**Exception Handling**

What is exception?

Exception is an abnormal condition.

An exception 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.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

**What is Exception Handling?**

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

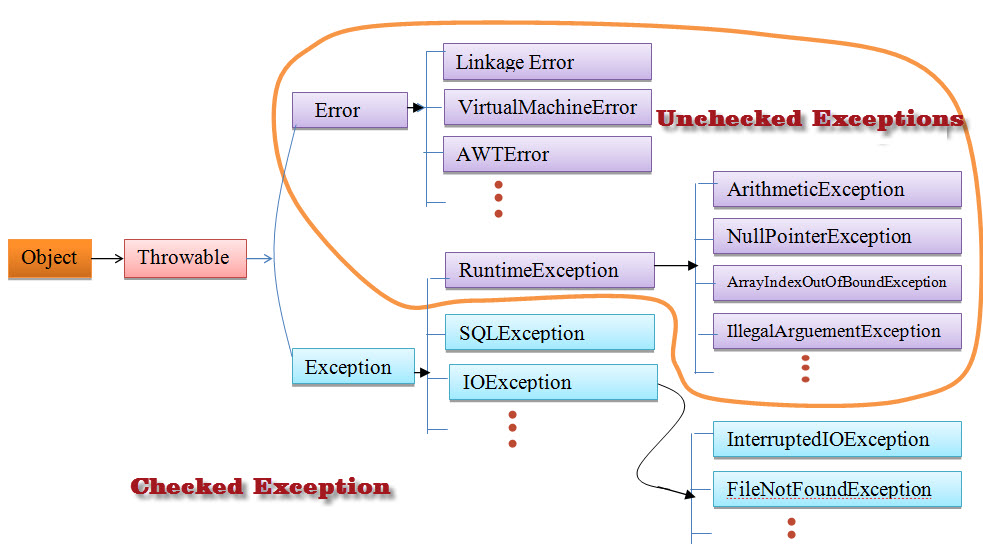
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;

suppose there are 7 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6,7 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling.

**Hierarchy of Java Exception classes**

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below



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.

**Types of Exception**

1. Checked
2. Unchecked
3. Error

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

**Exceptions Methods:** Following is the list of important methods available in the Throwable class.

**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.

**Catching Exceptions:** A method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

**public** **class** JavaExceptionExample{

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

**try**{

      //code that may raise exception

**int** data=100/0;

   }**catch**(ArithmeticException e)

{

System.out.println(e);

}    //rest code of the program

   System.out.println("rest of the code...");

  }  }

**Java catch block**

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

**Internal Working of Java try-catch block**



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

**Java Multi-catch block**

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.



**public** **class** MultipleCatchBlock1 {

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

**try**{

**int** a[]=**new** **int**[5];

                a[5]=30/0;

               }

**catch**(ArithmeticException e)

                  {

                   System.out.println("Arithmetic Exception occurs");

                  }

**catch**(ArrayIndexOutOfBoundsException e)

                  {

                   System.out.println("ArrayIndexOutOfBounds Exception occurs");

                  }

**catch**(Exception e)

                  {

                   System.out.println("Parent Exception occurs");

                  }

               System.out.println("rest of the code");

    }

}

**Java Nested try block**

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

**Why use nested try block?**

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

/main try block

**try**

{

    statement 1;

    statement 2;

//try catch block within another try block

**try**

    {

        statement 3;

        statement 4;

//try catch block within nested try block

**try**

        {

            statement 5;

            statement 6;

     }

**catch**(Exception e2)

        {

//exception message

        }

    }

**catch**(Exception e1)

    {

//exception message

    }

}

//catch block of parent (outer) try block

**catch**(Exception e3)

{

//exception message

}

....

**Java finally block:-**

**Java finally block** is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

**Flowchart of finally block**



#### **Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).**

* finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.

Case 1: When an exception does not occur

where the Java program does not throw any exception, and the finally block is executed after the try block.

**class** TestFinallyBlock {

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

**try**{

//below code do not throw any exception

**int** data=25/5;

   System.out.println(data);

  }

//catch won't be executed

**catch**(NullPointerException e){

System.out.println(e);

}

//executed regardless of exception occurred or not

**finally** {

System.out.println("finally block is always executed");

}

**Case 2:** When an exception occurr but not handled by the catch block

Let's see the the following example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

**public** **class** TestFinallyBlock1{

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

**try** {

        System.out.println("Inside the try block");

        //below code throws divide by zero exception

**int** data=25/0;

       System.out.println(data);

      }

      //cannot handle Arithmetic type exception

      //can only accept Null Pointer type exception

**catch**(NullPointerException e){

        System.out.println(e);

      }

      //executes regardless of exception occured or not

**finally** {

        System.out.println("finally block is always executed");

      }

      System.out.println("rest of the code...");

      }

    }

Case: 3  **When an exception occurs and is handled by the catch block**

**Example:**

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

#### **Rule: For each try block there can be zero or more catch blocks, but only one finally block.**

#### **Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).**