Exception Handling in Java

We have already discussed exception handling in C++. Basic concepts are similar in Java. Here we will only discuss how exception in handled in Java.

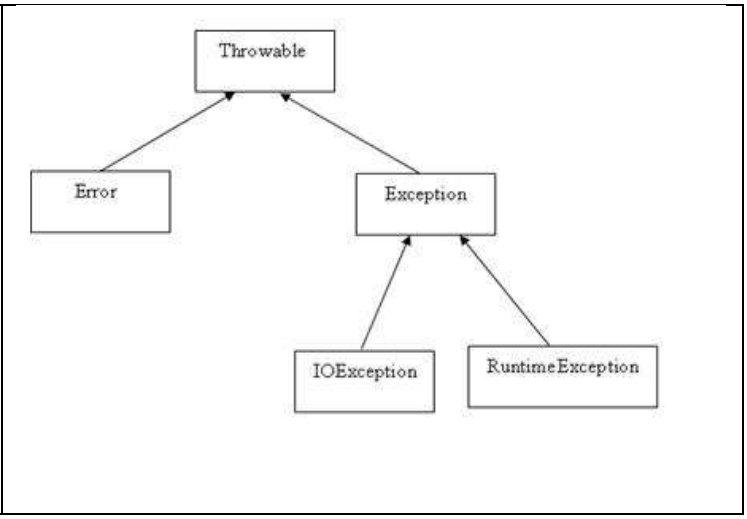
We know that Exception handling is a process of handling exceptional situations in such a way that:

- The program will terminate gracefully i.e. it will give a proper message and then will terminate the program.
- Java supports both checked (compile time) and unchecked (runtime) exceptions.

Exception object Hierarchy in Java

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.
The Exception class has two main subclasses: IOException class and RuntimeException Class.



Here is a list of most common checked and unchecked Java's Built-in Exceptions.

Exceptions Methods:

Following is the list of important methods available in the **Throwable** class.

SN	Methods with Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage()

4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Catching Exceptions in Java:

A **try/catch** block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  //Protected code
}catch(ExceptionName e1)
{
  //Catch block
}
```

- A catch statement involves declaring the type of exception.
- If an exception occurs in protected code, the catch block (or blocks) that follow the try is checked.
- Control will pass to the catch block of the type of exception that occurred in try block.

Example:
The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{

  public static void main(String args[]){
    try{
      int a[] = new int[2];
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Exception thrown  :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

This would produce the following result:

```
Exception thrown
:java.lang.ArrayIndexOutOfBoundsEx
ception: 3
```

Multiple catch Blocks: A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{  //Protected code }
catch(ExceptionType1 e1) {    //Catch block }
catch(ExceptionType2 e2) {    //Catch block }
catch(ExceptionType3 e3) {    //Catch block }
```

- If an exception occurs the exception is thrown to the first catch block in the list.
- If the type of the exception thrown matches ExceptionType1, it gets caught there.
- If not, the exception passes down to the second catch statement.
- This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the **previous method on the call stack.**

Checked and Unchecked Exception Handling in Java

Checked exception must be handled in the code. Let's understand this with this **example**:

In the following example we are reading the file myfile.txt and displaying its content on the screen.

As per the program there are three places where a checked exception can occur.

- 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 throws FileNotFoundException which is a checked exception
      fis = new FileInputStream("B:/myfile.txt");
      int k;
      while(( k = fis.read() ) != -1)      //  read() method also throws a checked exception: IOException
      {
            System.out.print((char)k);
      }
      fis.close();       //   The close() method throws IOException
  }
}
```

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? Checked exceptions get 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.

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.

Our code may be throwing FileNotFoundException and IOException both. But in below code 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 that too 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();
  }
}
```

So no compilation error as encountered earlier.

Output:
File content is displayed on the screen.

It is not a best exception handling practice.

Method 2: Handle them using try-catch blocks.

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");
```

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

```
        } 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);
        }
   }
}
```

Unchecked exceptions / runtime exception : 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.

```
class Example {
  public static void main(String args[])
  {
      int num1=10;
      int num2=0;
      /* Since 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.

Note: Compiler is not checking these exceptions do not mean that we shouldn't handle them. In fact we should handle them more carefully.

The throws/throw Keywords in Java

- The ***throws*** keyword is used to **declare** explicitly an exception. It means an exception may occur so it is better for the programmer to provide the exception handling code in caller.

  ```
  public void sample() throws IOException, SQLException
  {
    //Statements
  }
  ```

  Syntax of throws

- If a method is using throws clause along with few exceptions then this implicitly tells other methods that – " If you call me, you must handle these exceptions that I throw".

- When defining a method you must include a throws clause to declare those exceptions that might

be thrown but doesn't get caught in the method.

- The ***throw*** keyword is used to explicitly throw an exception (newly instantiated one).
  We can throw either **checked** or **unchecked exception.** The ***throw*** keyword is mainly used to throw **custom exception**, which we cover next.

Example of Java throws Clause

```
class Demo
{
  static void throwMethod() throws NullPointerException
  {
    System.out.println ("Inside throwMethod");
    throw new NullPointerException ("Demo");
  }
  public static void main(String args[])
  {
    try
    {
      throwMethod();
    }
    catch (NullPointerException exp)
    {
      System.out.println ("The exception get caught" + exp);
    }
  }
}
```

The output of the above program is:
Inside throwMethod
The exception get caught
java.lang.IllegalAccessException: Demo

Note : if the method code generates some exception during execution of other statement, you need not explicitly write the throw new … statement

Examples :

```
class Test
{
   static void  check()  throws  ArithmeticException
   {
       System.out.println("Inside check function");
       throw new ArithmeticException("demo");
   }
   public static void main( String args[] )
   {
       try {   check();   }
       catch( ArithmeticException e )
       {
           System.out.println("caught" + e);
       }
   }
}
```

ArithmeticException : Already declared exception in java.

Output :
Inside check function
caught java.lang.ArithmeticException: demo

Program which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Simple{
 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[]){
     Simple obj=new Simple();
     obj.p();
     System.out.println("normal flow...");
 }
}
```

***Rule*: If you are calling a method that declares an exception, you must either caught or declare the exception.**
Here in method p() we call a method n() which declare the exception. Again method n() call the method m() which again declare the exception.

Output:
exception handled
normal flow...

- In case you declare the exception, if exception does not occur, the code will be executed fine.
- In case you declare the exception and not handled anywhere : if exception occurs, an exception will be thrown at runtime because throws does not handle the exception. See following example :

Program if exception occurs

```
import java.io.*;
class Test{
  void method() throws IOException {
        throw new IOException("device error");
  }
}
```

```
class Mth{
  public static void main(String args[])
            throws IOException{ //declare exception
   Test t=new Test();
   t.method();
   System.out.println("normal flow...");
  }
}
```

Note : As the exception is not handled in main Method, but declared again, it will be handled by **runtime**.

throw keyword can also be used to break a switch statement without using break keyword as shown in below example:

```
int number = 5;
switch(number) {
        case 1:
          throw new RuntimeException("Exception number 1");
        case 2:
          throw new RuntimeException("Exception number 2");
```

}

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:

throw	throws
It is used to explicitly throw an exception.	This keyword is used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

User defined exception in java

User defined exceptions in java are also known as **Custom exceptions**. Most of the times the standard exceptions already defined are more than sufficient. If we need any which is not covered under standard exception we can also define it.

Example 1 :

```
class MyException extends Exception{
   String str1;
   MyException(String str2) {
     str1=str2;
   }
   public String toString(){
     return ("Output String = "+str1) ;
   }
}

class CustomException{
   public static void main(String args[]){
     try{
          // Throwing user defined exception
        throw new MyException("Custom");
     }
     catch(MyException exp){
       System.out.println("Hi this is my catch
            block") ;
       System.out.println(exp) ;
     }
   }
```

Example : 2

```
public class MyException extends Exception
{
   public MyException(String mymsg)
   {
      super(mymsg);
   }
}
public class ExceptionSample
{
   public static void main(String args[])
                        throws Exception
   {
      ExceptionSample es = new ExceptionSample();
      es.displayMymsg();
   }
   public void displayMymsg() throws MyException
   {
      for(int j=8;j>0;j--)
      {
        System.out.println("j= "+j);
        if(j==7)
        {
```

} Output: Hi this is my catch block Output String = Custom • User defined exception needs to **inherit (extends)** Exception class in order to **act as an exception**. • throw keyword is used to throw such exceptions.	throw new MyException("This is my own Custom Message"); } } } } Output: j = 8 j = 7 Exception in thread "main" MyException: This is my own Custom Message at ExceptionSample.displayMymsg(ExceptionSample.java.19) ...

The finally Keyword

- The finally keyword is used to create a block of code that follows a try block.
- A finally block of code always executes, whether or not an exception has occurred.
- Generally this block allows us to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try {  //Protected code  }
catch(ExceptionType1 e1) {    //Catch block  }
catch(ExceptionType2 e2) {    //Catch block  }
catch(ExceptionType3 e3) {    //Catch block  }

finally
{
  //The finally block always executes.
}
```

Example:

```
public class ExcepTest{

  public static void main(String args[]){
    int a[] = new int[2];
    try{
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Exception thrown  :" + e);
    }
```

Output:
Exception thrown :java.lang.ArrayIndexOutOfBounds Exception: 3
First element value: 6
The finally statement is executed

```
    finally{
      a[0] = 6;
      System.out.println("First element value: " +a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

Note the following:

- A catch clause cannot exist without a try statement.
- The finally clauses are not mandatory.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

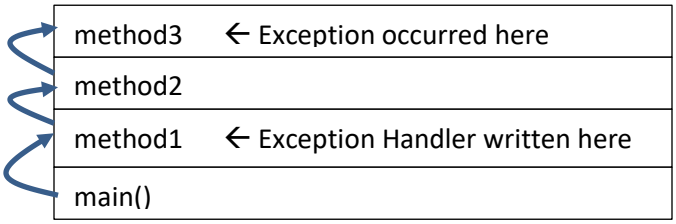
Exception Propagation in Java

When the exception occurs and is thrown to the runtime system,

- it is first thrown from the top of the stack to be caught and
- if not caught then it drops down the call stack to the previous method and
- if again it is not caught then it drops down to the previous methods and so on.

This process will go until they are caught or until they reach the bottom of the call stack. This process can be shown by the figure.

Here the main method calls the method1, method1 calls method2 and method2 calls method3. So the method3 will the on the top of call stack.



Basically exception propagation is held in the following two ways:

- Checked Exception
- Unchecked Exception

Example of Exception propagation: Unchecked Exception

In the calling chain of method calls, unchecked exceptions are forwarded by default.

```
class DivNo  {
    void method3() {    int error=1000/0;    // Exception generation   }
    void method2() {    method3();  }
    void method1() {
        try  {
                method2();
        }
         catch(Exception e)  {
                System.out.println("Exception is handled here");
         }
    }
    public static void main(String args[])
```

```
    {
        DivNo  dvn=new DivNo();
        dvn.method1();
       System.out.println("working in normal flow");
    }
}
```

Output:
Exception is handled here
working in normal flow

So we can see that

1. exception is occurred in the method3() and in method3() we don't have any exception handler.
2. Uncaught exception will be propagated downward in stack i.e it will check appropriate exception handler in the method2().
3. Again in method2 we don't have any exception handler then again exception is propagated downward to method1() where it finds exception handler

Thus we can see that uncaught exception is propagated in the stack until stack becomes empty, this propagation of uncaught exception is called as **Exception Propagation**.

Example of Exception propagation: checked Exception

In the calling chain of method calls, checked exceptions are "not" forwarded by default.

```
class ExceptionPropagation{
  void method3(){
     throw new java.io.IOException("Checked Exception..");
  }
  void method2() {  method3();  }
  void method1(){
     try{
          method2();
     } catch(Exception e){
          System.out.println("Exception is handled here");
     }
  }
  public static void main(String args[]){
          ExceptionPropagation obj=new ExceptionPropagation();
          obj.method1();
          System.out.println("Continue with Normal Flow...");
  }
}
```

Output :
Compile Time Error

You must remember one rule of thumb that – "**Checked Exceptions are not propagated in the chain by default**". thus we will get compile error.

Here IOException is a checked exception.

Some rules for declaring exceptions

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

Checked Exception declared in subclass

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

Output:Compile Time Error

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

Unchecked Exception declared in subclass

```
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();
  }
}
```

Unchecked exception declaration in overridden method **msg**.

```
Output:child
```

2. **Rule: If the superclass method declares an exception, subclass overridden method can declare same as superclass exception , sub class of Exception object or no exception but cannot declare parent exception.**

Example -1 Declaring parent of superclass exception : not permitted

```
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

Because Exception is parent of all exceptions.

```
 }
}
```

Example -2 Declaring same superclass exception : permitted

```
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
Because Same exception is declared in subclass.

Example -3 subclass overridden method declares subclass of Exception : permitted

```
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
Because ArithmeticException is a sub class of Exception