Types of Exception

As per sun micro systems standards The Exceptions are divided into three types

1) Checked Exception

2) Unchecked Exception

3) Error

Checked Exception:-

The Exceptions which are checked by the compiler at compilation time for the proper execution of the program at runtime is called CheckedExceptions. Ex:- IOException,SQLException etc…….

Exception	Description
ClassNotFoundException	If the loaded class is not available
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	If the requested method is not available.
…….	

Unchecked Exception:- The exceptions which are not checked by the compiler at compilation time is called uncheckedException . These checking down at run time only. Ex:- ArithmeticException, NullPointerException, etc…….

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.(out of range)
InputMismatchException	If we are giving input is not matched for storing input.
ClassCastException	If the conversion is Invalid.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalThreadStateException	Requested operation not compatible with current thread state
IndexOutOfBoundsException	Some type of index is out-of-bounds
NegativeArraySizeException	Array created with a negative size
NullPointerException	Invalid use of a null reference

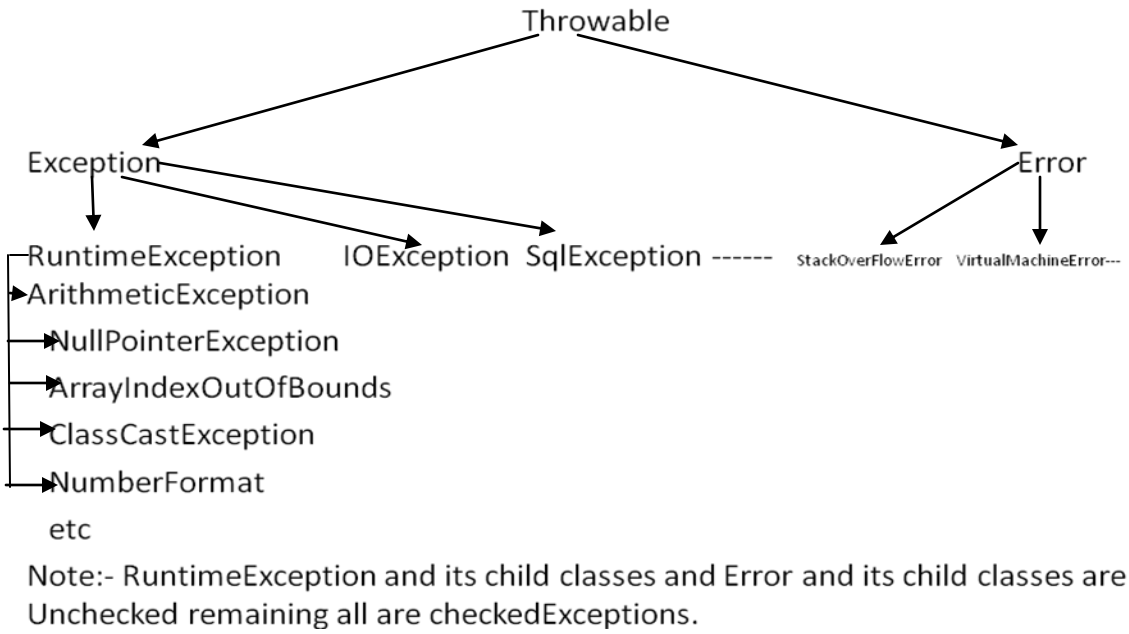
NumberFormatException Invalid conversion of a string to a numeric format
StringIndexOutOfBoundsException Attempt to index outside the bounds of a string

…….

Error:- Errors are caused by lack of system resources ,these are non recoverable.
Ex:- StackOverFlowError,OutOfMemoryError,AssertionError etc……

Note:- The Exception whether it is checked or unchecked the exceptions are occurred at runtime.

Exception Hierachy :



Note:- RuntimeException and its child classes and Error and its child classes are Unchecked remaining all are checkedExceptions.

Example of Checked Exceptions

```
import java.io.*;
class  UnHandledCheckedExceptionExample
 {
 public static void main(String[] args)
 {
    FileWriter f = new FileWriter("abc.txt ");
```

We will get compile time error :

unreported exception IOException; must be caught or declared to be thrown

Here compiler is expecting either use try….catch block or throws keyword in our code.

```
   f.write(102);
   f.write('z');
   f.close();
    }
}
```

In this program we are using resources, so may be that resource not avaible, so compiler says that handle related exception or pass this exception to the

If we rewrite code as below, we will not get any compilation error.

```
import java.io.*;
class CheckedExceptionExample
 {
 public static void main(String[] args)
 {
  try {
   FileWriter f = new FileWriter(" ");
   f.write(102);
   f.write('z');
   f.close();
     }
  catch(IOException e)
       { System.out.println("Checked Exception " + e);}
  }
 }
```

```
import java.io.*;
class CheckedExceptionExample  throws IOException
 {
 public static void main(String[] args)
 {
   FileWriter f = new FileWriter(" ");
   f.write(102);
   f.write('z');
   f.close();
  }
 }
```

throws keyword:

in our program if there is any chance of raising checked exception compulsory we should handle either by try catch or by throws keyword otherwise the code won't compile. Ex:

```
class RaisingCheckedException
{
public static void main(String[]args)
  {
   Thread.sleep(5000);
 }
}
```

We will get compile time error

Unreported exception java.lang.InterruptedException; must be caught or declared to be thrown.

We can handle this compile time error by using the following 2 ways.

a. by using try…….catch block

```
class RaisingCheckedException
{
public static void main(String[]args)
  {
  try{
   Thread.sleep(5000);
        }
  catch(InterruptedException ie)
     {
      System.out.println(ie);
    }
 }
}
```

b. by using throws keyword

```
class RaisingCheckedException
{
public static void main(String[]args)  throws InterrupedException
  {
   Thread.sleep(5000);
 }
}
```

We can use throws keyword to delicate the responsibility of exception handling to the caller method. Then caller method is responsible to handle that exception. In this example caller is JVM so if any exveption occurs at run time, JVM's default exception handler will be responsible to handle the exception.

- Hence themain objective of "throws" keyword is to delicate the responsibility of exception handling to the caller method.
- "throws" keywordrequired only for checked exceptions. Usage of throws for unchecked exception have no impact on the compiler or no meaning to use unchecked exception with throws keyword .
- "throws" keyword required only to convenience complier. Usage of throws keyword doesn't prevent abnormal termination of program
- We can use throws keyword only for Throwable types otherwise we will get compile time error saying incompatible types.

```
class Test
{
  public static void main(String[]args) throws  Test
   {    }
}
```

Output: Compile time error

Test .java:3: incompatible types found: Test3
required: java.lang.Throwable

public static void main(String[]args) throws

throw statement:
Some time we can create exception object explicitly and we can hand over to the JVM manually by using throw keyword.

class ThrowDemo
{
public static void main (String[] args)
{ System.out.println(10/0); }
}
• In this case creation of ArithmeticException object and hand over to the jvm will be performed automatically by the main()method.
	class ThrowDemo
	{
	public static void main (String[] args)
	{ throw new ArithmeticException("Divided by zero")
	}
	}
	• In this case we are creating exception object explicitly and handover to JVM

Customized Exceptions(UserdefinedExceptions):

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. Sometimes to meet our programming requirements we want our own exception . Such type of exceptions are called customized exceptions(user defined exceptions).

We can create our own exception in the following ways –

1.By creating a class which extends the superclass of all the exception classes – Exception class, or
2. By creating a class which extends any of the subclasses of the Exception class, such as ArrayStoreException, IOException, ParseException, NullPointerExceptionetc.

After creating our own exception class, we will be able to throw and catch our exception just as we throw and catch predefined checked/unchecked exceptions.

```
public class MinBalanceException extends Exception {

 public MinBalanceException()
 {
    System.out.println("Balance is low");
 }
}

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

      try{
```

```
    int acc[] = {100,101,102,103,104,105};      // input can be got from runtime too
    double balance[]={900,2000,1500,1560,1765.50};
    System.out.println("Account No\t"+"Balance\t");
    for (int i=0;i<5;i++)
    {
       System.out.println(acc[i]+"\t"+balance[i]+"\t");
       if(balance[i]<1000)
       {
          throw new MinBalanceException();  //throwing user defined exception
        }
    }
  }
    catch(MinBalanceException e)
         {
          System.out.println("Exception caught");
         }
  }
}
```

```
class TooYoungException extends RuntimeException
 {
 TooYoungException(Strings) { super(s); }
}
 class TooOldException extends RuntimeException
{
 TooOldException(Strings) {  super(s);  }
 }

class CustomizedExceptionDemo  {
  public static void main( String[] args ) {
          int age=Integer.parseInt(args[0]);
            if(age<18)  {
                 throw new TooYoungException("please wait some more time....You will get best match");
              }
          else if(age>60)  {
            throw  new TooOldException("your age already crossed....no chance of getting married");
              }
               else {
                    System.out.println("you will get match details soon  bye-byel");
                   }
           }//main
  }//class
```

DO NOT AVOID OR IGNORE AN EXCEPTION:

- Adding an empty catch block after a try block and leaving the system to handle is not a good practice of exception handling.
- API can identify checked exceptions and intimate the same so that programmer can catch them as needed. But unchecked exceptions need to be handled by analyzing the logic of code present in try block or by adding generic exception catch block
- If the programmer is not able to do anything with a checked exception he could throw a runtime exception instead of adding an empty catch block or just not handling at all.
- Unless it is very much important to handle an runtime exception, do not handle it. Not every exception has to be specified and handled in the catch block.
- Either log the exceptions or throw them. Do not perform both. This may confuse the person who is analyzing all the exceptions.
- Do not catch top level exceptions in general until a lower level exception cannot be found suitable for the try block logic.

Multi – catch block in java :

Prior to Java 7, we had to catch only one exception type in each catch block. So whenever we needed to handle more than one specific exception, but take same action for all exceptions, then we had to have more than one catch block containing the same code.
In the following code, we have to handle two different exceptions but take same action for both. So we needed to have two different catch blocks as of Java 6.0.

```
import java.util.Scanner;
public class TwoDifferentCatchBlockForSameAction
{
   public static void main(String args[])
   {
      Scanner scn = new Scanner(System.in);
      try
      {
         int n = Integer.parseInt(scn.nextLine());
         if (99%n == 0)
            System.out.println(n + " is a factor of 99");
      }
      catch (ArithmeticException ex)
      {
         System.out.println("Arithmetic " + ex);
      }
      catch (NumberFormatException ex)
      {
         System.out.println("Number Format Exception " + ex);
      }
   } }
```

From Java 7.0, it is possible for a single catch block to catch multiple exceptions by separating each with | (pipe symbol) in catch block.

Above code be re-written using Multi-Catch block as

```
import java.util.Scanner;
public class MultiCatchDemo
{
   public static void main(String args[])
   {
      Scanner scn = new Scanner(System.in);
      try
      {
         int n = Integer.parseInt(scn.nextLine());
         if (99%n == 0)
            System.out.println(n + " is a factor of 99");
      }
      catch (NumberFormatException | ArithmeticException ex)
      {
         System.out.println("Exception encountered " + ex);
      }
   }
}
```

Note : Single catch block can handle more than one type of exception. However, the base (or ancestor) class and subclass (or descendant) exceptions can not be caught in one statement.

```
// Not Valid as Exception is an ancestor of   NumberFormatException

catch(NumberFormatException | Exception ex)
```

try with Resource :

The try-with-resources Statement :

The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. **Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.**

The following example reads the first line from a file. It uses an instance of BufferedReader to read data from the file. BufferedReader is a resource that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {
   try (BufferedReader br =   new BufferedReader(new FileReader(path) ) )
    {
      return br.readLine();
   }
}
```

In this example, the resource declared in the try-with-resources statement is a BufferedReader. The declaration statement appears within parentheses immediately after the try keyword. The class BufferedReader, in Java SE 7 and later, implements the interface java.lang.AutoCloseable. Because the BufferedReader instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method BufferedReader.readLine throwing an IOException).

Prior to Java SE 7, you can use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. The following example uses a finally block instead of a try-with-resources statement:

```
static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException
    {
   BufferedReader br = new BufferedReader(new FileReader(path));
   try {
      return br.readLine();
   }
 finally {
      if (br != null)  br.close();
   }
}
```

Note :

1. We can declare multiple resources but these resources should be separeted with semicolon.

try(R1;R2;R3)
{ }
R1, R2,R3 must be autocloseable resources.

2. All resource variables declared in try with resource block are implicitly final and hence within the try block, we can't perform reassignment, otherwise we will get compile time error.
3. Until 1.6 version try should be associated with either catch or finally. But from 1.7 version onwards we can take only try with resource, without catch or finally.

Examples :

```
import java.io.*;
class TryWithResource1
{
public static void main(String[] args)
{
 try (  PrintWriter fw = new PrintWriter("w3.txt")  )
  {
   fw.write('x');
   fw.print(100);
   fw.write(100);
   fw.write("Heeloooo");
   fw.write(new char[] {'A','B'});
  }
 catch (IOException e) {   }
}
}

class TryWithResource2
{
public static void main(String[] args)
{
 try ( FileWriter fw = new FileWriter("w5.txt");BufferedWriter bw = new BufferedWriter(fw) )
  {
  bw.write('x');
  bw.write(100);
  }
 catch (IOException e) {   }
}
}

class TryWithResource3
{
public static void main(String[] args)
{
```

```
 try ( BufferedReader br = new BufferedReader(new FileReader("w3.txt")))
  {   int c=br.read();
     while(c!=-1)
      {
        System.out.println((char)c);
        c=br.read();
      }
   }
 catch (IOException e) {    }
}
}
// Try with resource without catch and finally
class TryWithResource4
{
public static void main(String[] args)  throws  IOException
{
 try ( BufferedReader br = new BufferedReader(new FileReader("TryWithResource.java")))
  {
    String c=br.readLine();

//br= new BufferedReader(new FileReader("TryWithResource.java"));
// reassignment of reference variable of resource , we will get compilation error
     while(c!=null)
      {
        System.out.println(c);
        c=br.readLine();
      }
   } //try

}// main
} //class
```