Exception Handling

By:Avinash Srivastava

What is an exception?

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.

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.

Exception Handling

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:
An exception generated by the system is given below

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at ExceptionDemo.main

(ExceptionDemo.java:5)

 ExceptionDemo : The class name

 main : The method name

 ExceptionDemo.java : The filename

 java:5 : Line number
```

Difference between error and exception

Errors indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:

NullPointerException – When you try to use a reference that points to null.

ArithmeticException – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.

ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.

Types of exceptions

There are two types of exceptions in Java:

1)Checked exceptions

2)Unchecked exceptions

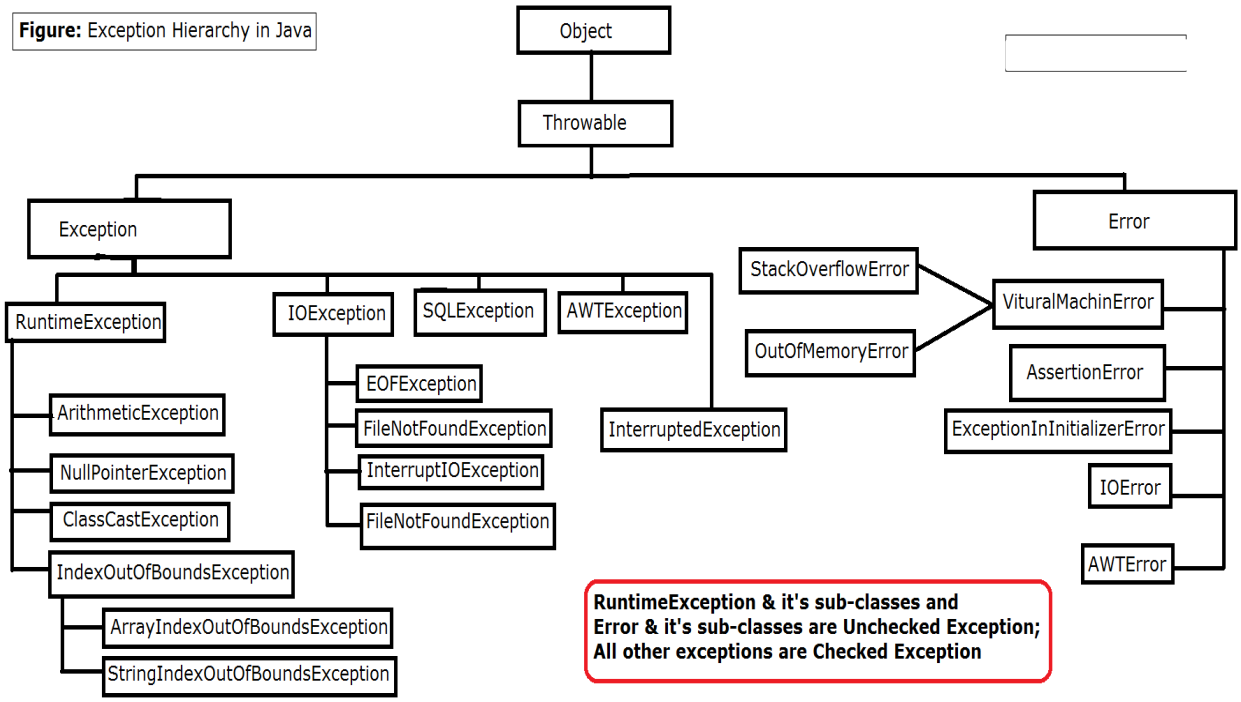
Checked exceptions

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Compiler will never force you to catch such exception or force you to declare it in the method using throws keyword.



Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Try Catch in Java

Try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

Syntax of try block

```
try
{
  //statements that may cause an exception
}
```

While writing a program, if you think that certain statements in a program can throw a exception, enclosed them in try block and handle that exception

Catch block

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try catch in java

```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

Example: try catch block

If an exception occurs in try block then the control of execution is passed to the corresponding catch block. A single try block can have multiple catch blocks associated with it, you should place the catch blocks in such a way that the generic exception handler catch block is at the last(see in the example below).
The generic exception handler can handle all the exceptions but you should place is at the end, if you place it at the before all the catch blocks then it will display the generic message. You always want to give the user a meaningful message for each type of exception rather then a generic message.

```
class Example1
 {
  public static void main(String args[])
```

```
{
    int num1, num2;
    try
{
       num1 = 0;
       num2 = 62 / num1;
       System.out.println(num2);
       System.out.println("Hey I'm at the end of try block");
    }
    catch (ArithmeticException e)
   {

       System.out.println("You should not divide a number by zero");
    }
    catch (Exception e)
   {

       System.out.println("Exception occurred");
    }
    System.out.println("I'm out of try-catch block in Java.");
  }
}
```

Output: You should not divide a number by zero

I'm out of try-catch block in Java.

Multiple catch blocks in Java

The example we seen above is having multiple catch blocks, lets see few rules about multiple catch blocks with the help of examples. To read this in detail, see catching multiple exceptions in java.

1. As I mentioned above, a single try block can have any number of catch blocks.

2. A generic catch block can handle all the exceptions. Whether it is ArrayIndexOutOfBoundsException or ArithmeticException or NullPointerException or any other type of exception, this handles all of them. To see the examples of NullPointerException and ArrayIndexOutOfBoundsException, refer this article: Exception Handling example programs.

```
catch(Exception e)
{
  //This catch block catches all the exceptions
}
```

If you are wondering why we need other catch handlers when we have a generic that can handle all. This is because in generic exception handler you can display a message but you are not sure for which type of exception it may trigger so it will display the same

message for all the exceptions and user may not be able to understand which exception occurred. Thats the reason you should place is at the end of all the specific exception catch blocks

3. If no exception occurs in try block then the catch blocks are completely ignored.
4. Corresponding catch blocks execute for that specific type of exception:
catch(ArithmeticException e) is a catch block that can hanlde ArithmeticException
catch(NullPointerException e) is a catch block that can handle NullPointerException
5. You can also throw exception, which is an advanced topic and I have covered it in separate tutorials: user defined exception, throws keyword, throw vs throws.

Example of Multiple catch blocks

```
class Example2
{
  public static void main(String args[])
{
    try
{
        int a[]=new int[7];
        a[4]=30/0;
        System.out.println("First print statement in try block");
    }
    catch(ArithmeticException e){
      System.out.println("Warning: ArithmeticException");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Warning: ArrayIndexOutOfBoundsException");
    }
    catch(Exception e){
      System.out.println("Warning: Some Other exception");
    }
  System.out.println("Out of try-catch block...");
 }
}
```

Output:

```
Warning: ArithmeticException
Out of try-catch block...
```

In the above example there are multiple catch blocks and these catch blocks executes sequentially when an exception occurs in try block. Which means if you put the last catch block (catch(Exception e)) at the first place, just after try block then in case of any exception this block will execute as it can handle all exceptions. This catch block should be placed at the last to avoid such situations.

Finally block

A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

Syntax of Finally block

```
try
 {
    //Statements that may cause an exception
}
catch {
   //Handling exception
}
finally {
   //Statements to be executed
}
```

A Simple Example of finally block

Here you can see that the exception occurred in try block which has been handled in catch block, after that finally block got executed.

```
class Example
{
   public static void main(String args[])
 {
      try
{
                int num=121/0;
                System.out.println(num);
      }
      catch(ArithmeticException e){
         System.out.println("Number should not be divided by zero");
      }
      /* Finally block will always execute
       * even if there is no exception in try block
       */
      finally
    {
                System.out.println("This is finally block");
      }
      System.out.println("Out of try-catch-finally");
   }
}
```

Output:

```
Number should not be divided by zero
This is finally block
Out of try-catch-finally
```

Few Important points regarding finally block

1. A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always.

2. Finally block is optional, as we have seen in previous tutorials that a try-catch block is sufficient for exception handling, however if you place a finally block then it will always run after the execution of try block.

3. In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block.

4. An exception in the finally block, behaves exactly like any other exception.

5. The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.

try-catch-finally block

- Either a try statement should be associated with a catch block or with finally.
- Since catch performs exception handling and finally performs the cleanup, the best approach is to use both of them.

Syntax:

```
try {
    //statements that may cause an exception
}
catch (…) {
    //error handling code
}
finally {
   //statements to be executed
}
```

Examples of Try catch finally blocks

Example 1: The following example demonstrate the working of finally block when no exception occurs in try block

```
class Example1{

  public static void main(String args[]){
```

```
   try{
      System.out.println("First statement of try block");
      int num=45/3;
      System.out.println(num);
   }
   catch(ArrayIndexOutOfBoundsException e){
      System.out.println("ArrayIndexOutOfBoundsException");
   }
   finally{
      System.out.println("finally block");
   }
   System.out.println("Out of try-catch-finally block");
  }
}
```

Output:

```
First statement of try block
15
finally block
Out of try-catch-finally block
```

throw exception in java

We can define our own set of conditions or rules and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using throw keyword.

Syntax of throw keyword:

```
throw new exception_class("error message");
```

For example:

```
throw new ArithmeticException("dividing a number by 5 is not allowed in this program");
```

Example of throw keyword

```
public class ThrowExample {
   static void checkEligibilty(int stuage, int stuweight){
      if(stuage<12 && stuweight<40) {
```

```
        throw new ArithmeticException("Student is not eligible for registration");
     }
     else {
        System.out.println("Student Entry is Valid!!");
     }
  }

  public static void main(String args[]){
    System.out.println("Welcome to the Registration process!!");
    checkEligibilty(10, 39);
    System.out.println("Have a nice day..");
 }
}
```

Output:

```
Welcome to the Registration process!!Exception in thread "main"
java.lang.ArithmeticException: Student is not eligible for registration
at ThrowExample.checkEligibilty(ThrowExample.java:4)
at ThrowExample.main(ThrowExample.java:13)
```

Throws clause in java

Throws keyword is used for handling checked exceptions . By using throws we can declare multiple exceptions in one go.

Example of throws Keyword

In this example the method myMethod() is throwing two **checked exceptions** so we have declared these exceptions in the method signature using **throws** Keyword. If we do not declare these exceptions then the program will throw a compilation error.

```
import java.io.*;
class ThrowExample {
  void myMethod(int num)throws IOException, ClassNotFoundException{
    if(num==1)
       throw new IOException("IOException Occurred");
    else
       throw new ClassNotFoundException("ClassNotFoundException");
  }
}
```

```
public class Example1{
  public static void main(String args[]){
   try{
     ThrowExample obj=new ThrowExample();
     obj.myMethod(1);
   }catch(Exception ex){
     System.out.println(ex);
    }
  }
}
```

Output:

```
java.io.IOException: IOException Occurred
```

User defined exception in java

In java we can create our own exception class and throw that exception using throw keyword. These exceptions are known as **user-defined** or **custom** exceptions.

Example of User defined exception in Java

```
class MyException extends Exception
{
   String str1;

   MyException(String str2)
  {
              str1=str2;
   }
   public String toString()
  {
              return ("MyException Occurred: "+str1) ;
   }
}

class Example1
{
   public static void main(String args[])
  {
              try
               {
                        System.out.println("Starting of try block");
```

```
                throw new MyException("This is My error Message");
        }
        catch(MyException exp)
        {
                System.out.println("Catch Block") ;
                System.out.println(exp) ;
        }
  }
}
```

Output:

```
Starting of try block
Catch Block
MyException Occurred: This is My error Message
```

Notes:

1. User-defined exception must extend Exception class.
2. The exception is thrown using throw keyword.