg);
        }
}
```

Create and Rise Exception in Java Application as per the Application Requirement:

```
try {
        throw new MyException("My Custom Exception");
}
catch(MyException me) {
      me.printStackTrace();
}

class InsufficientFundsException extends Exception
{
   InsufficientFundsException(String err_Msg)
   {
      super(err_Msg);
   }
}
```

```java
class Account
{
    String accNo;
    String accName;
    String accType;
    int balance;

    Account(String accNo, String accName, String accType, int balance)
    {
        this.accNo=accNo;
        this.accName=accName;
        this.accType=accType;
        this.balance=balance;
    }
}
class Transaction
{
    public void withdraw(Account acc, int wd_Amt)
    {
        try
        {
            System.out.println("Transaction Details");
            System.out.println("------------------------");
            System.out.println("Account Number   :"+acc.accNo);
            System.out.println("Account Name     :"+acc.accName);
            System.out.println("Account Type     :"+acc.accType);
            System.out.println("Transaction Type :WITHDRAW");
            System.out.println("Withdraw Amount  :"+wd_Amt);
            if(acc.balance>wd_Amt)
            {
                acc.balance=acc.balance-wd_Amt;
                System.out.println("Total Balance    :"+acc.balance);
                System.out.println("Transaction Status:SUCCESS");
            }
            else
            {
                System.out.println("Total Balance    :"+acc.balance);
                System.out.println("Transaction Status:FAILURE");
                throw new InsufficientFundsException("Funds are not Sufficient in your Account, please enter valid Withdraw Amout");
            }
        }
        catch (InsufficientFundsException e)
        {
            System.out.println(e.getMessage());
```

```
        }
        finally
        {
            System.out.println("**********ThanQ, Visit Again*********");
        }
    }
}
class Test
{
    public static void main(String[] args)
    {
        Account acc1=new Account("abc123", "Durga", "Savings", 10000);
        Transaction tx1=new Transaction();
        tx1.withdraw(acc1, 5000);
        System.out.println();
        Account acc2=new Account("xyz123", "Anil", "Savings", 10000);
        Transaction tx2=new Transaction();
        tx2.withdraw(acc2, 15000);
    }
}
```

JAVA 7 Features in Exception Handling:

- Multi Catch block
- try-with-Resources/Automatic Resources Management/Auto close able Resources

• Multi Catch Block:

Consider the below Syntax

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

If we specify "Exception" class along with catch block then it able to catch and handle all the exceptions which are either same as Exception or child classes to Exception, this approach will not handling exceptions individually, it will handle all the exceptions in the common way.

If we want to handle to the Exceptions separately then we have to use multiple catch blocks for a single try block.

```
try {
} catch(ArithmeticException e) {
  } catch(NullPointerException e) {
    } catch(ClassCastException e) {
```

}

If we use this approach then number of catch blocks are increased.
In Java applications, if we want to handle all the exceptions separately and by using a single catch block then we have to use "JAVA7" provided multi- catch block.

Syntax:

```
try {
}
catch(Exception1 | Exception2 |....|Exception-n e) {
}
```

Where Exception1, Exception2....must not have inheritance relation otherwise Compilation error will be raised.

EX:

```
class Test {
    public static void main(String args[]) {
        try {
                /* int a = 10;
                   int b = 0;
                   float c = a/b;
                */
                /*java.util.Date d = null;
                  System.out.println(d.toString());
                */
                 int[] a = {1, 2, 3, 4, 5};
                 System.out.println(a[10]);
        }
        catch(ArithmeticException | NullPointerException |
ArrayIndexOutOfBoundsException e) {
             e.printStackTrace();
        }
}
```

try-With-Resources/Auto Closeable Resources:

In general, in Java applications, we may use the resources like Files, Streams, Database Connections....as per the application requirements.

If we want to manage the resources along with try-catch-finally in Java applications then we have to use the following conventions.

- Declare all the resources before "try" block.
- Create the resources inside "try" block.
- Close the resources inside finally block.

The main intention to declare the resources before "try" block is to make available resources variables to "catch" block and to "finally" block to use.

If we want to close the resources in "finally" block then we have to use close() methods, which are throwing some exceptions like IOException, SQLException depending on the resource by using "throws" keyword, to handle these exceptions we have to use "try-catch-finally" inside "finally" block.

```
//Declare the resources
File f = null;
BufferedReader br = null;
Connection con=null;
try {
        //create the resources
            f = new File("abc.txt");
            br = new BufferedReader(new InputStreamReader(System.in));
            con = DriverManager.getConnection("jdbc:odbc:nag","system","durga");
            -----
            -----
}
catch(Exception e) {
}
finally {
//close the resources
try {
          f.close();
          br.close();
          con.close();
} catch(Exception e) {
         e.printStackTrace();
  }
}
```

To manage the resources in Java applications, if we use the above convention then developers have to use close() methods explicitly, Developers have to provide try-catch-finally inside "finally" block, this convention will increase number of instructions in Java applications.
To overcome all the above problems, JAVA 7 version has provided a new Feature in the form of "Try-With-Resources" or "Auto Closeable Resources".

In the case of "Try-With-Resources", just we have to declare and create the resources along with "try"[not inside try block, not before try block] and no need to close these resources inside the finally block, why because, JVM will close all the resources automatically when flow of execution is coming out from "try" block.

In the case of "Try-With-Resources",it is not required to close the resources explicitly, it is not required to use close() methods in finally block explicitly, so that it is not required to use "finally" block in try-catch-finally syntax.

Synatx:

```
try(Resource1; Resource2;........Resource-n) {
     -------
     ------
}
catch(Exception e) {
       -----
       -----
}
```

Where all the specified Resources classes or interfaces must implement "java.io.AutoCloseable" interface.

Where if we declare resources as AutoCloseable resources along with "try" then the resources reference variables are converted as "final" variables.

EX:

```
try(File f = new File("abc.txt");
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
Connection con = DriverManager.getConnection("jdbc:odbc:nag","system","durga");)
{
      -------
      -------
}
catch(Exception e) {
      e.printStackTrace();
}
```

NOTE: In Java, all the predefined Stream classes, File class, Connection interface are extends/implemented "java.io.AutoCloseable" interface predefined.