10. EXCEPTION HANDLING

1. Exception and error:

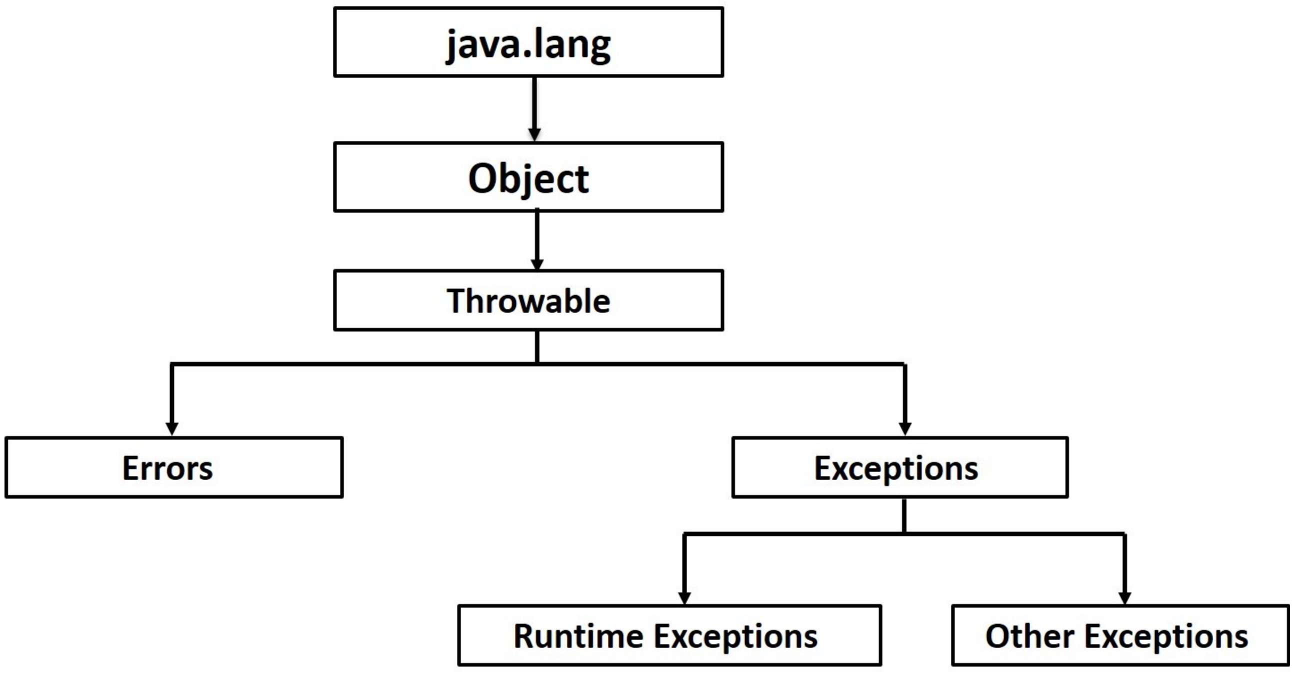
An exception (or exceptional event) is a problem that arises during the **execution** of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore these exceptions are to be handled.

An exception can occur for many different reasons, below given are some scenarios where exception occurs.

- 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.

An error occurs at compile time. It is not more dangerous than exception.

2. Exceptionn class Hierarchy:



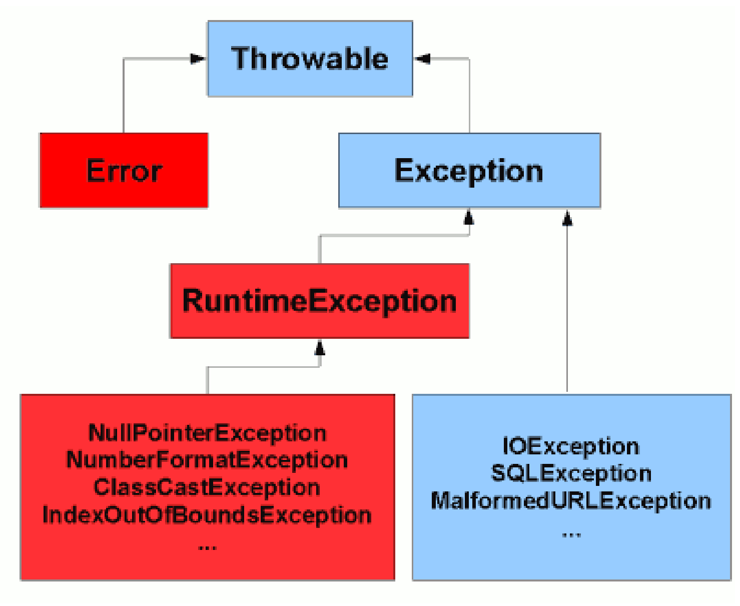
All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped from 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.

10. EXCEPTION HANDLING

Note: The Error is a class in Exception hierarchy. It is not related to compile time error.

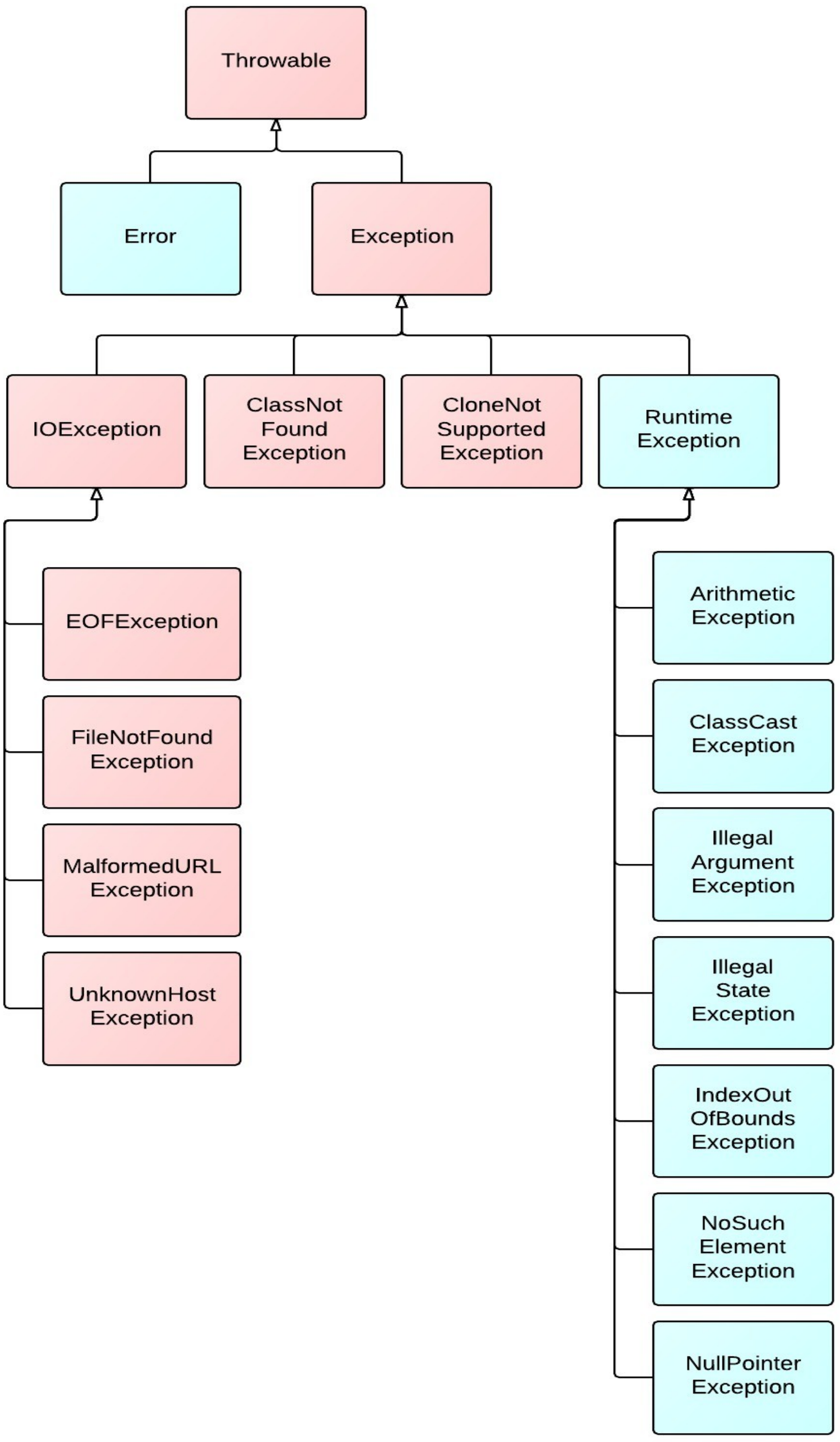
The Exception class has a subclass: **RuntimeException Class.**



3. Types of Exceptions:

1. Checked exceptions: A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the Programmer should take care of (handle) these exceptions.
For example, if you use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then an FileNotFoundException occurs, and compiler prompts the programmer to handle the exception.
Since the methods read() and close() of FileReader class throws IOException, you can observe that compiler notifies to handle IOException, along with FileNotFoundException.

2. Unchecked exceptions: An Unchecked exception is an exception that occurs at the time of execution but not checked at compile time, these are also called as Runtime Exceptions. Runtime exceptions are ignored at the time of compilation. For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an ArrayIndexOutOfBoundsExceptionexception occurs.



4. Inbuilt Methods:

Following is the list of important medthods 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.

5. Exception handling mechanism:

There are several keywords that are used in exception handling. These keywords can be used to create block and clause.

- try : block
- catch : block
- finally : block
- throw : clause / statement
- throws : clause / statement

5.1 Try block:

The code that you want to monitor for the exceptions write that code into try block. 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(ExceptionType e1) {
        //Exception handling code
}
```

5.2 Catch block:

10. EXCEPTION HANDLING

A try block is followed by a catch block which handles the exceptions. It contains the exception handling code.

Syntax:

```
catch(ExceptionType e1) {
        //Exception handling code
}
```

If exception occurs in try block, the exception object is created and that is thrown to matching catch block. The catch block will handle the exception. Here either we can print exception object or can call the printStackTrace method and display the information of the exception that has occured.

Exception object contains following information:
type of exception, class of exception.

But if we call method **printStackTrace()** from catch block then it will contain following information:
at which line it occured, in which method it occured, who is the caller of that method, from which line that method get called, in which class it occured, in which file it occured and so on.....

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.

```
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("Code after catch  block");
        }
}
```

Output:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Code after catch block
```

try with 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) { //Child ExceptionType
                      //Handling code
 }
catch(ExceptionType2 e2){ //Child ExceptionType
                      //Handling code
  }
 catch(ExceptionType3 e3){ // Parent Exception Type
                      //Handling code
  }
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data 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.
If there is no matching catch, exception will ne handled by default exception handler.

Example:
Here is code segment showing how to use multiple try/catch statements.

```
    try
        {
          file = new FileInputStream(fileName);
          x = (byte) file.read();
        }
        catch(FileNotFoundException f)
        {
          f.printStackTrace();
          return -1;
        }
        catch(IOException i)
        {
          i.printStackTrace();
          return -1;
        }
```

The following code will generate compile time error

```
 try
        {
          file = new FileInputStream(fileName);
          x = (byte) file.read();
        }
```

```
catch(Exception e)
{
  e.printStackTrace();
   return -1;
}
catch(IOException i)    // Not valid!
{
   i.printStackTrace();
   return -1;
}catch(FileNotFoundException f)   // Not valid!
{
   f.printStackTrace();
   return -1;
}
```

Note: We can not write the catch taking super class reference variable of exception hierarchy before the that of subclass.
Compiler Error will be : exception already caught.
Thus While writing try with multiple catches, catch with subclass should be before catch with superclass.

5.3 throw clause :

When we want to explicitly throw any exception then we use throw clause.
We can throw an exception, either using new instance or using exception object by using the **throw** keyword.

Syntax: **throw new** TypeOfException(parameters);

Thuse the method can explicitly throw an exception using throw keyword.

5.4 throws clause:

If a method is not able to handle a particular exception, the method must declare it using the **throws** keyword with its definition. It will throw that exception out of itself. Then caller has the responsiblity to handle that exception. The throws keyword appears at the end of a method's signature. A method can throw multiple exceptions using throws clause.

Syntax: **throws** Type1Exception,Type2Exception....

Try to understand the different in throws and throw keywords.
The following method shows use of both throw and throws

```
public class className
         {
           public void deposit(double amount) throws RemoteException
```

```
        {
          // Method implementation
            if(remoteObject==null)
             throw new RemoteException("null remote object");
        }
        //Remainder of class definition
      }
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

```
public class className {
public void withdraw(double amount) throws RemoteException, InsufficientFundsException
        {
                // Method implementation
         }
    //Remainder of class definition
 }
```

Difference between thorw and thorws:

throw	**throws**
To manually throw the exception	To throw the exception out of the method towards caller when the callee method is not able to handle that exception

Example:
public void show() {
  throw new NullPointerException();
}
public void show() {
  throw new NullPointerException("pointing to null reference");
}
	Example:
	public void show()**throws** IOException, NumberFormatException
	{
	  // body of the method
	}

It is followed by instance of class	It is followed by name of class
It is declared within the method body	It is declared with the method definition
It can throw only one excetion at a time	It can declare muliple exception classes at a time
The exception that is thrown by throw can be handled within that method	The exception declared in throws can not be handled by that same method thats why it is propagated towards caller.

5.5 The finally block :

10. EXCEPTION HANDLING

The finally keyword is used to create a block of code that follows a try block or try catch block. A finally block of code always executes, whether or not an exception has occurred.
Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code. For e.g. u can write the file closing or database closing code in finally block if the file is opened in try block or database connections are opened in try block.

Syntax:

```
finally
{
        // important code, cleanuptype statements
}
```

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

```
try
{
         //Protected code
}catch(ExceptionType1 e1)
{
        //handling code
}
catch(ExceptionType2 e2)
{
         //handling code
}
catch(ExceptionType3 e3)
{
         //handling code
}
finally
{
         //cleanuptype statements
}
```

Note: The try block must have atleast one catch block or finally block

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);
            }
            finally{
              System.out.println("The finally statement is executed");
            }
          }
        }
```

Output:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
The finally statement is executed
```

Execution Sequence :

1. try{}finally{} : first try block will get executed, then if exception occurs then, finally will get executed and later exception will be caught by default exception handler.

2. try{}catch{}finally{} : first try block will get executed, then if exception occurs and if there is matching catch, exception will be handled in that catch, then finally will get executed.

6. User Defined Exceptions:

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyException extends Exception{
}
```

You just need to extend the Exception class to create your own Exception class. These are considered to be checked exceptions. The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

Example:

```
import java.util.Scanner;
class CheckingAccount
{
        private double balance;
        private int number;
```

```
            public double getBalance() {
                        return balance;
            }

            public void setBalance(double balance) {
                        this.balance = balance;
      }
            public int getNumber() {
                        return number;
            }
            public void setNumber(int number) {
                        this.number = number;
            }
            public CheckingAccount(int number) {
            this.number = number;
            }
            public void deposit(double amount) {
                        balance += amount;
            }
            public void withdraw(double amount)throws InsufficientFundsException {
               if(amount <= balance)
               {
                        balance -= amount;
            }
         else
         {
            double needs = amount-balance;
            throw new InsufficientFundsException(needs);
         }
      }
   }

   class InsufficientFundsException extends Exception
   {
             double needs;
             InsufficientFundsException(double needs) {
                         this.needs=needs;
             }
            @Override
            public String toString() {
                        return "InsufficientFundsException [needs=" + needs + "]";
            }
   }

   public class UserDefinedException {

            public static void main(String[] args)  {
                        Scanner sc=new Scanner(System.in);
                        CheckingAccount ca=new CheckingAccount(4070856);

                        try {
                        ca.setBalance(50000);
                        System.out.println("The initial balance is: "+ca.getBalance());
                        System.out.println("Enter the amount to deposit: ");
                        double damount=sc.nextDouble();
                        ca.deposit(damount);
                        System.out.println("The balance is: "+ca.getBalance());
                        System.out.println("Enter the amount to withdraw: ");
                        double wamount=sc.nextDouble();
                        ca.withdraw(wamount);

                        } catch (InsufficientFundsException e) {
                                System.out.println("Balance is insufficient");
                                e.printStackTrace();
                        }
                        catch(Exception e) {
                                e.printStackTrace();
                        }
                        System.out.println("The code after catch continued....");
            }
   }
```

Output:

```
The initial balance is: 50000.0
Enter the amount to deposit:
3000
The balance is: 53000.0
Enter the amount to withdraw:
54000
Balance is insufficient

InsufficientFundsException [needs=1000.0]
```

```
        at exceptionhandling.CheckingAccount.withdraw(UserDefinedException.java:44)
        at exceptionhandling.UserDefinedException.main(UserDefinedException.java:84)
The code after catch continued....
```

Thus InsufficientFundsException get generated when user enters the amount to withdraw which is more than available balance and it is get handled at caller side.

7. Chained Exceptions:

The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an ArithmeticException because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly.

8. Exception Propagation:

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation. It can be achieved via throws clause.

ASSIGNMENTS

1. Create the account of user. Handle the exceptions in following situations
   1. User is trying to withdraw but balance is not sufficient.
   2. User trying to transfer the funds but destination account does not exists
   Throw all the exceptions towards caller.

2. Generate mini calculator. Handle the exceptions in following situations
   1. Mini calculator can not handle the values above 99999
   2. Mini calculator can not handle negative values
   Throw all the exceptions towards caller.

3. Which type of error is in following code?

```
try
{
int result=5.7/0;
}
catch(Exception e) {
    e.printStackTrace();
}
catch(ArithmeticException e) {
    e.printStackTrace();
}
```

4. What is the output of following code?

```
try
{
    try
    {
        int result=5.6/0;
    }
    catch(Exception e)
    {
        throw e;
    }
    finally
    {
        System.out.println("Inside Inner Finally");
    }
}
catch(Exception e)
{
    System.out.println("In outer catch");
    System.out.println(e);
}
finally
{
    System.out.println("Inside Outer Finally");
}
```

5. What is the output of following code?

```
private static void method2() {
        Connection connection = new Connection();
        connection.open();
        try {
                        // LOGIC
                 String str = null;
                str.toString();
                  return;
        } catch (Exception e) {
                        // NOT PRINTING EXCEPTION TRACE - BAD PRACTICE
                System.out.println("Exception Handled - Method 2");
               return;
        } finally {
                        System.out.println("In finally");
                        connection.close();
        }}
```