**Exception Handling** **:**

**What is an exception?**

An Exception can be anything which interrupts the normal flow of the program. When an exception occurs program processing gets terminated and doesn’t continue further. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled.

**When an exception can occur?**  
Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions).

**Reasons for Exceptions**  
There can be several reasons for an exception. For example, following situations can cause an exception – Opening a non-existing file, Network connection problem, Operands being manipulated are out of prescribed ranges, class file missing which was supposed to be loaded and so on

## Error Vs Exception In Java :

## java.lang.Error and java.lang.Exception classes are sub classes of java.lang.Throwable class, but there exist some significant differences between them. java.lang.Error class represents the errors which are mainly caused by the environment in which application is running. For example, OutOfMemoryError occurs when JVM runs out of memory or StackOverflowError occurs when stack overflows.

## Where as java.lang.Exception class represents the exceptions which are mainly caused by the application itself. For example, NullPointerException occurs when an application tries to access null object or ClassCastException occurs when an application tries to cast incompatible class types.

1) Recovering from **Error** is not possible. The only solution to errors is to terminate the execution. Where as you can recover from **Exception** by using either try-catch blocks or throwing exception back to caller.

2) You will not be able to handle the **Errors** using try-catch blocks. Even if you handle them using try-catch blocks, your application will not recover if they happen. On the other hand, **Exceptions** can be handled using try-catch blocks and can make program flow normal if they happen.

3) **Exceptions** in java are divided into two categories – checked and unchecked. Where as all **Errors** belongs to only one category i.e unchecked.

4) Compiler will not have any knowledge about unchecked exceptions which include **Errors** and sub classes of RunTimeException because they happen at run time. Where as compiler will have knowledge about checked **Exceptions**. Compiler will force you to keep try-catch blocks if it sees any statements which may throw checked exceptions.

5) **Exceptions** are related to application where as **Errors** are related to environment in which application is running.

# Checked and unchecked exceptions :

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using [try-catch block](http://beginnersbook.com/2013/04/try-catch-in-java/) or it should declare the exception using [throws keyword](http://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/), otherwise the program will give a compilation error. It is named as **checked exception** because these exceptions are **checked** at Compile time.

Lets look at below example :

**public** **static** **void** readFile() {

FileInputStream fis = **null**;

fis = **new** FileInputStream("B:/myfile.txt"); // checked exception

**int** k;

**while** ((k = fis.read()) != -1) { // Checked Exception

System.***out***.print((**char**) k);

}

fis.close(); // checked exception

}

}

How to resolve checked exception ?

In two ways we can handle the checked exception

1. Use try and catch
2. Throw the exception out side the method

**1 )Try and Catch :**

public class CheckedExceptionEx {

public static void readFile () {

FileInputStream fis = null;

**try {**

fis = new FileInputStream("B:/myfile.txt");

} catch (FileNotFoundException e) {

e.printStackTrace();

}

int k;

try {

while ((k = fis.read()) != -1) {

System.*out*.print((char) k);

}

} catch (IOException e) {

e.printStackTrace();

}

try {

fis.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

**2 ) Throw the Exception out of method using “throws” keyword**

**public** **class** CheckedExceptionEx {

**public** **static** **void** readFile() **throws** IOException {

FileInputStream fis = **null**;

fis = **new** FileInputStream("B:/myfile.txt");

**int** k;

**while** ((k = fis.read()) != -1) {

System.***out***.print((**char**) k);

}

fis.close();

}

}

Note : Now its caller responsibility to handle the exception

**Here are the few other Checked Exceptions –**

* SQLException
* IOException
* DataAccessException
* ClassNotFoundException
* InvocationTargetException

## What are Unchecked exceptions?

Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of **RuntimeException** class.

Ex:

**public** **class** UncheckedExceptionEx {

**private** **void** display() {

System.***out***.println("display() method called");

}

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

UncheckedExceptionEx obj = **null**;

obj.display();

}

}

Here it wont give any compile time exceptions , but it give Nullpointer exception at runtime as we are calling display() method with null object

**Here are the few most frequently seen unchecked exceptions –**

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException
* IllegalArgumentException

## Exception Hierarchy:

## 

**Try and Catch and finally ?**

**Try** block is used to write the logic which causes the exception

**Catch** block is used to log the exception or to write any particular logic which needs to be executed on exception

Finally block is used to write the code which must need to be executed even though exception comes.

Like resource closing logic.

Ex:

**public** **class** TryCatchFinallyEx {

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

**int** a = Integer.*parseInt*(args[0]);

**int** b = Integer.*parseInt*(args[1]);

**try**

{

**int** result = a/b;

System.***out***.println("result : "+result);

}**catch**(ArithmeticException exception)

{

System.***out***.println("Exception occured :"+exception.getMessage());

}**finally** {

System.***out***.println("finally executed!");

}

}

}

First execute the program using the values 10 2

Out put :

result : 5

finally executed!

Then execute the program using the values 10 0 (which causes Airthmatic exception)

Output :

Exception occured :/ by zero

finally executed!

Observation : in case 1 : no exception came, so catch block not executed but finally executed

In case 2 : exception occurred so catch block executed and finally block also executed. Which proves that finally calls everytime

**Importand Exception class methods :**

**public** **class** ExceptionMethods {

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

**try** {

} **catch** (Exception e) {

// Prints the result of toString() along with the stack trace to

// System.err, the error output stream.

e.printStackTrace();

// Returns a detailed message about the exception that has occurred.

// This message is initialized in the Throwable constructor.

e.getMessage();

// Returns the cause of the exception as represented by a Throwable

// object.

e.getCause();

// Returns the name of the class concatenated with the result of

// getMessage().

e.toString();

}

}

}

## Multiple Catch Blocks :

**A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −**

### Syntax

**try {**

**}catch(ExceptionType1 e1) {**

**}catch(ExceptionType2 e2) {**

**}catch(ExceptionType3 e3) {**

**}**

**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.**

## Throw vs throws :

## Throw is used to create and throw the new instance of exception.

## Ex :

**public** **class** ThrowEx {

**public** **void** display(ThrowEx obj) {

**if** (obj == **null**) {

**throw** **new** RuntimeException("Object should not be null");

}

**else**

{

System.***out***.println("No Exception");

}

}

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

ThrowEx obj = **new** ThrowEx();

obj.display(**null**);

}

## }

**Output :** Exception in thread "main" java.lang.RuntimeException: Object should not be null

at com.gbn.exception.ThrowEx.display(ThrowEx.java:7)

at com.gbn.exception.ThrowEx.main(ThrowEx.java:17)

if obj is null then we are creating new instance of exception and we are throwing

**throws is used to throw the exception out of the method. It is caller responsibility to handle the exception**

ex :

**public** **class** ThrowsEX {

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

Thread.*sleep*(100);

}

}

In the above code sleep() method in the Thread class throwing the InterruptedException, as we are calling Threa.sleep() it is our responsibility to handle the exception (using try or throws)

**Rules in Exception Handling :**

1. **No statement allowed in between try and catch and finally**

**public** **class** NoStatementsAllowed {

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

**try** {

Thread.*sleep*(1000);

}

// Sytem.out.println("not allowed")

**catch** (InterruptedException e) {

e.printStackTrace();

}

//System.out.println("not allowed");

**finally** {

}

}

}

2 ) when using multiple catches first child need to be catched then parent

Legal case :

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) { ---- child

e.printStackTrace();

}**catch** (RuntimeException e) { -- chile to exception parent to IE

}

**catch** (Exception e) { -- parent

}

Illegal case :

**try** {

Thread.*sleep*(1000);

} **catch** (Exception e) { // not allowed

e.printStackTrace();

}**catch** (RuntimeException e) {

}

**catch** (InterruptedException e) {

# 3 )Exception handling in Method overriding

**Rule**:  An overriding method (the method of child class) can throw any [unchecked exceptions](http://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/), regardless of whether the overridden method (method of base class) throws exceptions or not. However the overriding method should not throw [checked exceptions](http://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/) that are new or broader than the ones declared by the overridden method. The overriding method can throw those checked exceptions, which have less scope than the exception(s) declared in the overridden method.