Exceptions

- It is very common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing the program to produce unexpected results. It is therefore important to detect and manage properly all the possible error conditions in the program.
- Errors may broadly be classified into two categories: compile-time errors and run-time errors.
- **Compile-time errors:** Java compiler detects all syntactical errors during the compilation phase of a program. The compiler creates a .class file only when there is no such error in the program.
- **Run-time errors:** The logical errors in a program, usually not reported by the compiler, may produce wrong results during the program execution. Examples are- division by zero, out of bound subscript in an array etc.
- Exceptions: An exception is a condition that is caused by a run-time error in the program and when Java interpreter encounters such an error, it creates an exception object and throws it. Now if this exception object is not caught and handled properly, the interpreter displays an error message and terminates the program.
- If we want the program to continue with the execution of the remaining code then we need to catch the exception object thrown by the error condition and display an appropriate message for taking corrective actions. This is known as exception handling.
- The basic concept of exception handling are throwing an exception and catching it. This is done as follows-

```
try
{
    //Statement that causes an exception must be written here
   //exception object will be created and thrown from here
}
catch(Exception type)
{
  //statement that handles the exception are written here
}
```

- Java uses a keyword try to preface a block of code that is likely to cause an error condition and throw an error. A catch block defined by the keyword catch catches the exception thrown by the try block and handles it properly. The catch blocks are added immediately after try block.
- When any of the statement within the try block generates an exception, the remaining statements are by passed and the execution jumps to the catch block.
- We may have several catch statements in a program and when a match is found with the catch parameter (every catch statement is passed a single parameter) and the type of exception object, the exception is caught and the statement in the catch block is executed. Otherwise the exception is not caught and the default exception handler will cause the execution to terminate.
- Note: every try statement should be followed by at least one catch statement otherwise compilation error will occur.

➢ Consider the following example-

```
class  Error_example
{
  public static void main(String args[])
  {
    int x, y=0;
    try
    {
     x=3/0;    //this statement generates run time exception
    }
    catch(ArithmeticException err)
    {
      System.out.println("Division by zero=>Run time error!");
    }
    System.out.println(y);
  }
}
```

Output: Division by zero=>Run time error!
0

➢ Common Java Exceptions-

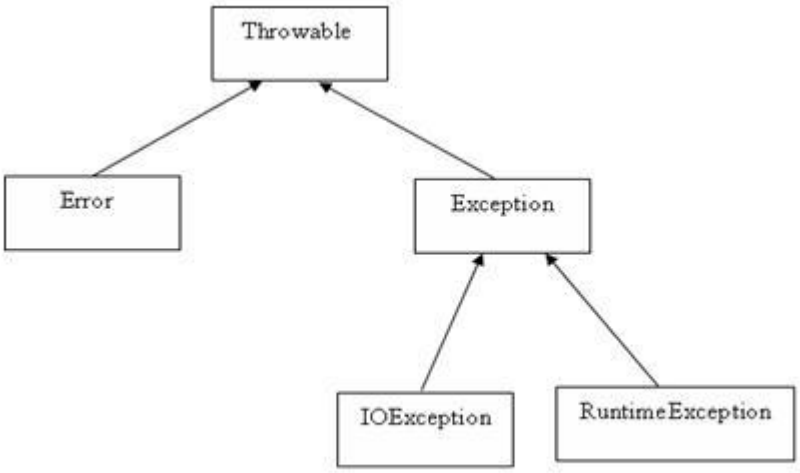
Exception type	Reason
ArithmeticException	mathematical errors(eg.division by zero).
ArrayIndexOutOfBoundaException	array index beyond the range 0 to n-1 for an array consisiting of n element.
ArrayStoreException	wrong type of data being stored in the array
IOException	general I/O failures(eg. reading a corrupted file).
NullPointerException	null object reference
NumberFormatException	string to number conversion failure
OutOfMemoryException	shortage of memory for a new object allocation
SecurityException	applets performing an action not allowed by the browser

Exception 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 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.



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
}
```

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.

Example:

```
try
{
   file = new FileInputStream(fileName);
   x = (byte) file.read();
}catch(IOException i)
{
   i.printStackTrace();
   return -1;
}catch(FileNotFoundException f) //Not valid!
{
   f.printStackTrace();
   return -1;
}
```

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.

The following method declares that it throws a RemoteException:

```
import java.io.*;
public class className
{
   public void deposit(double amount) throws RemoteException
   {
      // Method implementation
      throw new RemoteException();
   }
   //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:

```
import java.io.*;
public class className
{
   public void withdraw(double amount) throws RemoteException,
                               InsufficientFundsException
   {
       // Method implementation
   }
   //Remainder of class definition
}
```

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.

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.

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

This would produce following result:

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

Note the following:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses when ever a try/catch block is present.
- 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.

Declaring you own Exception:

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.

Example:

```
// File Name InsufficientFundsException.java
```

```
import java.io.*;

public class InsufficientFundsException extends Exception
{
   private double amount;
   public InsufficientFundsException(double amount)
   {
      this.amount = amount;
   }
   public double getAmount()
   {
      return amount;
   }
}
```

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

```
// File Name CheckingAccount.java
import java.io.*;

public class CheckingAccount
{
   private double balance;
   private int 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);
      }
   }
   public double getBalance()
   {
      return balance;
   }
   public int getNumber()
   {
      return number;
   }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java
public class BankDemo
{
   public static void main(String [] args)
   {
      CheckingAccount c = new CheckingAccount(101);
      System.out.println("Depositing $500...");
      c.deposit(500.00);
      try
      {
         System.out.println("\nWithdrawing $100...");
         c.withdraw(100.00);
         System.out.println("\nWithdrawing $600...");
         c.withdraw(600.00);
      }catch(InsufficientFundsException e)
      {
         System.out.println("Sorry, but you are short $"
                                  + e.getAmount());
         e.printStackTrace();
      }
    }
}
```

Compile all the above three files and run BankDemo, this would produce following result:

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```