Exception Handling

Exception is a run-time error which arises during the execution of java program. The term exception in java stands for an **“exceptional event”**.

So Exceptions are nothing but some abnormal and typically an event or conditions that arise during the execution which may interrupt the normal flow of program.

An exception can occur for many different reasons, including the following:

A user has entered invalid data.

A file that needs to be opened cannot be found.

A network connection has been lost in the middle of communications, or the JVM has run out of memory.

"If the exception object is not handled properly, the interpreter will display the error and will terminate the program.

Now if we want to continue the program with the remaining code, then we should write the part of the program which generate the error in the try{} block and catch the errors using catch() block.

Exception turns the direction of normal flow of the program control and send to the related catch() block and should display error message for taking proper action. This process is known as."

Exception handling

The purpose of exception handling is to detect and report an exception so that proper action can be taken and prevent the program which is automatically terminate or stop the execution because of that exception.

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

Try: Piece of code of your program that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown.

Catch: Catch block can catch this exception and handle it in some logical manner.

Neeraj Kumar

Throw: System-generated exceptions are automatically thrown by the Java run-time system. Now if we want to manually throw an exception, we have to use the throw keyword.

Throws: If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.

You do this by including a throws clause in the method's declaration. Basically it is used for IOException. A throws clause lists the types of exceptions that a method might throw.

This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.

All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

Finally: Any code that absolutely must be executed before a method returns, is put in a finally block.

General form:

```
try {
        // block of code to monitor for errors
}
catch (ExceptionType1  e1) {
        // exception handler for ExceptionType1
}
catch (ExceptionType2 e2) {
        // exception handler for ExceptionType2
}
// ...
finally {
        // block of code to be executed before try block
ends
}
```

The Exception class has two main subclasses:

(1) IOException or Checked Exceptions class and

(2) RuntimeException or Unchecked Exception class

(1) IOException or Checked Exceptions :

Exceptions that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*.

For example, if a file is to be opened, but the file cannot be found, an exception occurs.

These exceptions cannot simply be ignored at the time of compilation.

Java's Checked Exceptions Defined in java.lang

(2) RuntimeException or Unchecked Exception :

Exceptions need not be included in any method's **throws** list. These are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.

As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

Java's Unchecked RuntimeException Subclasses

Try And Catch

We have already seen introduction about try and catch block in java exception handling.

Now here is the some examples of try and catch block.

EX :

```
public class TC_Demo
{
   public static void main(String[] args)
   {
      int a=10;       int b=5,c=5;       int x,y;
      try    {
         x = a / (b-c);
      }
      catch(ArithmeticException e){
         System.out.println("Divide by zero");
      }
      y = a / (b+c);
      System.out.println("y = " + y);
```

Output :

Divide by zero
y = 1

Multiple catch blocks :

It is possible to have multiple catch blocks in our program.

EX :

```
public class MultiCatch
{
   public static void main(String[] args)
   {
      int a [] = {5,10};   int b=5;
      try       {
         int x = a[2] / b - a[1];
      }
      catch(ArithmeticException e)       {
         System.out.println("Divide by zero");
      }
      catch(ArrayIndexOutOfBoundsException e)       {
         System.out.println("Array index error");
      }
      catch(ArrayStoreException e)       {
         System.out.println("Wrong data type");
      }
      int y = a[1]/a[0];
      System.out.println("y = " + y);
   }          }
```

Output :

Array index error
y = 2

Neeraj Kumar

```
class etion3
{
   public static void main(String args[])
   {
      int num1 = 100;
      int num2 = 50;
      int num3 = 50;
      int result1;

      try
      {
         result1 = num1/(num2-num3);
         System.out.println("Result1 = " + result1);
      }

      catch (Exception e)
      {
         System.out.println("This is mistake. ");
      }
      catch(ArithmeticException g)
      {
         System.out.println("Division by zero");

      }
   }
}
```

Output :

Array index error
y = 2

Nested try statements :

The try statement can be nested.

That is, a try statement can be inside a block of another try.

Each time a try statement is entered, its corresponding catch block has to entered.

The catch statements are operated from corresponding statement blocks defined by try.

EX :

```
public class NestedTry
{
    public static void main(String args[])
    {
        int num1 = 100;
        int num2 = 50;
        int num3 = 50;
        int result1;
        try     {
            result1 = num1/(num2-num3);
            System.out.println("Result1 = " + result1);
            try   {
                result1 = num1/(num2-num3);
                System.out.println("Result1 = " + result1);
            }
            catch(ArithmeticException e)
            {
                System.out.println("This is inner catch");
            }
        }
        catch(ArithmeticException g)
        {
            System.out.println("This is outer catch");
        }
    }
}
```

Output :

This is outer catch

Neeraj Kumar

Finally

Java supports another statement known as **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements.

We can put finally block after the try block or after the last catch block.

The finally block is executed in all circumstances. Even if a try block completes without problems, the finally block executes.

EX :

```
public class Finally_Demo
{
    public static void main(String args[])
    {
        int num1 = 100;
        int num2 = 50;
        int num3 = 50;
        int result1;

        try
        {
            result1 = num1/(num2-num3);
            System.out.println("Result1 = " + result1);
        }
        catch(ArithmeticException g)
        {
            System.out.println("Division by zero");

        }
        finally
        {
            System.out.println("This is final");
        }
    }
```

Output :

Division by zero
This is final

Throw

We saw that an exception was generated by the JVM when certain run-time problems occurred.
It is also possible for our program to explicitly generate an exception.

This can be done with a throw statement. Its form is as follows:

Throw object;

Inside a catch block, you can throw the same exception object that was provided as an argument.

This can be done with the following syntax:

```
catch(ExceptionType object)
{
        throw object;
}
```

Alternatively, you may create and throw a new exception object as follows:

```
Throw new ExceptionType(args);
```

Throws

If a method is capable of causing an exception that it does not handle,it must specify this behavior so that callers of the method can guard themselves against that exception.

You do this by including a throws clause in the method's declaration. Basically it is used for IOException.

A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error or RuntimeException**,or any of their subclasses.

```
class customexception
{
   public static void main(String args[])
   {
      try
      {
         doWork(3);
         doWork(2);
         doWork(1);
         doWork(0);
      }
      catch (NewException e)
      {
         System.out.println("Exception : " + e.toS());
      }
   }
   static void doWork(int value) throws NewException
   {
      if (value == 0)
      {
         throw new NewException();
      }
      else
      {
         System.out.println("****No Problem.****");
      }
   }
}
```

Output :

****No Problem.****
****No Problem.****
****No Problem.****
Exception : You are in NewException

Neeraj Kumar