INTRODUCTION TO EXCEPTIONS IN JAVA

An exception is a divergence from an application's normal behavior. Exception in java follow a throw and catch mechanism. When something happens that is not expected at a particuler point in the program execution (e.g. not finding a file when trying to read a file), java stops the current execution flow and let us know by throwing an exception. We can handle this exception by catching the exception in the same method and try to recover from it, or catch it and rethrow it, or just leave the handling part to the caller of this method. If no one handle the exception, it is finally thrown to JVM and the program halts execution.

Some exceptions can be avoided with proper coding, but some are not within our control, and hence need to be handled. In java, handling of certain type of exceptional conditions are mandatory, some others are optional and few are not recommended. We will see all these cases here. For that we should understand the types of exceptional conditions and the exception hierarchy in java.

Some advantages of Java's exception handling are:

- Error handling code is separated from the normal program function and hence it improves the program structure.
- The programmer can choose where to handle exceptions or whether to handle at all. We can handle this exception by catching the exception in the same method and try to recover from it, or catch it and rethrow it, or just leave the handling part to the caller of this method.
- You can create youe own exception specific to your application.

Exception Hierarchy

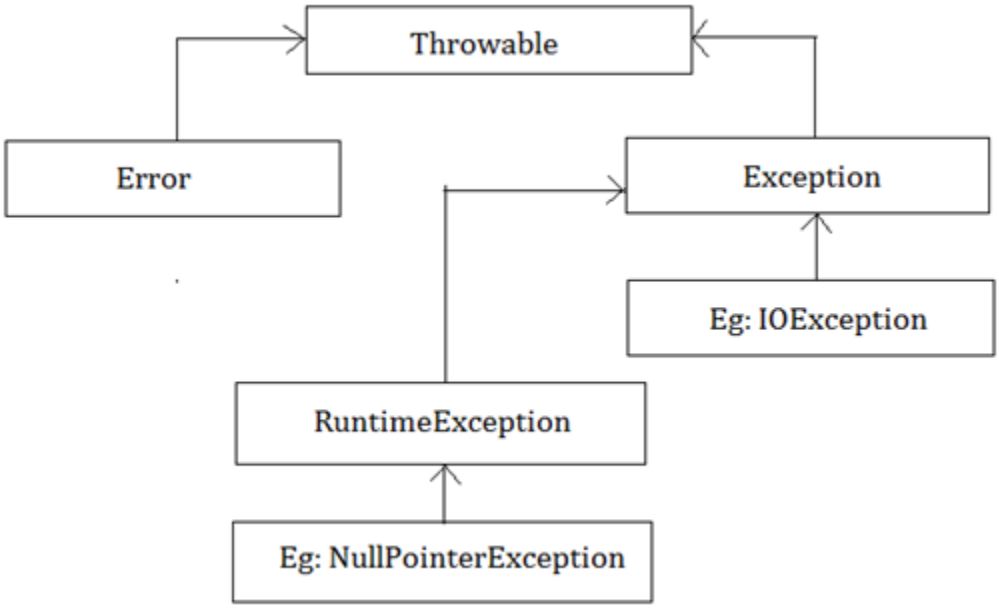
The parent class for all exception related classes is the java.lang.Throwable class.

Throwable has two children: Exception and Error.

Exception and all its children are usually recoverrable and hence we can handle them and try to recover from them.
Exception and all its subclasses except RuntimeException are called **checked excpetions**. Java mandates that we need to handle checked exceptions.
RuntimeException and its subclasses are called **unchecked exceptions or runtime exceptions**. Handling of Runtime exceptions or unchecked exceptions are optional, and java doesn't mandate it.
Usually there is nothing much we can do from within our program to recover from an **error** and hence it is recommended not to try to recover from error within the program. For example, consider the OutOfMemory error. You can't do much from the program when system's memory is run out; instead you

should restart the system or just application and try to optimize the program and system resources to avoid the error from occuring again.
Any Throwable (Exception, RuntimeException, Error or any of their subclasses) can actually be handled the same way; however, handling checked exception is mandated, handling runtime exception is optional and handling error is not recommended.

Exception hiararchy can be summarized as below.



Checked Exception v/s Runtime Exception

Exception and all its subclasses except RuntimeException are called as checked exceptions. RuntimeException and its subclasses are called unchecked exceptions. Java mandates that we need to handle checked exceptions. Handling of Runtime exceptions or unchecked exceptions are optional.

A **checked exception** is an exception for which the developer doesn't have any control over the occurrences with the possibility of recovery. IOException is an example for a checked exception. IOException usually occurs when there is some problem with an IO device and the developer doesn't have any control on that; but can try workarounds, like retries after waiting for some time. Therefore, the developer must handle it, either explicitly using try-catch block or say that it is being handled elsewhere through the use of throws clause in that method signature. Else program won't compile.
A **runtime exception** is an exception that mostly represent a coding mistake. It is called unchecked exception because handling is optional. Instead of handling it, you should actually try to avoid it through

better programming. NullPointerException is an example for an unchecked exception. A NullPointerException can be avoided by having proper null checks on variables before accessing them.

Handling exceptions

We can handle exceptions using the try-catch-finally construct or explicitly tell as being handled elsewhere through the use of throws clause in that method signature.

The try-catch-finally block will look as below:

```
try{

//Some code that can throw IOException

}

catch(Exception ex)

{

//do some workaround.

//you can use the  IOException object ex for getting more details on the exception.

}

finally{

//always executed. So can do some cleanup activities.

}
```

We enclose the code that may throw an exception in the try block.

The remaining statments within the try block are not executed after an exception is thrown for a statement.

The try block is usually followed by one or more catch blocks.

Each try block specifies an exception and if that exception or any of its subclasses occur, that try block will be executed.

If none of the exceptions specified within the catch blocks match, the exeception is thrown to the caller of the method.

The catch block is usually followed by a finally block. The finally block will be executed always irrespective of whether exception is thrown or not. So finally block is the best place to release resources like closing connections or files.

A try block should always be followed by a catch block or finally block or both as above. Few valid cases are:

- A try block with only finally block.
- A try block with only catch block.
- A try block with catch block and finally block.

An invalid case is:

- A try block without catch or finally

Order should be always try followed by catch followed by finally. For instance, you cannot have catch after finally.

If we are having multiple catch blocks, a parent exception catch block cannot come before a child exception catch block. This is because, if a parent exception catch block appears before a child exception catch block, parent exception catch block will catch the child exception as well and child exception catch block will never get executed.

Use of throws

We can handle exceptions using the try-catch-finally construct or explicitly tell as being handled elsewhere through the use of throws clause in that method signature.

We can declare a throws clause as:

```
public void myMethod() throws IOException
{

//method body with some code that can throw IOException

}
```

When you declare the throws clause for a checked exception like this, you don't have to handle that exception within the method (here myMethod), but any other code that will call this method need to either handle the exception or again declare a throws clause in its signature.

Throws and throw

The **throws** keyword is used to declare that a mathod might throw an exception.
The **throw** keyword is used to throw or rethrow an exception.

```
try{
throw new IOException();
}
catch(IOException ex)
{
//re-throwing as a custom exception
throw new MyCustomException(ex);
}
```

You can also write your own custom exceptions.

Inheritance and throws keyword

If a superclass method declares a throws clause for a checked exception, the overriding subclass method can't replace it with a broader one. Child class however can replace it with a subclass of the parent exception or even remove the throws clause for that exception completely. This is because, when a subclass is used in the context of super class by assigning a subclass object to superclass, the caller expects only those exceptions that are declared on the superclass. So we can throw a subclass of the exception or even decide not to throw any, which will not create any problem for the caller. However if we add more exceptions, parent will not be expecting it and hence won't be prepared to handle; hence compiler won't allow that. This restriction is not there for unchecked exceptions as their handling is optional.

Java 7 Multi-Catch Exceptions

With java 7, you can catch multiple exceptions using a single catch block as:

```
try {
  File file = new File("filename.txt");
  Scanner sc = new Scanner(file);
  throw new SQLException();
}
```

```
catch (FileNotFoundException | SQLException e) {
  System.out.println("FileNotFoundException or SQLException called!!!");

}
```

Java 7 Try-with-Resources

Finally block is usually used to close and release resources. Now with the new java 7 syntax, you can declare your resources withing the try itself and java automatically closes those resources (if they are not already closed) after the execution of the try block.

```
try (BufferedReader reader =
    new BufferedReader(

        new FileReader(

    "filename.txt")))
    {

      // try block contents

} catch (IOException e1) {

  // TODO Auto-generated catch block

  e1.printStackTrace();

}
```

The resource type specified within the try should implement java.lang.AutoCloseable.

Source : http://javajee.com/introduction-to-exceptions-in-java