**Exception Handling**

## Exception

An **Exception** is an event, which occurs during the execution of a program, which disrupts the normal flow of the program's Instructions.

Or

Exceptions in Java are nothing but the objects representing the runtime errors and logical errors.

**Exception classes:**

The list of classes which are exclusively designed to represent the logical errors and runtime errors.

**Let’s see some of the exceptions:**

**Example 1:**

*public class ExceptionHandling*

*{*

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

*{*

*System.out.println("Stmt is executed");*

*Integer I = new Integer(1);*

*System.out.println(I);*

*System.out.println("Stmt is also executed");*

*}*

*}*

**Example 2:**

*public class ExceptionHandling*

*{*

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

*{*

*System.out.println("Stmt is executed");*

***Integer I = new Integer("ABC");***

***//Statement throws NumberFormatException***

*System.out.println("Stmt is not executed");*

*System.out.println(I);*

*}*

*}*

***Exception:******Exception in thread "Main Thread" java.lang.NumberFormatException: For input string: "ABC"***

**Example 3:**

*public class ExceptionHandling*

*{*

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

*{*

*System.out.println("Stmt is executed");*

***int i = 1000/0;***

***//Statement throws ArithmaticException : / by zero***

*System.out.println("Stmt is not executed");*

*}  
}*

**Uncaught Exception**   
  
A program that does not properly handle the exception at all, will end abnormally.  The program will then produce a message that describes the exception that occurred and where in the code it was generated.

## Errors

These are not exceptions, but the problems that are beyond the control of the user or the programmer.

**Example:**

Stack overflow, Virtual Machine Memory, AssertionError

**The following list describes some errors that are easy to make in writing Java ...**

* Class name does not match file name   
  (usually this is due to uppercase vs. lower case mistake or simple typo).
* misspelled variable name   
  (use of variable name does not match name in its declaration)
* missing semicolon after assignment or method call statement
* missing semicolon after variable definition
* missing semicolon after import statement
* import statement naming package not classes
* giving semicolon after method signature
* missing parenthesis "(" and ")" around condition in if or while statement
* missing parentheses "(" and ")" in method call without arguments
* missing the variable type for an argument in the parameter list of method declaration
* supplying the variable type for an argument in the parameter list of method call
* redefining the type of a variable (defining a variable  which is already defined)
* missing return statement at end of some unreachable path in code
* declaring method as static when it mentions instance variables
* confusing numeric char '2' with int 2
* confusing one character string "x" with character 'x'

**There are two types of errors are there.**

1. Compilation Error.
2. Runtime Error.

**Compilation Error :**

Any error that occurs at the time of compilation could be called as **compilation error**.

There are **three reasons**, because of which compilation error occurs, they are:

1. A compilation error occurs because of wrong syntax, which is known as “**Syntax Error”**.
2. A compiler error may occur because of the **environment required for the JVM to execute** a statement is not existing.

At the same time of compilation, the compiler checks whether the syntax of the statement is proper or not. If the syntax of the statement is proper, then the compiler checks whether JVM can execute that statement or not, i.e. the environment or whatever that is required for the JVM to execute that statement is existing or not.

If the compiler finds that the environment required for the JVM to execute a statement, is not existing, then the compiler wouldn’t compile that statement and it leads for the compilation error.

1. Compilation error may occur because of the **checked Exception**.

**Note:**

The compiler understands everything that happens at the time of execution of a program. **Thus Java compilers are known as “Strongly Typed”.**

**Example:**

*Class Demo{*

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

*A a=new A;*

*//C.E due to syntax error****🡪(1)***

*B b=new B();*

*//If class B does not exist at the time of compilation, then it gives C.E****🡪 (2)***

*Thread.sleep(100);*

*//C.E due to checked exception* ***🡪(3)***

*}*

*}*

**Runtime Error**

Any error that occurs at the time of execution of a program is known as “**Runtime Error**”.  
Every runtime error occurs only because of the logical errors. But every logical error may not lead to Runtime Error.

**Example 1:**

*public class Test1 {*

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

*int x=50;*

*int sum=0;*

*for(int i=0;i<x;i++){*

*sum=sum+i;*

*}*

*System.out.println("50 no. sum :"+sum);*

*}*

*}*

**Output:** 50 no. sum : 1225

This is an example of logical error leading to a wrong output, because it’s a program to calculate the sum of 50 numbers. But actually it’s calculating the sum of 49 numbers only.

So, every logical error may not lead to Runtime Error.

**Example 2:**

*public class Test {*

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

*int i=10;*

*int j=0;*

*int x=i/j;*

*System.out.println(x);*

*}*

*}*

This is also an example of logical error, because infinity cannot be assigned to the variable x.

So, we can say every runtime error occurs only because of the logical errors.

**Note:**

1. When JVM enters into the state of ambiguity, it just suspends the execution of the program in the middle.
2. So, Runtime error is nothing but that state of JVM, where it enters into the state of ambiguity or confusion.
3. JVM enters into the state of ambiguity or confusion because of wrong logic/wrong procedure. But every wrong logic may not lead JVM to enter into the state of ambiguity.

You will get an error message telling you the kind of error, and a **stack trace** that tells not only where the error occurred, but also what other [method](https://www.cis.upenn.edu/~matuszek/General/JavaSyntax/methods.html) or methods you were in. For example,

**Exception in thread "main" java.lang.NullPointerException**

**at Car.placeInCity(Car.java:25)**

**at City.<init>(City.java:38)**

**at City.main(City.java:49)**

This says that a NullPointerException was detected in the method placeCarInCity at line 25 in Car.java, which was called from the [constructor](https://www.cis.upenn.edu/~matuszek/General/JavaSyntax/constructors.html) for City at line 38 in City.java, which was called from the main method at line 49 in City.java. Sometimes there will be additional lines describing methods in the Java system itself; you can ignore these.

Runtime errors are intermediate in difficulty. Java tells you where it discovered that your program had gone wrong, but you need to trace back from there to figure out where the problem originated.

**Logical Error:**

**Any wrong logic which leads to the wrong output and ambiguity can be called as a logical error.**

The Java system, of course, has no idea what your program is *supposed* to do, so it provides no additional information to help you find the error.

Ways to track down a logic error include:

* Think about what the program must have done in order to produce the results it did. This will lead you to where the error must have occurred.
* Put in [print statements](https://www.cis.upenn.edu/~matuszek/General/JavaSyntax/print-statements.html) to help you figure out what the program is actually doing.
* Use a **debugger** to step through your program and watch what it does.

**Abnormal Termination:**

**The concept of suspending or terminating a program in the middle of its execution unconditionally and the control coming out of the program from the middle of the program without executing the last statement of the program is known as Abnormal Termination.**

**Abnormal termination is one of the most undesirable situations in programming which may occur because of runtime errors.**

**N.B:**

**In order to represent the state of ambiguity of JVM, there are list of predefined library classes called Exception Classes.**

**When JVM enters into a state of ambiguity, it just blindly creates the object of the corresponding exception class.**

**The exception should be handled. Else, every unhandled exception is reaching to the JVM and then the program terminates the program execution. So, abnormal termination is due to the exception class object reaching the JVM.**

## The Most Common Syntax Errors in Java:

## 1. Spelling Mistakes

### Simple spelling errors

The Java language is a strict language. A variable or method CAN NOT be used before it is declared and a common mistake is to misspell the variable/method name when using it

Here are some general rules for naming variables in java:

* Java variable names are case sensitive.
* Java names start with any letter or underscore (\_) or dollar sign ($).
* Java names can be of any length up to 64 characters

### Case-sensitive errors with classes

The Java language is ***Case Sensitive***! For example, Java will not recognize the word  **string** as a valid type in the language as you should have written **String**.

Java will generate an error message of the form ...

Line nn: Class xxxx not found in type declaration.

... where xxxx is the name of the class which has not been given the correct capitalization.

### Case-sensitive errors with variables and Methods

The Java language is ***Case Sensitive***! It is a common mistake to miss the fact that variables are case sensitive. For example, you may have declared the variable Name as an String and then later on in your program you try to refer to the variable name.

This gives rise to error messages of the form ...

Line nn: Undefined variable: xxxx

... where xxxx is the name of the variable which has been mistyped.

### Capitalization of Java key words

The Java language is ***Case Sensitive***! So that the method **System.out.print()** is different from the method  **system.out.print()** and the method **main()** is different than the method **Main()**.

The Java compiler will issue an error message such as ...

Line nn: class or interface declaration expected

... when it does not recognize a variable, method or Java Object because it is misspelled (including its case)

## 2. Bracketing Mistakes

### Missing } brackets

This is a common programming error in many programming languages and can be fixed by adding the closing bracket. All blocks of code must be started with a { bracket and ended/closed with a }bracket. This type of error can be reduced by using a good code indentation system.

### Missing ) brackets

This is a common programming error in many programming languages and can be fixed by adding the closing bracket. Methods must use the ( and ) to surround their parameter list.

### Improper bracket use

This is a common programming error for beginning programmers. It is important to know when to use the different types of bracket.

|  |  |
| --- | --- |
| { } | Used for blocks of code - like loops, if statements, classes and method |
| (  ) | Used in method headers to surround the parameter list  Use in Math equations |
| [  ] | Used in arrays to denote the array index position |

### Improper Nesting of { } brackets

This is a common programming error for beginning programmers. Each block of code surround by the { and } brackets must be properly structured so that two blocks of code are not straggled but rather one block of code must be contained inside of another block of code.

 This type of error can be reduced by using a good code indentation system.

### Missing brackets in a no-argument Method

When you use a method which has no arguments you should place brackets after the name of the method.

For example, if you have declared a method PrintBlankScreen with no arguments then you should code this as:

*PrintBlankScreen( )   
        rather than:   
PrintBlankScreen*

The compiler will usually indicate an error message of the form:

Line nn: Invalid expression statement

## 3. Improper Declaration/Use of Methods

### Writing the wrong format for a class method

Class methods have the form:

ClassName.MethodName(Argument(s))

A common error is to forget the class name. If you do, then you will get an error message of the form:

Line nn: '}' expected

### Specifying method arguments wrongly

When you define classes you should prefix each argument with the name of a scalar type or the name of an existing class. For example:

*public void tryIt(int a, int b, URL c)*

A common error that programmers from other languages make is to forget to prefix *every* argument with its type. For example, an erroneous version of the definition above would be:

*public void tryIt(int a, b URL c)*

This type of error will give rise to error messages of the form:

Line nn: Identifier expected

### Omitting void in methods

When a method returns no result, but just carries out some action, you need to use the keyword void in front of the name of the method. If you do not use this keyword, then it will give rise to error messages of the form:

Line nn: Invalid method declaration; return type required

### Omitting the return in a method

When a method returns a value, then the body of the method should include at least one return statement which returns the right type of value.

Failing to include a return statemnet will generate an error message of the form ...

Line nn: Return required at end of xxxx

... where xxxx is the method which does not contain the return.

### Treating a static method as if it were an instance method

Static methods are associated with messages sent to classes rather than objects. A common error is to send static method messages to objects. For example, in order to calculate the absolute value of an int value and place it into the int variable you should write:

*int result = Math.abs(value);*

*rather than:   
int result = value.abs();*

This gives rise to a variety of syntax errors. The most common one is of the form:

*Line nn: Method yyyy not found in class xxxx.*

where *yyyy* is the name of the method and *xxxx* is the name of the class within which it is called.

## 4. Forgetting to import a package

This one of the most common errors that inexperienced Java programmers make. If you forget to put the required import statement at the beginning of a program, then the compiler will respond with a message such as ...

**Line nn: Class xxxx not found in type declaration**

Don't forget, though, that java.lang is imported automatically and, hence, does not need an import statement.

### 5. Treating a static method as if it were an instance method

Static methods are associated with messages sent to classes rather than objects. A common error is to send static method messages to objects. For example, in order to calculate the absolute value of an int value and place it into the int variable you should write:

int result = Math.abs(value);

rather than:

int result = value.abs();

This gives rise to a variety of syntax errors. The most common one is of the form:

Line nn: Method yyyy not found in class xxxx.

where *yyyy* is the name of the method and *xxxx* is the name of the class within which it is called.

### 6. Assuming that == stands for value equality

== is used with scalars as a means of comparing values. However, when it is applied to objects then it compares addresses. For example, the if statement:

if(newObj1 == newObj2){

...

}

will execute the code denoted by the three dots *only* if the first object occupies the same address as the second object. If the objects occupied different addresses, but still had the same values for their instance variables, then it would evaluate to false. Unfortunately this does not give rise to any syntax errors, but will show up when any program containing the error is executed.

### 7. Making an instance variable private and then referring to it by name in another class

When you tag an instance variable as private you are not allowed to access it by name outside its class. The only way that you can access such instance variables is through methods which are declared in the class in which the instance variables are defined. This gives rise to error messages of the form:

Line nn: **Variable xx in class xxxx not accessible from class yyyy**

where xx is the private variable, xxxx is the class in which it is defined and class yyyy is the class in which it is referred to.

### 8. Using a variable before it is given a value

Again this is a common error found in both object-oriented and procedural languages. In Java, scalars are intialised to zero or some default value so there will be no error indication and any problems that arise will be signaled by erroneous results or some side effect such as an array going over its bounds. Objects will be initalised to null and any attempt to reference an uninitialised object will be caught at run time.

### 9. Assuming the wrong type of value is generated by a message

This is a common error to make when using the Java packages. A typical example is using a method which delivers a string that contains digits and treating it like an integer. For example, the method getInteger within java.lang.Integer delivers an Integer and any attempt to use that value as, say, an int will give rise to an error message of the form:

Line nn: **Incompatible type for declaration can't convert xxxx to yyyy**

### 10. Confusing prefix operators with postfix operators

This is an error that comes with any C-like language. Postfix operators such as ++ and -- deliver the old value of the variable to which they are applied, while prefix operators deliver the new value. Thus, if x is 45 and the statement:

*y = ++x*

*is executed, then y and x both become 46. If the statement*

*y = x++*

is executed, then y becomes 45, while x becomes 46. These errors will not be signalled at compile time, but will emerge during run time.

### 11. Forgetting that arguments are passed by reference to methods if they are objects

When an object is used as an argument to a method, then its address is passed over and not a value. This means that you can assign values to such arguments. If you treat them as values this will not strictly be an error, but will not be making use of the full facilities of an object-oriented programming language.

### 12. Forgetting that scalars are passed by value to methods

You cannot treat an argument which is a scalar as if it can be assigned to. This will not be signaled as a syntax error. However, it will show up as a run-time error when you write code which assumes that the scalar has been given a value by a method.

### 13. Misusing size when applied to strings and arrays

size is an instance variable associated with arrays and a method when associated with strings. If you mix them up by, for example writing:

*arrayVariable.size()*

*or*

*stringVariable.size*

then the first would generate an error message of the form:

Line nn: **Method size() not found in class java.lang.Object**

and the second would generate an error message of the form:

Line nn: **No variable size defined in java.lang.String**

### 14. Using a constructor which does not exist

You may use a constructor which has not been defined. For example, you may have a class X which has a one int constructor, a two int constructor and a three int constructor and yet you may have used a four int constructor. This would be picked up at compile time and an error of the form:

Line nn: **No constructor matching xxxx found in class yyyy**

would be generated, where *xxxx* is the signature of the constructor that you have tried using and *yyyy* is the name of the class which it should have been defined in.

### 15. Calling a constructor in a constructor with the same name

For example, you may have defined a class X with a two int constructor and a one int constructor and inside the two int constructor there is a reference to X(argument). This will be flagged as an error and will generate an error message of the form:

Line nn: ***Method xxxx not found in yyyy***

where *xxxx* is the name of the constructor and its arguments and *yyyy* is the name of the class which it is defined in. The solution is to use the this keyword.

### 16. Assuming that two-dimensional arrays are directly implemented in Java

This gives rise to erroneous code such as:

int [,] arrayVariable = new [10,20] int

This is illegal and will give rise to an errors of the form:

Line nn: **Missing term**

and:

Line nn: **']' expected**

You can implement many-dimensional arrays in Java, but they are treated like single-dimension arrays which contain single-dimensional arrays which contain single dimension arrays, etc.

### 17. Treating a scalar like an object

Scalars such as int and float are not objects. However, sometimes you want to treat them as such, for example when you want to deposit them in a Vector, as in the code:

*Vector vec = new Vector();*

*vec.addElement(12);*

If you write code such as that shown above then it will give rise to syntax errors of the form:

Line nn: **No method matching xxxx found in yyyy**

where xxxx is the name of the method which is used and yyyy is the name of the class which expects an Object. The solution is to use the object wrapper classes found in java.lang to convert them to objects.

### 18. Confusing scalars and their corresponding object types

When you have scalars such as int it is easy to write code which assumes that they can be treated as if they were objects. For example, the code:

*int y = 22;*

*Integer x = y;*

will give rise to an error message of the form:

Line nn: ***Incompatible type for declaration. Can't convert xxxx to yyyy***

where *xxxx* and *yyyy* are the classes involved.

### 19. Mistyping the header for the main method

When you want to execute a Java application you need to declare a method which starts with:

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

If you mistype any part of this line or miss out a keyword, then a run-time error will be generated. For example, if you miss out the keyword static then an error message of the form:

Exception in thread main.....

will be generated at run time.

**Exception handling**

The process of converting system error messages into user friendly error message is known as **Exception handling**.

Or

The concept of identifying exceptions or logical errors or runtime errors and catching these exceptions, and assigning these exception objects to the reference of the corresponding exception classes is known as **Exception Handling**.

If this exception is not handled properly, the rest of the program will not be executed. Hence, it causes the abrupt termination of the program. Therefore, you must handle the exceptions for the smooth flow of the program. To handle the run time exceptions, one mechanism is provided in java and it is called exception handling.

Because of Exception handling, we can avoid the exception class objects reaching to the JVM.

## Why use Exception Handling ?

* Handling the exception is nothing but converting system error generated message into user friendly error message.
* Whenever an exception occurs in the java application, JVM will create an object of appropriate exception of sub class and generates system error message, these system generated messages are not understandable by user. So, need to convert it into user friendly error message.
* You can convert system error message into user friendly error message by using exception handling feature of java.

**Example:**

When you divide any number by zero then system generate **/ by zero.** So, this is not understandable by user. So, we can convert this message into user friendly error message like **Don't enter zero for denominator.**

**Example 1:**

*public class ExceptionHandling{*

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

*System.out.println("This statement will be executed");*

*try{*

***Integer I = new Integer("abc");***

***//This statement throws NumberFormatException***

*} catch (Exception e) {*

*System.out.println("exception caught");*

*}*

*System.out.println("Now, This statement will also be executed");*

*}*

*}*

**Example 2:**

*public class ExceptionHandling{*

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

*System.out.println("This statement will be executed");*

*try{*

***int i = 1000/0;***

***//This statement throws ArithmaticException : / by zero***

*} catch (Exception e) {*

*System.out.println("Exception Caught");*

*}*

*System.out.println("Now, This statement will also be executed");*

*}*

*}*

## How Exception Handling Works in Java ?

1. When any error occurs in a method then new object ( i.e Exception Object) is created by a method.
2. Exception Object contains information about error such as type of error and state of the program.
3. Newly created exception object is passed to the Runtime System.
4. Runtime system will handle the exception to keep system stable

**Reasons for Exceptions**

There can be several reasons for an exception.

**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.

## Advantages of Exception Handling :

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling.

Let's take 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;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run.

**If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.**

**Therefore,**

* Exception handling allows us to control the normal flow of the program by using exception handling in program.
* It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.
* It also gives us the scope of organizing and differentiating between different errors types using a separate block of codes. This is done with the help of try-catch blocks.
* Using the concept of exception handling, we can identify the logical errors and provide in detail report of the logical errors to the end user and the programmer because of which debugging of the logical errors becomes simple and easy.

**Consider the following example** –

***public******class*** *ExceptionExample {*

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

*int num1 = 50;*

*int num2 = 0;*

*int ans;*

*ans = num1 / num2;*

***System****.out.println("Result of Division : " + ans);*

*}*

*}*

In this example, we know that any number divided by 0 will cause program to go into unexpected situation. If program contains such unexpected statements then it’s better to handle these unexpected results.

Below is the unexpected error output –

***Exception in thread "main" java.lang.ArithmeticException: / by zero at DivideByZeroNoExceptionHandling.main(***

***DivideByZeroNoExceptionHandling.java:7)***

Now we have done slight modification in the program by providing exception handling code in the program –

***public******class*** *ExceptionExample {*

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

*int num1 = 50;*

*int num2 = 0;*

*int ans;*

***try*** *{*

*ans = num1 / num2;*

***System****.out.println("Result of Division : " + ans);*

*}* ***catch****(****ArithmeticException*** *e) {*

***System****.out.println("Divide by Zero Error");*

*}*

*}*

*}*

So, the output of the above code will be –

**Divide by Zero Error**

**Situations that can cause exception:**

In Java, there are multiple situations that can cause exception. These situations are divided into 4 different types –

| **Situation** | **Cause of Exception** |
| --- | --- |
| **Code Error** | Exceptions occurred due to wrong or invalid data. If user tries to access the array element greater than size or something divided by zero error then these exceptions are called as code errors or data errors. |
| **Standard Method Exception** | Exceptions may be thrown if we try to access the standard methods with invalid input parameter. |
| **Own Exception** | User may generate his/her own exceptions depending on situation and type of code. |
| **Java Errors** | Errors occurred due to JVM |

**Hierarchy of Exception classes**

## java.lang.Throwable :

java.lang.Throwable is the super class of all errors and exceptions in java. Throwable class extends java.lang.Object class. The only argument of catch block must be it’s type or it’s sub class type.

**It has two sub classes.**

1) java.lang.Error

2) java.lang.Exception

## 1) java.lang.Error:

java.lang.Error is the super class for all types of errors in java.

Some of the common errors are,

* [**java.lang.VirtualMachineError**](http://docs.oracle.com/javase/7/docs/api/java/lang/VirtualMachineError.html)**:**  
   The most common virtualMachineErrors are [StackOverFlowError](http://docs.oracle.com/javase/7/docs/api/java/lang/StackOverflowError.html) and [OutOfMemoryError.](http://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html)
* [**java.lang.AssertionError**](http://docs.oracle.com/javase/7/docs/api/java/lang/AssertionError.html)
* [**java.lang.LinkageError**](http://docs.oracle.com/javase/7/docs/api/java/lang/LinkageError.html)**:**The common LinkageError are [NoClassDefFoundError](http://docs.oracle.com/javase/7/docs/api/java/lang/NoClassDefFoundError.html) and subclasses of [IncompatibleClassChangeError](http://docs.oracle.com/javase/7/docs/api/java/lang/IncompatibleClassChangeError.html).
* The most frequent IncompatibleClassChangeErrors are NoSuchMethodError, NoSuchFieldError, AbstractMethodError, IllegalAccessError and InstantiationError.

All sub classes of Error class are unchecked type of exceptions. i.e They occur during run time only.

## 2) java.lang.Exception :

java.lang.Exception is the super class for all types of Exceptions in java.

All sub classes of Exception class except sub classes of RunTimeException are checked type of exceptions.

Some of the common sub classes of Exception are,

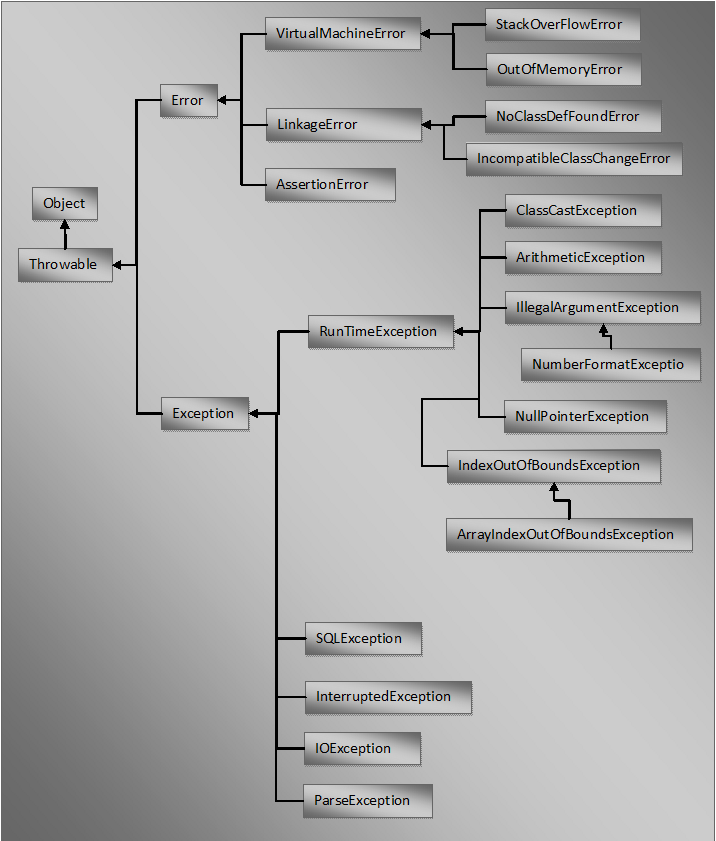
* [java.lang.RunTimeException](http://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html)

All sub classes of RunTimeException are unchecked type of exceptions. They occur during run time only. Some common RunTimeException are [ArithmeticException](http://docs.oracle.com/javase/7/docs/api/java/lang/ArithmeticException.html), [NumberFormatException](http://docs.oracle.com/javase/7/docs/api/java/lang/NumberFormatException.html), [NullPointerException](http://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html), [ArrayIndexOutOfBoundsException](http://docs.oracle.com/javase/7/docs/api/java/lang/ArrayIndexOutOfBoundsException.html) and [ClassCastException](http://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html).

* [java.lang.InterruptedException](http://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html)
* [java.lang.IOException](http://docs.oracle.com/javase/7/docs/api/java/io/IOException.html)
* [java.lang.SQLException](http://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)
* [java.lang.ParseException](http://docs.oracle.com/javase/7/docs/api/java/text/ParseException.html)

All above classes are placed in java.lang package.

**The above hierarchy can be represented as:**



## Type of Exception

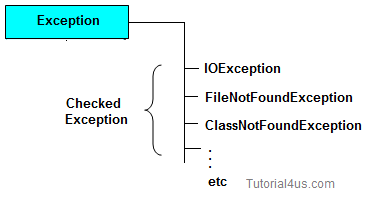
* Checked Exception
* Un-Checked Exception

## Checked Exception:

**Checked Exception** are the exception which checked at compile-time.

1. These exceptions are direct sub-class of java.lang.Exception class.
2. A checked exception is an exception which is error or a problem occurred because of code written by programmer
3. Checked Exception cannot be neglected by the programmer.
4. Suppose we are opening the file and file is not present then exception occurred during the compile time can be considered as checked exception.
5. These exceptions cannot be ignored at the time of compilation.
6. Class that extend throwable class except RuntimeException and Error is considered as Checked exception.

**Examples of Checked Exceptions**  
  
ClassNotFoundException  
IllegalAccessException  
NoSuchFieldException  
EOFException  
SQLException  
IOException  
DataAccessException  
ClassNotFoundException  
InvocationTargetException etc..



**There are two types of checked Exceptions are there:**

* 1. Pure Checked Exception.
  2. Partially Checked Exception.

**Pure Checked Exception:**

A Pure Checked Exception is a checked exception which must have all the sub-classes as checked exceptions, no sub-class is unchecked.

**Example**:   
 IOException

**Partially Checked Exception**

Partially checked Exception is a checked exception, which must have atleast one sub-class as unchecked Exception.

**Example:** Exception and Throwable

**Example**:

In this example we are reading the file myfile.txt and displaying its content on the screen. In this program there are three places where a checked exception is thrown as mentioned in the comments below.

* FileInputStream which is used for specifying the file path and name, throws FileNotFoundException.
* The read() method which reads the file content throws IOException and
* The close() method which closes the file input stream also throws IOException.

**Example:**

*import java.io.\*;*

*class Example {*

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

*FileInputStream fis = null;*

***/\*This constructor FileInputStream(File filename)***

***\* throws FileNotFoundException which is a checked***

***\* exception\*/***

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

*int k;*

***/\*Method read() of FileInputStream class also throws***

***\* a checked exception: IOException\*/***

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

*{*

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

*}*

***/\*The method close() closes the file input stream***

***\* It throws IOException\*/***

*fis.close();*

*}*

*}*

**Output**:

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

Unhandled exception type FileNotFoundException

Unhandled exception type IOException

Unhandled exception type IOException

**Why this compilation error?**

As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn’t handled/declared the exceptions, our program gave the compilation error.

**How to resolve the error?**

There are two ways to avoid this error.

**Method 1: Declare the exception using throws keyword:**

All three occurrences of checked exceptions are inside main() method. So, one way to avoid the compilation error is:

**Declare the exception in the method using throws keyword**

We may be thinking that our code is throwing FileNotFoundException and IOException both. Then why we are declaring the IOException alone. The reason is that IOException is a parent class of FileNotFoundException. So, by default it covers that.

If you want you can declare that too like this public static void main(String args[]) throws IOException, FileNotFoundException.

*import java.io.\*;*

*class Example {*

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

*}*

*}*

**Output**:  
  
File content is displayed on the screen.

The above approach is not good at all.

**Method 2: Handle them using try-catch blocks:**

We should give meaningful message for each exception type. So, that it would be easy for someone to understand the error. The code should be like this:

*import java.io.\*;*

*class Example {*

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

*FileInputStream fis = null;*

*try{*

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

*}catch(FileNotFoundException fnfe){*

*System.out.println("The specified file is not " +*

*"present at the given path");*

*}*

*int k;*

*try{*

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

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

*}*

*fis.close();*

*}catch(IOException ioe){*

*System.out.println("I/O error occurred: "+ioe);*

*}*

*}*

*}*

This code will run fine and will display the file content.

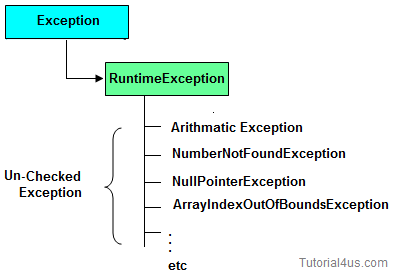
## Un-Checked/ Runtime Exception:

**Un-Checked Exceptions** are the exception both identifies or raised at run time.

1. These exceptions are **directly sub-class of java.lang.RuntimeException** class.
2. Runtime exceptions are ignored at the time of compilation.
3. In real time application, mostly we can handle un-checked exception.
4. Un-checked means not checked by compiler. So, un-checked exception are checked at run-time not compile time.

**Examples of Unchecked Exceptions:**

ArithmeticException  
ArrayIndexOutOfBoundsException  
NullPointerException  
llegalArgumentException  
NegativeArraySizeException etc.



Most of the times, these exceptions 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.

**Example**:

*class Example {*

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

*{*

*int num1=10;*

*int num2=0;*

***/\*Since I'm dividing an integer with 0***

***\* it should throw ArithmeticException\*/***

***int res=num1/num2;***

*System.out.println(res);*

*}*

*}*

If you compile this code, it would compile successfully. However, when you will run it, it would throw ArithmeticException.

That clearly shows that unchecked exceptions are not checked at compile-time, they are being checked at runtime.

**Example:**

*class Example {*

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

*{*

*int arr[] ={1,2,3,4,5};*

***/\*My array has only 5 elements but***

***\* I'm trying to display the value of***

***\* 8th element. It should throw***

***\* ArrayIndexOutOfBoundsException\*/***

***System.out.println(arr[7]);***

*}*

*}*

This code would also compile successfully. Since **ArrayIndexOutOfBoundsException** is also an unchecked exception.

**Note**:

It **doesn’t mean** that compiler is not checking these exceptions. So, we shouldn’t handle them. In fact we should handle them more carefully.

**Example:**

In the above example, there should be an exception message to user that they are trying to display a value which doesn’t exist in array so that user would be able to correct the issue.

*class Example {*

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

*{*

*try{*

*int arr[] ={1,2,3,4,5};*

*System.out.println(arr[7]);*

*}catch(ArrayIndexOutOfBoundsException e){*

*System.out.println("The specified index does not exist in array");*

*}*

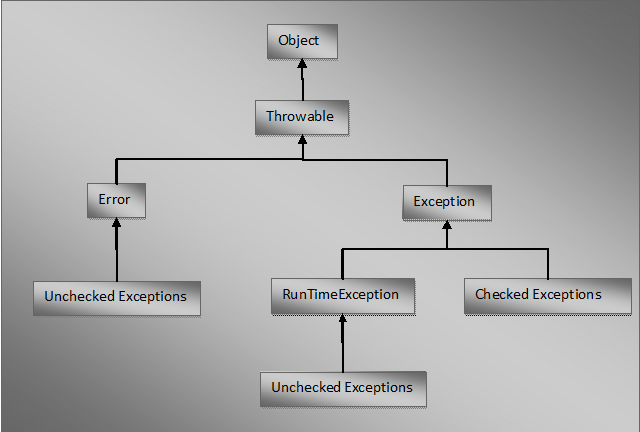
*}*

*}*

### Difference between checked Exception and un-checked Exception

|  |  |
| --- | --- |
| **Checked Exception** | **Un-Checked Exception** |
| Checked Exception are checked at compile time | Un-checked Exception are checked at run time |
| **Example**:  FileNotFoundException, NumberNotFoundException etc. | **Example:** ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. |
| They are known at **compile time**. | They are known at **run time**. |
| If these exceptions are not handled properly in the application, they give compile time error. | If these exceptions are not handled properly, they don’t give compile time error. But **application will be terminated prematurely at run time.** |
| All sub classes of java.lang.Exception Class except sub classes of RunTimeException are checked exceptions. | All sub classes of RunTimeException and sub classes of java.lang.Error are unchecked exceptions. |

The following diagram shows which are checked and unchecked exceptions in the hierarchy of exceptions.



## Difference between Error and Exception

## Both ****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. Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. Example:***OutOfMemoryError*** *occurs when JVM runs out of memory,* ***StackOverflowError*** *occurs when stack overflows, memory error, hardware error, JVM error etc.* ****java.lang.Exception**** class represents the exceptions which are mainly caused by the application itself. Exceptions are conditions within the code. A developer can handle such conditions and take necessary corrective actions.

## ****NullPointerException**** occurs when an application tries to access null object ****ClassCastException**** occurs when an application tries to cast incompatible class types. DivideByZero exception NullPointerException ArithmeticException ArrayIndexOutOfBoundsException

## Error Vs Exception In Java :

* Recovering from **Error** is not possible. The only solution to errors is to terminate the execution. Whereas you can recover from **Exception** by using either try-catch blocks or throwing exception back to caller.
* **Exceptions** in java are divided into two categories – checked and unchecked. Whereas all **Errors** belongs to only one category i.e unchecked.
* Compiler will not have any knowledge about unchecked exceptions which include **Errors** and sub classes of Runtime Exception because they happen at run time. Whereas 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.
* **Exceptions** are related to application whereas **Errors** are related to environment in which application is running.

**Below is the quick recap of above points.**

|  |  |
| --- | --- |
| **Errors** | **Exceptions** |
| Errors in java are of type java.lang.Error. | Exceptions in java are of type java.lang.Exception. |
| All errors in java are unchecked type. | Exceptions include both checked as well as unchecked type. |
| Errors happen at run time. They will not be known to compiler. | Checked exceptions are known to compiler whereas unchecked exceptions are not known to compiler because they occur at run time. |
| It is impossible to recover from errors. | You can recover from exceptions by handling them through try-catch blocks. |
| Errors are mostly caused by the environment in which application is running. | Exceptions are mainly caused by the application itself. |
| **Examples** : java.lang.StackOverflowError, java.lang.OutOfMemoryError | **Examples** : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, ClassCastException, NullPointerException |

### Keywords for Handling the Exception

* try
* catch
* finally
* throws
* throw

**Syntax for handling the exception**

***try***

***{***

***//This is the try block***

***//In this block, keep those statements which may***

***//throw run time exceptions***

***}***

***catch(Exception e)***

***{***

***//This is the catch block.***

***//It takes one argument of type java.lang.Exception***

***//This block catches the exceptions thrown by try block***

***}***

***finally***

***{***

***// statements which will execute compulsory***

***}***

**Working of Try-Catch blocks:**

Exception Object

try{

-----  
 -----

-----

-----

} catch(Exception ref){

-----

-----

-----

}

**Example:**

*public class Test {*

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

*int i=10;*

*int j=0;*

*try{*

*int x=i/j;*

*//Object of ArithmeticException class, created by JVM.*

*System.out.println(x);*

*}catch(Exception e){*

*System.out.println(e);*

*//Here, it will execute the println() defined to accept the object of object class.*

*}*

*}*

## *}*

## Example without Exception Handling:

*class ExceptionDemo {*

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

*int a=10, ans=0;*

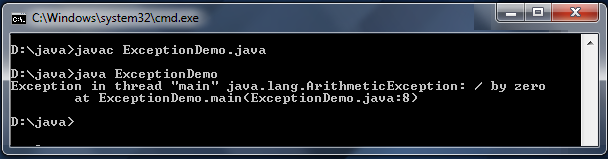
*ans=a/0;*

*System.out.println("Denominator not be zero");*

*}*

*}*

Abnormally terminate program and give a message like below, this error message is not understandable by user. So, we convert this error message into user friendly error message, like **"denominator not be zero**".



## Example with Exception Handling

*class ExceptionDemo {*

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

*int a=10, ans=0;*

*try{*

*ans=a/0;*

*}catch (Exception e){*

*System.out.println("Denominator not be zero");*

*}*

*}*

*}*

## Output

Denominator not be zero

### Try block

Inside **try block,** we write the block of statements which causes exceptions at run time.

In other words try block always contains **problematic statements.**

### Important points about try block

* If any exception occurs in try block, then CPU controls comes out to the try block and executes appropriate catch block.
* After executing appropriate catch block, even though we use run time statement, CPU control never goes to try block to execute the rest of the statements.
* Each and every try block must be immediately followed by catch block that is no intermediate statements are allowed between try and catch block.

## Syntax

*try*

*{*

*.....*

*}*

*/\* Here no other statements are allowed*

*between try and catch block \*/*

*catch()*

*{*

*....*

*}*

Each and every try block must contain at least one catch block. But it is highly recommended to write multiple catch blocks for generating multiple user friendly error messages.

## Java Nested Try Block:

One try block can contains another try block that is nested or inner try block can be possible.

## Syntax:

try

{

.......

try

{

.......

}

}

The try block contains a block of program statements within which an exception might occur.

A try block is always followed by a catch block, which handles the exception that occurs in associated try block.

A try block must followed by a Catch block or Finally block or both.

**Example:**

*public class ExceptionExample {*

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

*int result = 0;*

*int arr[] = new int[5];*

*arr[5] = 5;*

*result = 100 / 0;*

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

*}*

*}*

In the above example of nested try block we can see that

**arr[5] = 5;**

**result = 100 / 0;**

Both these lines will generate exception. If we put both these codes in same try block, then it will generate one exception and the next code won’t be executed.

*public class ExceptionExample {*

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

*int num1 = 10;*

*int num2 = 0;*

*int result = 0;*

*int arr[] = new int[5];*

*try {*

*try {*

*arr[5] = 5;*

*}catch (ArrayIndexOutOfBoundsException e) {*

*System.out.println("Err: Array Out of Bound");*

*}*

*try {*

*result = num1 / num2;*

*}catch (ArithmeticException e) {*

*System.out.println("Err: Divided by Zero");*

*}*

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

*}catch (Exception e) {*

*System.out.println("Exception Occured !");*

*}*

*}*

*}*

***Output****:*

*Err: Array Out of Bound*

*Err: Divided by Zero*

*Result of Division : 0*

In the above nested try block code each and every exception is handled. In order to handle each exception we need to use try block.

### Catch block

Inside **catch** block, we write the block of statements which will generates user friendly error messages.

### Catch block important points

* Catch block will execute, if exception occurs in try block.
* You can write multiple catch blocks for generating multiple user friendly error messages to make your application strong.
* **At a time only one catch block will execute out of multiple catch blocks**.
* In catch block you declare an object of sub class and it will be internally referenced by JVM.

## Example without Exception Handling

*class ExceptionDemo {*

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

*int a=10, ans=0;*

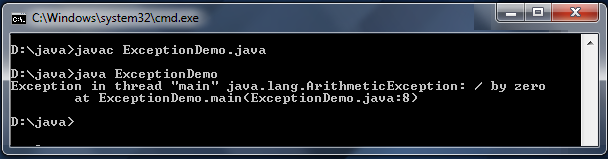
*ans=a/0;*

*System.out.println("Denominator not be zero");*

*}*

*}*

Abnormally terminate program and give a message like below, this error message is not understandable by user so we convert this error message into user friendly error message, like "denominator not be zero".



## Example of Exception Handling

*class ExceptionDemo{*

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

*int a=10, ans=0;*

*try{*

*ans=a/0;*

*} catch (Exception e){*

*System.out.println("Denominator not be zero");*

*}*

*}*

*}*

## Output:

Denominator not be zero

#### Multiple catch blocks in Java

1. A try block can have any number of catch blocks.  
2. A catch block that is written for catching the class Exception can catch all other exceptions

**Syntax**:

*catch(Exception e){*

*//This catch block catches all the exceptions*

*}*

3. If multiple catch blocks are present in a program then the above mentioned catch block should be placed at the last as per the exception handling best practices.  
  
4. If the try block is not [throwing any exception](http://beginnersbook.com/2013/04/throw-in-java/), the catch block will be completely ignored and the program continues.  
  
5. If the try block [throws an exception](http://beginnersbook.com/2013/04/java-throws/), the appropriate catch block (if one exists) will catch it  
  
**Example:**

* **catch(ArithmeticException e)** is a catch block that can catch ArithmeticException
* **catch(NullPointerException e)** is a catch block that can catch NullPointerException

6. All the statements in the catch block will be executed and then the program continues.

#### Example of Multiple catch blocks

*class ExceptionExample{*

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

*try{*

*int a[]=new int[7];*

*a[4]=30/0;*

*System.out.println("First print statement in try block");*

*}*

*catch(ArithmeticException e){*

*System.out.println("Warning: ArithmeticException");*

*}*

*catch(ArrayIndexOutOfBoundsException e){*

*System.out.println("Warning: ArrayIndexOutOfBoundsException");*

*}*

*catch(Exception e){*

*System.out.println("Warning: Some Other exception");*

*}*

*System.out.println("Out of try-catch block...");*

*}*

*}*

**Output**:

Warning: ArithmeticException

Out of try-catch block...

In the above example there are multiple catch blocks and these catch blocks executes sequentially, when an exception occurs in try block.

Which means if you put the last catch block **(catch(Exception e))** at the first place, just after try block then in case of any exception this block will execute as it has the [ability to handle all exceptions](http://beginnersbook.com/2013/04/java-exception-handling/).

This catch block should be placed at the last to avoid such situations.

**Example**:

*class ExceptionExample{*

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

*try{*

*int a[]=new int[7];*

*a[4]=30/0;*

*System.out.println("First print statement in try block");*

*}*

*catch(Exception e){*

*System.out.println("Warning: Some Other exception");*

*}*

*catch(ArrayIndexOutOfBoundsException e){*

*System.out.println("Warning: OutOfBoundsException");*

*// Unreachable catch block for ArithmeticException. It is already handled by the catch block for Exception.*

*}*

*catch(ArithmeticException e){*

*System.out.println("Warning: ArithmeticException");  
// Unreachable catch block for ArithmeticException. It is already handled by the catch block for Exception*

*}*

*System.out.println("Out of try-catch block...");*

*}*

*}*

***public******class*** *ExceptionHandling{*

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

***try****{*

***int*** *i = Integer.parseInt("abc");*

*//This statement throws NumberFormatException*

*}*

***catch****(Exception ex){*

*System.out.println("handles all exception types");*

*}*

***catch****(NumberFormatException ex){*

*//Compile time error*

*//This block becomes unreachable as*

*//exception is already handled by above catch block*

*}*

*}*

*}*

***import*** *java.util.\*;*

***class*** *ExceptionDemo {*

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

***int*** *a, b, ans=0;*

*Scanner s=****new*** *Scanner(System.in);*

*System.out.println("Enter any two numbers: ");*

***try*** *{*

*a=s.nextInt();*

*b=s.nextInt();*

*ans=a/b;*

*System.out.println("Result: "+ans);*

*}*

***catch****(ArithmeticException ae){*

*System.out.println("Denominator not be zero");*

*}*

***catch****(Exception e) {*

*System.out.println("Enter valid number");*

*}*

*}*

*}*

## Output:

Enter any two number: 5 0

Denominator not be zero

**Example:**

***public******class*** *ExceptionExample {*

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

***int*** *num1 = 10;*

***int*** *num2 = 0;*

***int*** *result = 0;*

***int*** *arr[] =* ***new******int****[5];*

***try*** *{*

*arr[0] = 0;*

*arr[1] = 1;*

*arr[2] = 2;*

*arr[3] = 3;*

*arr[4] = 4;*

*arr[5] = 5;*

*result = num1 / num2;*

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

*}****catch*** *(ArithmeticException e) {*

*System.out.println("Err: Divided by Zero");*

*}****catch*** *(ArrayIndexOutOfBoundsException e) {*

*System.out.println("Err: Array Out of Bound");*

*}*

*}*

*}*

### Output :

Err: Array Out of Bound

In the above example, we have two lines that might throw an exception i.e

arr[5] = 5;

Above statement can cause array index out of bound exception and

result = num1 / num2;

This can cause arithmetic exception. To Handle these two different types of exception we have included two catch blocks for single try block.

}catch (ArithmeticException e) {

**System**.out.println("Err: Divided by Zero");

}catch (ArrayIndexOutOfBoundsException e) {

**System**.out.println("Err: Array Out of Bound");

}

Now Inside the try block when exception is thrown then type of the exception thrown is compared with the type of exception of each catch block.

If type of exception thrown is matched with the type of exception from catch, then it will execute corresponding catch block.

**Example:**

*public class ExceptionHandling{*

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

*String[] s = {"abc", "123", null, "xyz"};   
//String array containing one null object*

*for (int i = 0; i < 6; i++) {*

*try{*

*int a = s[i].length() + Integer.parseInt(s[i]);*

*//This statement may throw NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException*

*}catch(NumberFormatException ex){*

*System.out.println("NumberFormatException will be caught here");*

*}*

*catch (ArrayIndexOutOfBoundsException ex){ System.out.println("ArrayIndexOutOfBoundsException will be caught here");*

*}*

*catch (NullPointerException ex) {*

*System.out.println("NullPointerException will be caught here");*

*}*

*System.out.println("After executing respective catch block, this statement will be executed");*

*}*

*}  
}*

## In the above example, a string array is iterated through for loop. First statement of try block (Line 11) may throw NumberFormatException because string array contain some non-numeric values or It may throw NullPointerException because string array contains one null object or it may throw ArrayIndexOutOfBoundsException because we are trying to iterate 6 elements, but actually string array contains only 4 elements. Depending upon the type of exception thrown, corresponding catch block will be executed.

## Notes :

1. At a time only **single catch block can be executed**. After the execution of catch block control goes to the statement next to the try block.
2. At a time Only single exception can be handled.
3. All the exceptions or catch blocks must be arranged in order.

## Catch Block must be arranged from Most Specific to Most General :

If we wrote catch block like this –

*}catch (Exception e) {*

***System****.out.println("Err: Exception Occurred");*

*}catch (ArrayIndexOutOfBoundsException e) {*

***System****.out.println("Err: Array Out of Bound");*

*}*

1. All the exceptions will be catched in the First Catch block because **Exception** is superclass of all the exceptions.
2. In the above program **ArrayIndexOutOfBoundsException** is a subclass of **Exception** Class.

## Catching Multiple Exception in single catch :

Whenever [try block](http://www.c4learn.com/java/java-try-catch-block/) will throw any exception of these type then and then only we can handle the exception.

*catch (IOException ex) {*

*System.out.println("IO Exception");*

*throw ex;*

*} catch (SQLException ex) {*

*System.out.println("SQL Exception");*

*throw ex;*

*}*

1. If in a try block we need to [handle multiple exceptions](http://www.c4learn.com/java/java-multiple-catch-blocks/) then we need to write exception handler for each type of exception.
2. We can combine the **multiple exceptions in catch block using Pipe (|) Operator**.

*catch (IOException|SQLException ex) {*

*System.out.println("Exception thrown");*

*throw ex;*

*}*

Above single catch block can handle IO, as well as SQL exceptions. So, it is better to use **Pipe Operator** to handle multiple exceptions instead of writing individual catch block for each exception. (**Pipe Operator** **applicable for 1.7version or above**).

From Java 7 onward, there is one more way for handling multiple exceptions. Multiple exceptions thrown by the try block can be handled by a single catch block using **pipe (|) operator**. By using pipe operator, the above example can be written as,

***public******class*** *ExceptionHandling{* ***public******static******void*** *main(String[] args) {  
String[] s = {"abc", "123",* ***null****, "xyz"};*

*//String array containing one null object*

***for*** *(****int*** *i = 0; i < 6; i++) {*

***try*** *{*

***int*** *a = s[i].length() + Integer.parseInt(s[i]);*

*//This statement may throw NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException*

*}*

***catch****(NumberFormatException | NullPointerException | ArrayIndexOutOfBoundsException ex) {*

*System.out.println("Now, this block handles NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException");*

*}  
 }*

*}  
}*

**Note:**

java.lang.Exception is super class of all types of exception. (Types of exceptions will be discussed later). It handles all types of exceptions. In the above example, all catch blocks can be replaced by one catch block which handles all types of exceptions. This type of exception handling comes very handy when you are not sure about the types of exceptions your code may throw.

***public******class*** *ExceptionHandling{*

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

*String[] s = {"abc", "123",* ***null****, "xyz"};*

*//String array containing one null object*

***for*** *(****int*** *i = 0; i < 6; i++){*

***try****{*

***int*** *a = s[i].length() + Integer.parseInt(s[i]);*

*//This statement may throw NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException*

*}*

***catch****(Exception ex)*

*{*

*System.out.println("This block handles all types of exceptions");*

*}*

*}*

*}*

*}*

**Note:**

*public class ExceptionHandling{*

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

*try{  
 int i = Integer.parseInt("abc");*

*//This statement throws NumberFormatException*

*}catch(NumberFormatException ex) {*

*System.out.println("It handles NumberFormatException");  
}*

*catch(Exception ex) {   
  System.out.println("This block handles all exception types");*

*}*

*catch (Throwable ex) {*

*System.out.println("Throwable is super class of Exception");*

*}*

*}*

*}*

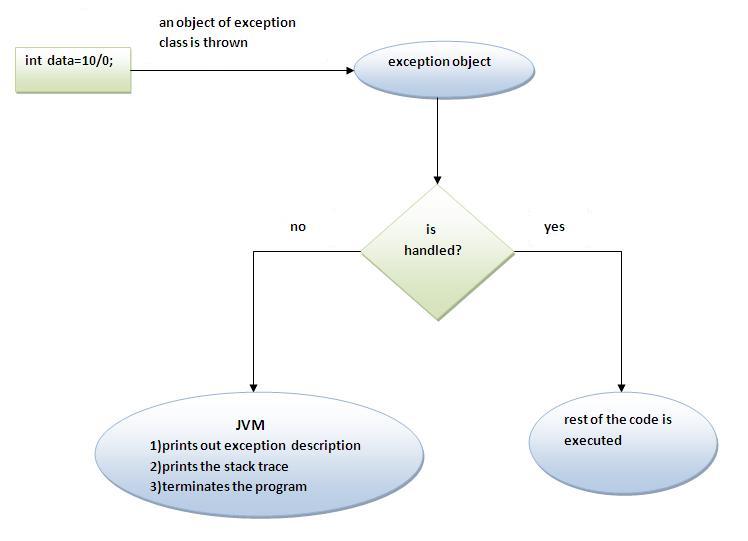
## \*\*Parameter Accepted by Catch Block is Final :

1. If a catch block handles more than one exception type, then the catch parameter is implicitly final.
2. In this example, the catch parameter ex is final.
3. We cannot assign any values to it within the catch block.

**Note :**

We can use Pipe (|) to catch multiple exceptions only in [Java SE 7](http://www.c4learn.com/java/java-download/) or in higher versions.

## 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 exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

#### Flow of try catch block

1. If an exception occurs in try block, then the control of execution is passed to the catch block from try block. The exception is caught up by the corresponding catch block. A single try block can have multiple catch statements associated with it, but each catch block can be defined for only one exception class. The program can also contain [nested](http://beginnersbook.com/2013/04/nested-try-catch/) [try-catch-finally blocks](http://beginnersbook.com/2013/05/flow-in-try-catch-finally/).
2. After the execution of all the try blocks, the code inside the finally block executes. It is not mandatory to include a finally [block](http://beginnersbook.com/2013/04/java-finally-block/) at all, but if you do, it will run regardless of whether an exception was thrown and handled by the try and catch blocks.

#### An example of Try catch in Java

***class*** *ExceptionExample {*

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

***int*** *num1, num2;*

***try*** *{*

*// Try block to handle code that may cause exception*

*num1 = 0;*

*num2 = 62 / num1;*

*System.out.println("Try block message");*

*}* ***catch*** *(ArithmeticException e) {*

*// This block is to catch divide-by-zero error*

*System.out.println("Error: Don't divide a number by zero");*

*}*

*System.out.println("I'm out of try-catch block in Java.");*

*}*

*}*

**Output**:

Error: Don't divide a number by zero

I'm out of try-catch block in Java.

**Example:**

## A string array, containing valid and invalid numeric values, is iterated through for loop. Each element of an array is parsed to primitive int type. Element with valid numeric value is parsed without throwing an exception. Element with invalid numeric value cannot be parsed to int type and it throws the NumberFormatException.

## This exception is caught in catch block making the flow of the program normal. finally block is executed for every iteration whether element is parsed or not.

***public******class*** *ExceptionHandling{*

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

*String[] s = {"abc", "123", "xyz", "456"};*

*//String Array containing valid and invalid numeric values*

***for*** *(****int*** *i = 0; i < s.length; i++){*

***try****{*

***int*** *intValue = Integer.parseInt(s[i]);*

*//This statement may throw NumberFormatException*

*}*

***catch****(NumberFormatException ex) {*

*System.out.println("The thrown NumberFormatException will be caught here");*

*}*

***finally****{*

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

*}*

*}*

*}*

*}*

## Output:

The thrown NumberFormatException will be caught here-***abc***

This block is always executed

This block is always executed-***123***

The thrown NumberFormatException will be caught here-***xyz***

This block is always executed

This block is always executed-***456***

## Note:

## When a statement throws an exception in the try block, the remaining part of the try block will not be executed. Program control comes out of the try block and enters directly into catch block.

*public class ExceptionHandling{*

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

*try{*

*int i = 10/0;*

*//This statement throws ArithmeticException*

*System.out.println("This statement will not be executed");*

*} catch(Exception ex) {*

*System.out.println("This block is executed immediately after an exception is thrown");*

*}*

*finally{*

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

*}*

*}*

*}*

## Note:

## try, catch and finally blocks form one unit. i.e You can’t keep other statements in between try, catch and finally blocks.

***public******class*** *ExceptionHandling{*

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

*System.out.println("You can keep any number of statements here");*

***try****{*

***int*** *i = 10/0;*

*//This statement throws ArithmeticException*

*System.out.println("Statement will not be executed");*

*}*

*//You can't keep statements here*

***catch****(ArithmeticException ex) {*

*System.out.println("This block is executed immediately after an exception is thrown");*

*}*

*//You can't keep statements here*

***finally****{*

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

*}*

*System.out.println("You can keep any number of statements here");*

*}*

*}*

## Note:

## You can display the description of an exception thrown using Exception object in the catch block.

***public******class*** *ExceptionHandling{*

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

***try****{*

*String s =* ***null****;*

*System.out.println(s.length());*

*//This statement throws NullPointerException*

*System.out.println("This statement will not be executed");*

*}*

***catch****(Exception ex){*

*System.out.println(ex);*

*//Output : java.lang.NullPointerException*

*ex.printStackTrace();  
//This prints stack trace of exception*

*}*

***finally****{*

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

*}*

*}*

## *}*

## Nested try catch:

*class Nest{*

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

*//Parent try block*

*try{*

*//Child try block1*

*try{*

*System.out.println("Inside block1");*

*int b =45/0;*

*System.out.println(b);*

*}*

*catch(ArithmeticException e1){*

*System.out.println("Exception: e1");*

*}*

*//Child try block2*

*try{*

*System.out.println("Inside block2");*

*int b =45/0;*

*System.out.println(b);*

*}*

*catch(ArrayIndexOutOfBoundsException e2){*

*System.out.println("Exception: e2");*

*}*

*System.out.println("Just other statement");*

*}*

*catch(ArithmeticException e3){*

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

*System.out.println("Inside parent try catch block");*

*}*

*catch(ArrayIndexOutOfBoundsException e4){*

*System.out.println("ArrayIndexOutOfBoundsException");*

*System.out.println("Inside parent try catch block");*

*}*

*catch(Exception e5){*

*System.out.println("Exception");*

*System.out.println("Inside parent try catch block");*

*}*

*System.out.println("Next statement..");*

*}*

*}*

**Output:**

Inside block1

Exception: e1

Inside block2

Arithmetic Exception

Inside parent try catch block

Next statement..

The above example shows Nested try catch use in Java. You can see that there are two try-catch block inside main try block’s body. I’ve marked them as block 1 and block 2 in above example.  
  
**Block1:**I have divided an integer by zero and it caused an arithmetic exception however the catch of block1 is handling arithmetic exception. So, "Exception: e1" got printed.

**Block2:**In block2 also, ArithmeticException occurred but block 2 catch is only handling ArrayIndexOutOfBoundsException. So, in this case control jump back to Main try-catch(parent) body. Since catch of parent try block is handling this exception that’s why “Inside parent try catch block” got printed as output.

**Parent try Catch block:**Since all the exception handled properly so program control didn’t get terminated at any point and at last “Next statement..” came as output.

**Note:**

The main point to note here is that whenever the child try-catch blocks are not handling any exception, the control comes back to the parent try-catch, if the exception is not handled there also then the program will terminate abruptly.

**Consider this example:**

Here we have deep(two level) nesting which means we have a try-catch block inside a child try block. To make you understand better. I have given the names to each try block in comments like try-block2 etc.

This is how the structure is: try-block3 is inside try-block2 and try-block2 is inside main try-block, you can say that the main try-block is a grand parent of the try-block3. Refer the explanation which is given at the end of this code.

***class*** *NestingDemo{*

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

*//main try-block*

***try****{*

*//try-block2*

***try****{*

*/try-block3*

***try****{*

***int*** *arr[]= {1,2,3,4};*

*/\* I'm trying to display the value of*

*\* an element which doesn't exist. The*

*\* code should throw an exception\*/*

*System.out.println(arr[10]);*

*}****catch****(ArithmeticException e){*

*System.out.print("Arithmetic Exception");*

*System.out.println("handled in try-block3");*

*}*

*}*

***catch****(ArithmeticException e){*

*System.out.print("Arithmetic Exception");*

*System.out.println(" handled in try-block2");*

*}*

*}*

***catch****(ArithmeticException e3){*

*System.out.print("Arithmetic Exception");*

*System.out.println(" handled in main try-block");*

*}*

***catch****(ArrayIndexOutOfBoundsException e4){*

*System.out.print("ArrayIndexOutOfBoundsException");*

*System.out.println(" handled in main try-block");*

*}*

***catch****(Exception e5){*

*System.out.print("Exception");*

*System.out.println(" handled in main try-block");*

*}*

*}*

*}*

**Output**:

***ArrayIndexOutOfBoundsException handled in main try-block***

As you can see that the ArrayIndexOutOfBoundsException has occurred in the grand child try-block3.

Since try-block3 is not handling this exception, the control then gets transferred to the parent try-block2 and looked for the catch handlers in try-block2. Since the try-block2 is also not handling that exception, the control got transferred to the main grand parent try-block where it found the appropriate catch block for exception.

This is how the **routing of exception is done in nested structure**.

**Note:**

## Nested try blocks are useful when different statements of try block throw different types of exceptions.

***public******class*** *ExceptionHandling*

*{*

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

*String[] s = {"abc", "123",* ***null****, "xyz"};*

*//String array containing one null object*

***for*** *(****int*** *i = 0; i < s.length; i++){*

***try*** *//Outer try block*

*{*

***int*** *a = s[i].length();*

*//This statement may throw NullPointerException*

***try*** *//Inner try block*

*{*

*a = Integer.parseInt(s[i]);*

*// statement may throw NumberFormatException*

*}*

***catch*** *(NumberFormatException ex)*

*//Inner catch block*

*{*

*System.out.println("NumberFormatException will be caught here");*

*}*

*}*

***catch****(NullPointerException ex)*

*//Outer catch block*

*{*

*System.out.println("NullPointerException will be caught here");*

*}*

*}*

*}  
}*

**Note:**

try blocks can be nested at any level.

***public******class*** *ExceptionHandling{*

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

*String[] s = {"abc", "123",* ***null****, "xyz"};*

*//String array containing one null object*

***for*** *(****int*** *i = 0; i < s.length; i++) {*

*//First Level try-catch block*

***try****{*

*i****nt*** *a = s[i].length();*

*//This statement may throw NullPointerException*

*//second level try-catch block*

***try****{*

*System.out.println(s[i+1]);*

*//This statement may throw ArrayIndexOutOfBoundsException*

*//third level try-catch block*

***try****{*

*a = Integer.parseInt(s[i]);*

*//This statement may throw NumberFormatException*

*}*

***catch*** *(NumberFormatException e) {*

*System.out.println("NumberFormatException will be caught here");*

*}*

*}*

***catch*** *(ArrayIndexOutOfBoundsException ex)*

*{*

*System.out.println("ArrayIndexOutOfBoundsException will be caught here");*

*}*

*}*

***catch****(NullPointerException ex) {*

*System.out.println("NullPointerException will be caught here");*

*}*

*}*

*}*

*}*

**Note:**

If the exception thrown by the inner try block cannot be caught by its catch block, then this exception is propagated to outer try blocks. Any one of the outer catch block should handle this exception otherwise program will terminate abruptly.

***public******class*** *ExceptionHandling{*

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

*String[] s = {"abc", "123",* ***null****, "xyz"};*

*//String array containing one null object*

***for*** *(****int*** *i = 0; i < s.length; i++){*

*//First Level try-catch block*

***try****{*

***int*** *a = s[i].length();*

*//This statement may throw NullPointerException*

*//second level try-catch block*

***try****{*

*System.out.println(s[i+1]);*

*//This statement may throw ArrayIndexOutOfBoundsException*

*//third level try-catch block*

***try****{*

*a = Integer.parseInt(s[i]);*

*//This statement may throw NumberFormatException*

*}****catch*** *(NullPointerException e) {*

*System.out.println("NumberFormatException will not be caught here");*

*}*

*}*

***catch*** *(NumberFormatException ex)*

*{*

*System.out.println("NumberFormatException will be caught here");*

*}*

*}*

***catch****(Exception ex) {*

*System.out.println("This block catches all types of exceptions");*

*}*

*}*

*}*

*}*

**Note:**

## In the following example also, try-catch blocks are nested. main() method calls nestedTry() method. nestedTry() method has one try-catch block. First statement in try block throws NumberFormatException which is not handled by it’s catch block. So, It propagates to try-catch block of main method which handles this exeption.

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

*try{*

*nestedTry();*

*}*

*catch(Exception ex)*

*{*

*System.out.println("NumberFormatException will be caught here");*

*}*

*}*

*static void nestedTry(){*

*try{*

*int i = Integer.parseInt("abc");*

*//This statement throws NumberFormatException*

*}*

*catch(NullPointerException ex) {*

*System.out.println("NumberFormatException will not be caught here");*

*}*

*}*

*}*

## Finally block

Inside **finally,** block we write the block of statements which will relinquish (released or close or terminate) the resource (file or database) where data store permanently.

1. A [finally statement](http://beginnersbook.com/2013/05/java-finally-return/) must be associated with a [try statement](http://beginnersbook.com/2013/04/try-catch-in-java/). It identifies a block of statements that needs to be executed regardless of whether or not an [exception occurs](http://beginnersbook.com/2013/04/java-exception-handling/) within the try block.

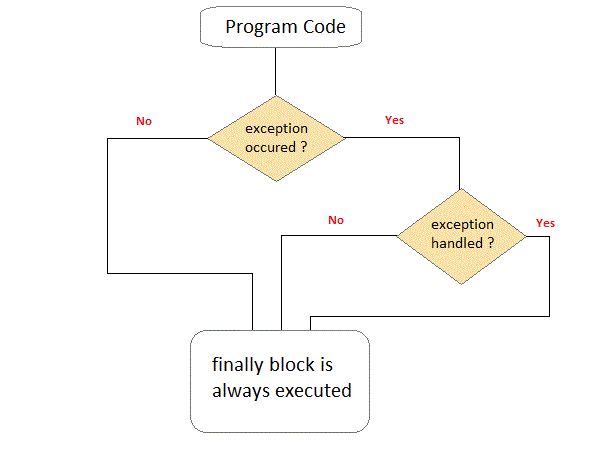
2. After all other try-catch processing is complete, the [code inside the finally block executes](http://beginnersbook.com/2013/05/flow-in-try-catch-finally/). It is not mandatory to include a finally block at all, but if you do, it will run regardless of whether an exception was thrown and handled by the try and catch parts of the block.

3. In normal execution the finally block is executed after try block. When any exception occurs first the catch block is executed and then finally block is executed.

4. An exception in the finally block, exactly [behaves like any other exception](http://beginnersbook.com/2013/04/user-defined-exception-in-java/).

5. The code present in the **finally block** executes even if the try or catch block contains control transfer statements like [return](http://beginnersbook.com/2013/05/java-finally-return/), break or continue.

## Flowchart for Finally Statement –



### finally block important points

* You can write finally block for the entire java program
* In some of the circumstances one can also write try and catch block in finally block.
* Finally is not a function, It is **keyword** in java
* Finally performs the functions such as closing the stream or file and closing connections.
* Finally always gets **executed before terminating the program** by JVM.
* Finally should be after try and catch and is optional
* **Each try block has minimum 0 and maximum 1 finally block**.

## Example

*class ExceptionDemo{*

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

*int a=10, ans=0;*

*try{*

*ans=a/0;*

*}catch (Exception e)*

*{*

*System.out.println("Denominator not be zero");*

*}*

*finally*

*{*

*System.out.println("I am from finally block");*

*}*

*}*

*}*

## Output

Denominator not be zero

I am from finally block

## Tip 1 :Finally block is optional

In the program we can have minimum zero and maximum one finally statement.

*public class MyExceptionExample {*

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

*try{*

*int i = 10/0;*

*} catch(Exception ex){*

*System.out.println("Inside 1st catch Block");*

*}*

*try{*

*int i = 10/10;*

*} catch(Exception ex){*

*System.out.println("Inside 2nd catch Block");*

*} finally {*

*System.out.println("Inside 2nd finally block");*

*}*

*}*

*}*

In the 1st try statement block, we can see that we don’t have finally block but in the 2nd try block we have finally block. So, finally block is always optional.

***finally*** *{*

*System.out.println("Inside 2nd finally block");*

*}*

2nd try block does not throw any exception but still after the execution of try block code executes finally statement block

### Output :

*Inside 1st catch Block*

*Inside 2nd finally block*

## \*\*Tip 2 : Finally block is not executed in below two cases

1. ***Using of the System.exit() method :***  
   If following statement is included in the try block (Terminating code using system exit)

*System.exit(0);*

1. ***Death of a Thread :***

If the thread executing the try or catch code is interrupted or killed, the finally statement block may not execute

1. **Due to an exception arising in the finally block.**

The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.

But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.

Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

**Note:**

If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

**Example:**

The try block of the writeList method that you've been working with here opens a PrintWriter. The program should close that stream before exiting the writeList method. This poses a somewhat complicated problem because writeList's try block can exit in one of three ways.

1. The new FileWriter statement fails and throws an IOException.
2. The list.get(i) statement fails and throws an IndexOutOfBoundsException.
3. Everything succeeds and the try block exits normally.

The runtime system always executes the statements within the finally block regardless of what happens within the try block. So it's the perfect place to perform cleanup.

The following finally block for the writeList method cleans up and then closes the PrintWriter.

*finally {*

*if (out != null) {*

*System.out.println("Closing PrintWriter");*

*out.close();*

*} else {*

*System.out.println("PrintWriter not open");*

*}*

*}*

**Important:**   
 The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is *always* recovered.

#### Finally block and Return statement :

If method returns a value and also has try, catch and finally blocks in it, then following two rules need to follow.

1) If finally block returns a value then try and catch blocks may or may not return a value.

2) If finally block does not return a value then both try and catch blocks must return a value.

**Example 1:**

***public******class*** *ReturnValueFromTryCatchFinally{*

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

*System.out.println(methodReturningValue());*

*}*

***static******int*** *methodReturningValue(){*

***try****{*

*//This block may or may not return a value as finally block is returning a value*

*}****catch*** *(Exception e) {*

*//This block may or may not return a value as finally block is returning a value*

*}*

***finally****{*

***return*** *20;*

*}*

*}*

*}*

**Example 2**:

*public class ReturnValueFromTryCatchFinally{*

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

*System.out.println(methodReturningValue());*

*}*

*static int methodReturningValue(){*

*try{*

*return 10;*

*}catch (Exception e) {*

*return 20;*

*}*

*finally{*

*//Now, This block may or may not return a value*

*//as both try and catch blocks are returning a value*

*}*

*}*

*}*

Finally block executes, even if there is a return statement in try-catch block.

**Example:**

*class JavaFinally*

*{*

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

*{*

*System.out.println(JavaFinally.myMethod());*

*}*

*public static int myMethod()*

*{*

*try {*

*return 112;*

*}*

*finally {*

*System.out.println("This is Finally block");*

*System.out.println("Finally block ran even after return statement");*

*}*

*}*

*}*

**Output:**

This is Finally block

Finally block ran even after return statement

112

**Example:**

***class*** *JavaFinally{*

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

*System.out.println(JavaFinally.myMethod());*

*}*

***public******static*** *String myMethod() {*

***try*** *{*

***return*** *"112";*

*}****catch****(Exception E){*

*System.out.println("Hi");*

*}****finally****{*

*System.out.println("Hello");*

*}*

***return******null****;*

*}*

*}*

**Output**:  
Hello

112

**Note**:

**If try-catch-finally blocks are returning a value according to above rules, then you should not keep any statements after finally block. Because they become unreachable and in Java, Unreachable code gives compile time error.**

*public class ReturnValueFromTryCatchFinally{*

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

*try{*

*return;*

*}*

*catch (Exception e){*

*return;*

*}*

*finally{*

*return;*

*}*

*System.out.println("Unreachable code");*

*//Compile Time Error : Unreachable Code*

*}*

*}*

**Note:**

**finally block overrides any return values from try and catch blocks.**

*public class ReturnValueFromTryCatchFinally{*

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

*System.out.println(methodReturningValue()); //Output : 50*

*}*

*static int methodReturningValue(){*

*try{*

*return 10;*

*}*

*catch (Exception e){*

*return 20;*

*}*

*finally{*

*return 50; //This method returns 50 not 10 or 20*

*}*

*}*

*}*

**Note:**

**finally block will be always executed even though try and catch blocks are returning the control.**

*public class ReturnValueFromTryCatchFinally{*

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

*System.out.println(methodReturningValue());*

*//Output : 10*

*}*

*static int methodReturningValue() {*

*try{*

*return 10;*

*//control will not be passed to main() method here*

*}*

*catch (Exception e) {*

*return 20;*

*//Control will not be passed to main() method here*

*}*

*finally{*

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

*//Control will be passed to main() method after executing this block*

*}*

*}*

*}*

***Example 1:***

*public class ReturnValueFromTryCatchFinally{*

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

*System.out.println(methodReturningValue());*

*}*

*static String methodReturningValue(){*

*String s = null;*

*try{*

*s = "return value from try block";*

*return s;*

*}*

*catch (Exception e) {*

*s = s + "return value from catch block";*

*return s;*

*}*

*finally{*

*s = s + "return value from finally block";*

*}*

*}*

*}*

**OUTPUT** :

return value from try block

***Example 2:***

*public class ReturnValueFromTryCatchFinally{*

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

*System.out.println(methodReturningValue());*

*}*

*static int methodReturningValue(){*

*int i = 0;*

*try{*

*i = 1;*

*return i;*

*}*

*catch (Exception e){*

*i = 2;*

*return i;*

*}*

*finally{*

*i = 3;*

*}*

*}*

*}*

**OUTPUT** : 1

**Example 3:**

*public class ReturnValueFromTryCatchFinally{*

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

*System.out.println(methodReturningValue());*

*}*

*static int methodReturningValue(){*

*int i = 0;*

*try{*

*i = 1;*

*return i;*

*}*

*catch (Exception e) {*

*i = 2;*

*}*

*finally{*

*i = 3;*

*}*

*return i;*

*}*

#### }

**OUTPUT** : 1

**Example 4:**

*public class ReturnValueFromTryCatchFinally{*

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

*System.out.println(methodReturningValue());*

*}*

*static int methodReturningValue(){*

*try{*

*int i = Integer.parseInt("123");*

*return 20;*

*}*

*finally{*

*return 50;*

*}*

*}*

*}*

**OUTPUT** : 50

**Example 5:**

*public class ReturnValueFromTryCatchFinally{*

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

*System.out.println(methodReturningValue());*

*}*

*static int methodReturningValue() {*

*try{*

*int i = Integer.parseInt("abc");*

*//This statement throws NumberFormatException*

*return 20;*

*}*

*finally{*

*return 50;*

*}*

*}*

*}*

**OUTPUT** : 50

#### Finally and Close()

**Close()** is generally used to close all the open streams in one go. It’s a good practice to use close() inside finally block.

Since finally block executes, even if exception occurs. So, you can be sure that all input and output streams are closed properly regardless of whether the exception occurs or not.

**Example:**

*....*

*try{*

*OutputStream osf = new FileOutputStream( "filename" );*

*OutputStream osb = new BufferedOutputStream(opf);*

*ObjectOutput op = new ObjectOutputStream(osb);*

*try{*

*output.writeObject(writableObject);*

*}*

*finally{*

*op.close();*

*}*

*}*

*catch(IOException e1){*

*System.out.println(e1);*

*}*

*...*

#### Finally block without catch

A try-finally block is possible without catch block. Which means a try block can be used with finally without having a catch block.

*...*

*InputStream input = null;*

*try {*

*input = new FileInputStream("inputfile.txt");*

*}*

*finally {*

*if (input != null) {*

*try {*

*input.close();*

*}catch (IOException exp) {*

*System.out.println(exp);*

*}*

*}*

*}*

*...*

#### Finally block and System.exit()

**System.exit()** statement behaves differently than **return statement**. Unlike return statement, whenever System.exit() gets called in try block then **Finally block** doesn’t get executed.

**Example:**

*....*

*try {*

*//try block*

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

*System.exit(0)*

*}*

*catch (Exception exp) {*

*System.out.println(exp);*

*}*

*finally {*

*System.out.println("Java finally block");*

*}*

*....*

In the above example, if the **System.exit(0)** gets called without any exception then finally won’t execute.

However, if any exception occurs while calling **System.exit(0)** then finally block will be executed.

#### Handling try-catch-finally block

* Either a try statement should be associated with a catch block or with finally.
* Since catch performs exception handling and finally performs the cleanup, the best approach is to merge both of them.

**Syntax**:

*try{*

*//statements that may cause an exception*

*}*

*catch (…)‏{*

*//error handling code*

*}*

*finally{*

*//statements to be executed*

*}*

#### Examples of Try catch finally blocks

**Example 1:**

**Below example illustrates finally block when no exception occurs in try block**

*class Example1{*

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

*try{*

*System.out.println("First statement of try block");*

*int num=45/3;*

*System.out.println(num);*

*}*

*catch(ArrayIndexOutOfBoundsException e){*

*System.out.println("ArrayIndexOutOfBoundsException");*

*}*

*finally{*

*System.out.println("finally block");*

*}*

*System.out.println("Out of try-catch-finally block");*

*}*

*}*

**Output:**

*First statement of try block*

*15*

*finally block*

*Out of try-catch-finally block*

**Example 2:**

Below example illustrates finally block execution, when exception occurs in try block but doesn’t get handled in catch block.

*class Example2{*

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

*try{*

*System.out.println("First statement of try block");*

*int num=45/0;*

*System.out.println(num);*

*}*

*catch(ArrayIndexOutOfBoundsException e){*

*System.out.println("ArrayIndexOutOfBoundsException");*

*}*

*finally{*

*System.out.println("finally block");*

*}*

*System.out.println("Out of try-catch-finally block");*

*}*

*}*

**Output:**

*First statement of try blockException in thread "Main Thread" java.lang.ArithmeticException: / by zero*

*finally block*

*at Example2.main(Example2.java:5)*

**Example 3:**

**Below example illustrates execution of finally, when exception occurs in try block and handled in catch block.**

*class Example3{*

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

*try{*

*System.out.println("First statement of try block");*

*int num=45/0;*

*System.out.println(num);*

*}*

*catch(ArithmeticException e){*

*System.out.println("ArithmeticException");*

*}*

*finally{*

*System.out.println("finally block");*

*}*

*System.out.println("Out of try-catch-finally block");*

*}*

*}*

**Output:**

First statement of try block

ArithmeticException

finally block

Out of try-catch-finally block

## Tips for Try-Catch Block :

1. The code within try block may or may not raise an exception. If Try block does not throw any exception then catch block gets executed.
2. The catch block contains an exception handler and some statements used to overcome that exception
3. Each and every Try Block must have catch or finally statement associated with it.

| **Try Block Present** | **Catch Block Present** | **Finally Block Present** | **Legal Statement** |
| --- | --- | --- | --- |
| Present | Present | - | Legal/Complete Try Block |
| Present | - | Present | Legal/Complete Try Block |
| Present | - | - | Illegal/Incomplete Try Block |

## Throw Statement

## throw Keyword :

1. The throw keyword is used to explicitly throw an exception.
2. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exception.

### Requirement :

**We want to show an error message, if the marks of student are below 75. We need to throw an error message – “Please reappear for exam”.**

*class StudentExcepeption{*

*static void validateMarks(int age){*

*if(marks < 75)*

*throw new ArithmeticException("Reappear for exam");*

*else*

*System.out.println("Student is having Distinction");*

*}*

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

*validateMarks(71);*

*System.out.println("Remaining code...");*

*}*

*}*

***Output:***

*Exception in thread "Main Thread" java.lang.ArithmeticException: Reappear for exam*

*at ExceptionalHandling.Test.validateMarks(Test.java:5)*

*at ExceptionalHandling.Test.main(Test.java:11)*

## Exception thrown and Type of Exception Object

Exception thrown by try block is of type Arithmetic Exception. So, it can be caught inside the catch block, because we have provided the handler of same type.

*try {*

*result = number1 / number2;*

*System.out.println("Result of Division : " + ans);*

*}catch(FileNotFoundException e) {*

*System.out.println("Divide by Zero Error");*

*}*

If we execute the above code instead then thrown exception is of type **Arithmetic Exception** and Exception Type provided into Catch block is of type **FileNotFoundException** then exception may not be catched. Thus above code can cause compile time error.

*try {*

*result = number1 / number2;*

*System.out.println("Result of Division : " + ans);*

*}catch(Exception e) {*

*System.out.println("Divide by Zero Error");*

*}*

In the above code, we can catch all types of exception because Exception is superclass of all the exceptions.

# How to throw exception in java with example?

In java, we have already defined exception classes such as ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException etc.

There are certain conditions defined for these exceptions and on the occurrence of those conditions they are implicitly thrown by JVM (java virtual machine).

**Do you know that a programmer can create a new exception and throw it explicitly**?

These exceptions are known as [user-defined exceptions](http://beginnersbook.com/2013/04/user-defined-exception-in-java/).

In order to throw user defined exceptions, [throw keyword](http://beginnersbook.com/2013/12/throw-keyword-example-in-java/) is being used.

You can also throw an already defined exception like ArithmeticException, IOException etc.

## Syntax of throw statement

*throw AnyThrowableInstance;*

**Example:**

*//A void method*

*public void sample()*

*{*

*//Statements*

*//if (somethingWrong) then*

*IOException e = new IOException();*

*throw e;*

*//More Statements*

*}*

**Note:**

A call to the above mentioned sample method should be always placed in a try block as it is throwing a [checked exception](http://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/) – IOException. This is how it the call to above method should be done:

*MyClass obj = new MyClass();*

*try{*

*obj.sample();*

*}catch(IOException ioe)*

*{*

*//Your error Message here*

*System.out.println(ioe);*

*}*

**Exceptions in java are compulsorily of type Throwable.**

If you attempt to throw an object that is not throwable, the compiler refuses to compile your program and it would show a compilation error.

**Flow of execution while throwing an exception using throw keyword**

Whenever a throw statement is encountered in a program the next statement doesn’t execute. Control immediately transferred to catch block to see if the thrown exception is handled there.

If the exception is not handled there, then next catch block is being checked for exception and so on.

If none of the [catch block](http://beginnersbook.com/2013/04/try-catch-in-java/) is handling the thrown exception then a system generated exception message is being populated on screen, same what we get for un-handled exceptions.

**Example:**

*class ThrowDemo{*

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

*try{*

*char array[] = {'a','b','g','j'};*

*/\*I'm displaying the value which does not*

*\* exist so this should throw an exception*

*\*/*

*System.out.println(array[78]);*

*}catch(ArithmeticException e){*

*System.out.println("Arithmetic Exception!!");*

*}*

*}*

}

**Output**:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:

78 at beginnersbook.com.ThrowDemo.main(Details.java:9)

Since the exception thrown was not handled in the catch blocks, the system generated exception message got displayed for that particular exception.

**Example 1:**

**How to throw your own exception explicitly using throw keyword**

*package beginnersbook.com;*

*class MyOwnException extends Exception {*

*public MyOwnException(String msg){*

*super(msg);*

*}*

*}*

*class EmployeeTest {*

*static void employeeAge(int age) throws MyOwnException{*

*if(age < 0)*

*throw new MyOwnException("Age can't be less than zero");*

*else*

*System.out.println("Input is valid!!");*

*}*

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

*try {*

*employeeAge(-2);*

*}*

*catch (MyOwnException e) {*

*e.printStackTrace();*

*}*

*}*

*}*

**Output:**

beginnersbook.com.MyOwnException: Age can't be less than zero

**Points to Note:**

Method call should be in try block as it is throwing an exception.

**Example2:**

**How to throw an already defined exception using throw keyword**

*package beginnersbook.com;*

*class Exception2{*

*static int sum(int num1, int num2){*

*if (num1 == 0)*

*throw new ArithmeticException("First parameter is not valid");*

*else*

*System.out.println("Both parameters are correct!!");*

*return num1+num2;*

*}*

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

*int res=sum(0,12);*

*System.out.println(res);*

*System.out.println("Continue Next statements");*

*}*

*}*

**Output:**

Exception in thread main java.lang.ArithmeticException: First parameter is not valid.

**Example**:

*public class ExceptionHandling{*

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

*try{*

*methodWithThrow();*

*}*

*catch(NumberFormatException ex) {*

*System.out.println("NumberFormatException object thrown in methodWithThrow() method will be handled here");*

*}*

*}*

*static void methodWithThrow(){*

*try{*

*NumberFormatException ex = new NumberFormatException("This is an object of NumberFormatException");*

*throw ex;*

*//throwing NumberFormatException object explicitly using throw keyword*

*}*

*catch(ArithmeticException ex) {*

*System.out.println("Explicitly thrown NumberFormatException object will not be caught here");*

*}*

*}*

*}*

## Re-throwing An Exception :

We all know that exceptions occurred in the try block are caught in catch block. Thus caught exceptions can be re-thrown using **throw** keyword. Re-thrown exception must be handled somewhere in the program, otherwise program will terminate abruptly. For example,

***public******class*** *ExceptionHandling{*

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

***try****{*

*methodWithThrow();*

*}*

***catch****(NullPointerException ex) {*

*System.out.println("NullPointerException Re-thrown in methodWithThrow() method will be handled here");*

*}*

*}*

***static******void*** *methodWithThrow() {*

***try****{*

*String s =* ***null****;*

*System.out.println(s.length());*

*//This statement throws NullPointerException*

*}*

***catch****(NullPointerException ex) {*

*System.out.println("NullPointerException is caught here");*

***throw*** *ex;*

*//Re-throwing NullPointerException*

*}*

*}*

*}*

# Throws clause

## Use of throws keyword in Java

1. The [**throws keyword**](http://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/) is used in method declaration, in order to explicitly specify the exceptions that a particular method might throw. When a method declaration has one or more exceptions defined, using throws clause then the method-call must handle all the defined exceptions.  
  
2. When defining a method you must include a throws clause to [declare those exceptions](http://beginnersbook.com/2013/04/java-exception-handling/) that might be thrown but doesn’t get caught in the method.  
  
3. If a method is using throws clause along with few exceptions then this implicitly tells other methods that – “ **If you call me, you must handle these exceptions that I throw**”.

### Syntax of Throws in java:

*void MethodName() throws ExceptionName{*

*Statement1*

*...*

*...*

*}*

**Example:**

*public void sample() throws IOException{*

*//Statements*

*//if (somethingWrong)*

*IOException e = new IOException();*

*throw e;*

*//More Statements*

*}*

**Note:**

In case a method throws more than one exception, all of them should be listed in throws clause.

**Example:**

*public void sample() throws IOException, SQLException*

*{*

*//Statements*

*}*

The above method has both IOException and SQLException listed in throws clause. There can be [any number of exceptions](http://beginnersbook.com/2013/04/user-defined-exception-in-java/) defined using throws clause.

#### Complete Example of Java throws Clause

*class Demo{*

*static void throwMethod() throws NullPointerException{*

*System.out.println ("Inside throw Method");*

*throw new NullPointerException ("Demo");*

*}*

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

*try{*

*throwMethod();*

*}*

*catch (NullPointerException exp){*

*System.out.println ("The exception get caught" +exp);*

*}*

*}*

*}*

**The output of the above program is:**

Inside throwMethod

The exception get caught java.lang. *NullPointerException*: Demo

## Example:

***package*** *ExceptionalHandling;*

***class*** *Test {*

***static******void*** *throwMethod()* ***throws*** *NullPointerException{*

*System.out.println ("Inside throw Method");*

***throw******new*** *NullPointerException("hi");*

*}*

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

***try****{*

*throwMethod();*

*}*

***catch*** *(ArrayIndexOutOfBoundsException exp){*

*System.out.println ("The ArrayIndexOutOfBoundsException get caught : " +exp);*

*}****catch*** *(NullPointerException exp){*

*System.out.println ("The NullPointerException get caught : " +exp);*

*}****catch*** *(Exception exp){*

*System.out.println ("The General Exception get caught : " +exp);*

*}*

*}*

*}*

**Output:**

**Inside throw Method**

**The NullPointerException get caught : java.lang.NullPointerException: hi**

## Note:

## Multiple exceptions can be declared using throws keyword separated by commas.

***public******class*** *ExceptionHandling{*

***static******void*** *methodWithThrows()* ***throws*** *NumberFormatException, NullPointerException {*

***int*** *i = Integer.parseInt(“;abc”);*

*//This statement throws NumberFormatException*

*String s =* ***null****;*

*System.out.println(s.length());*

*//This statement throws NullPointerException*

*}*

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

***try****{*

*methodWithThrows();*

*}*

***catch****(Exception ex){*

*System.out.println(“This block can handle all types of exceptions”);*

*}*

*}*

*}*

## Note:

## The main use of throws keyword in java is that an exception can be propagated through method calls.

***class*** *Test {*

***static******void*** *methodOne()* ***throws*** *NumberFormatException{*

*System.out.println("in mehodone");*

***int*** *i = Integer.parseInt("abc");*

*//This statement throws NumberFormatException*

*}*

***static******void*** *methodTwo()* ***throws*** *NumberFormatException{*

*methodOne();*

*//NumberFormatException is propagated here*

*}*

***static******void*** *methodThree()* ***throws*** *NumberFormatException{*

*methodTwo();*

*//NumberFormatException is propagated here*

*}*

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

***try****{*

*methodThree();*

*}****catch****(NumberFormatException ex){*

*System.out.println("NumberFormatException will be caught here");*

*}*

*}*

*}*

## Even constructor can use throws keyword.For this, object creation statement must be enclosed in try-catch blocks.

*class A{*

*int i;*

*public A(String s) throws NumberFormatException{*

*i = Integer.parseInt(s);*

*//This statement throws NumberFormatException*

*}*

*}*

*public class ExceptionHandling{*

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

*try{*

*A a = new A(“abc”);*

*//Object creation statement enclosed in try-catch block*

*}*

*catch (NumberFormatException ex){*

*System.out.println(“NumberFormatException will be caught here”);*

*}*

*}*

*}*

## Note:

## When a method is throwing unchecked type of exceptions, then you need not to mention it using throws keyword. But for a method throwing checked type of exceptions, you must declare it with throws keyword or enclose the statement which is throwing an exception in try-catch block.

*public class ExceptionHandling{*

*//method throwing Unchecked Exception declared without throws clause*

*static void methodThrowingUncheckedException(){*

*int i = Integer.parseInt(“abc”);*

*//Above statement throws NumberFormatException which is unchecked type of exception*

*}*

*//method throwing checked Exception declared with throws clause*

*static void methodThrowingCheckedException() throws ClassNotFoundException{*

*Class.forName(“AnyClassName”);*

*//Above statement throws ClassNotFoundException which is checked type of exception*

*}*

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

*try{*

*methodThrowingUncheckedException();*

*}*

*catch(NumberFormatException ex) {*

*System.out.println(“NumberFormatException will be caught here”);*

*}*

*try{*

*methodThrowingCheckedException();*

*}*

*catch (ClassNotFoundException e){*

*System.out.println(“ClassNotFoundException will be caught here”);*

*}  
 }*

*}*

# [Method Overriding With throws Clause](http://javaconceptoftheday.com/method-overriding-with-throws-clause/)

If super class method is not throwing any exceptions, then it can be overridden with any unchecked type of exceptions, but cannot be overridden with checked type of exceptions.

*class SuperClass{*

*void methodOfSuperClass(){*

*System.out.println("Super class method is not throwing any exceptions");*

*}*

*}*

*class SubClass extends SuperClass{*

*@Override*

*void methodOfSuperClass() throws ArrayIndexOutOfBoundsException{*

*System.out.println("can be overrided with any unchecked Exception");*

*}*

*}*

*class SubClassOne extends SuperClass{*

*@Override*

*void methodOfSuperClass() throws NumberFormatException, NullPointerException, RuntimeException {*

*System.out.println("Can be overrided with any number of Unchecked Exceptions");*

*}*

*}*

*class SubClassTwo extends SuperClass{*

*@Override*

*void methodOfSuperClass() throws SQLException*

*{*

***//Compile time error***

***//Cannot be overrided with checked exception***

*}*

*}*

## If a super class method is throwing unchecked exception, then it can be overridden in the sub class with same exception or any other unchecked exceptions but can not be overridden with checked exceptions.

*class SuperClass*

*{*

*void methodOfSuperClass() throws ArrayIndexOutOfBoundsException*

*{*

*System.out.println("Super class method is throwing Unchecked exception");*

*}*

*}*

*class SubClass extends SuperClass*

*{*

*@Override*

*void methodOfSuperClass() throws ArrayIndexOutOfBoundsException*

*{*

*System.out.println("Can be Overrided with same unchecked exception");*

*}*

*}*

*class SubClassOne extends SuperClass*

*{*

*@Override*

*void methodOfSuperClass() throws NumberFormatException, NullPointerException, RuntimeException*

*{*

*System.out.println("Can be overrided with any other Unchecked Exceptions");*

*}*

*}*

*class SubClassTwo extends SuperClass*

*{*

*@Override*

*void methodOfSuperClass() throws IOException*

*{*

*//Compile time error*

*//Can not be overrided with checked exception*

*}*

## *}*

## If super class method is throwing checked type of exception, then it can be overridden with same exception or with it’s sub class exceptions i.e you can decrease the scope of the exception, but cannot be overridden with it’s super class exceptions i.e you cannot increase the scope of the exception.

*class SuperClass{*

*void methodOfSuperClass() throws IOException*

*{*

*System.out.println("Super class method is throwing checked exception");*

*}*

*}*

*class SubClass extends SuperClass{*

*@Override*

*void methodOfSuperClass() throws IOException{*

*System.out.println("Can be Overrided with same checked exception");*

*}*

*}*

*class SubClassOne extends SuperClass{*

*@Override*

*void methodOfSuperClass() throws FileNotFoundException*

*{*

*System.out.println("Can be overrided with checked Exception with lesser scope");*

*}*

*}*

*class SubClassTwo extends SuperClass{*

*@Override*

*void methodOfSuperClass() throws NullPointerException, ArrayIndexOutOfBoundsException, FileNotFoundException*

*{*

*System.out.println("Can be overrided with any unchecked exceptions and checked exception with lesser scope");*

*}*

*}*

*class SubClassThree extends SuperClass{*

*@Override*

*void methodOfSuperClass() throws Exception*

*{*

*//Compile time error*

*//Can not be overrided with checked exception with higher scope*

*}*

*}*

## One more example with overriding a method throwing both checked and unchecked exceptions.

***class*** *SuperClass*

*{*

***void*** *methodOfSuperClass()* ***throws*** *IOException, ClassNotFoundException, NumberFormatException*

*{*

*System.out.println("Super class method is throwing both checked and unchecked exceptions");*

*}*

*}*

***class*** *SubClass* ***extends*** *SuperClass*

*{*

*@Override*

***void*** *methodOfSuperClass()* ***throws*** *IOException, ClassNotFoundException*

*{*

*System.out.println("Can be Overrided with same checked exceptions");*

*}*

*}*

***class*** *SubClassOne* ***extends*** *SuperClass*

*{*

*@Override*

***void*** *methodOfSuperClass()* ***throws*** *FileNotFoundException*

*{*

*System.out.println("Can be overrided with checked Exception with lesser scope");*

*}*

*}*

***class*** *SubClassTwo* ***extends*** *SuperClass*

*{*

*@Override*

***void*** *methodOfSuperClass()* ***throws*** *NullPointerException, ArrayIndexOutOfBoundsException, FileNotFoundException*

*{*

*System.out.println("Can be overrided with any other unchecked exceptions and checked exception with lesser scope");*

*}*

*}*

***class*** *SubClassThree* ***extends*** *SuperClass*

*{*

*@Override*

***void*** *methodOfSuperClass()* ***throws*** *Exception*

*{*

***//Compile time error***

***//Can not be overrided with checked exception with higher scope***

*}*

## *}*

## Java Exception propagation

## A. Exception Propagation : Unchecked Exception

***class*** *ExceptionPropagation{*

*void method3(){*

*int result = 100 / 0; //Exception Gere*

*}*

*void method2(){*

*method3();*

*}*

*void method1(){*

***try****{*

*method2();*

*}* ***catch****(****Exception*** *e){*

***System****.out.println("Exception is handled here");*

*}*

*}*

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

*ExceptionPropagation obj=****new*** *ExceptionPropagation();*

*obj.method1();*

***System****.out.println("Continue with Normal Flow...");*

*}*

*}*

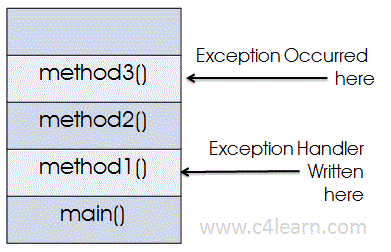
### Output :

*Exception is handled here*

*Continue with Normal Flow...*

## Explanation :

When exception is occurred at the top of the stack and no exception handler is provided then exception is propagated –



**We can see that**

1. exception is occurred in the method3() and in method3() we don’t have any exception handler.
2. Uncaught exception will be propagated downward in stack i.e it will check appropriate exception handler in the method2().
3. Again in method2 we don’t have any exception handler then again exception is propagated downward to method1() where it finds exception handler

Thus we can see that uncaught exception is propagated in the stack until stack becomes empty, this propagation of uncaught exception is called as **Exception Propagation**.

## B. Exception Propagation : Checked Exception

***class*** *ExceptionPropagation{*

*void method3(){*

***throw******new*** *java.io.****IOException****("Checked Exception..");*

*}*

*void method2(){*

*method3();*

*}*

*void method1(){*

***try****{*

*method2();*

*}* ***catch****(****Exception*** *e){*

***System****.out.println("Exception is handled here");*

*}*

*}*

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

*ExceptionPropagation obj=****new*** *ExceptionPropagation();*

*obj.method1();*

***System****.out.println("Continue with Normal Flow...");*

*}*

*}*

### Output :

Compile Time Error

You must remember one rule of thumb that – “**Checked Exceptions are not propagated in the chain**“. Thus we will get compile error in the above case

#### Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

***Program of Exception Propagation***

***class*** *TestExceptionPropagation1{*

***void*** *m(){*

***int*** *data=50/0;*

*}*

***void*** *n(){*

*m();*

*}*

***void*** *p(){*

***try****{*

*n();*

*}****catch****(Exception e){*

*System.out.println("exception handled");*

*}*

*}*

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

*TestExceptionPropagation1 obj=****new*** *TestExceptionPropagation1();*

*obj.p();*

*System.out.println("normal flow...");*

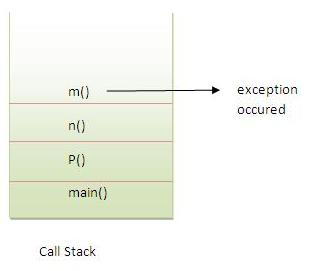
*}*

*}*

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestExceptionPropagation1)

**Output**: exception handled

normal flow...



In the above example, exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

#### Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

***Program which describes that checked exceptions are not propagated***

*class TestExceptionPropagation2{*

*void m(){*

*throw new java.io.IOException("device error");//checked exception*

*}*

*void n(){*

*m();*

*}*

*void p(){*

*try{*

*n();*

*}catch(Exception e){System.out.println("exception handeled");}*

*}*

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

*TestExceptionPropagation2 obj=new TestExceptionPropagation2();*

*obj.p();*

*System.out.println("normal flow");*

*}*

*}*

**Output**:

Compile Time Error: **Unhandled exception type IOException**

## Exception classes

Exception are mainly classified into two type checked exception and un-checked exception.

## Checked Exception Classes

* FileNotFoundException
* ClassNotFoundException
* IOException
* InterruptedException

## Un-Checked Exception Classes

* ArithmeticException
* ArrayIndexOutOfBoundsException
* StringIndexOutOfBoundsException
* NumberFormateException
* NullPointerException
* NoSuchMethodException
* NoSuchFieldException

### FileNotFoundException

If the given filename is not available in a specific location ( in file handling concept), then FileNotFoundException will be raised.

This exception will be thrown by the FileInputStream, FileOutputStream, and RandomAccessFile constructors.

### ClassNotFoundException

If the given class name is not existing at the time of compilation or running of program then ClassNotFoundException will be raised.

In other words this exception is occured when an application tries to load a class but no definition for the specified class name could be found.

### IOException

This is exception is raised whenever problem occurred while writing and reading the data in the file.

This exception is occurred due to following reason;

* When try to transfer more data but less data are present.
* When try to read data which is corrupted.
* When try to write on file but file is read only.

### InterruptedException

This exception is raised whenever one thread is disturb the other thread.

In other words this exception is thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity.

### ArithmeticException

This exception is raised because of problem in arithmetic operation like divide by zero.

In other words this exception is thrown when an exceptional arithmetic condition has occurred.

For example, an integer "divide by zero".

## Example

*class ExceptionDemo {*

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

*int a=10, ans=0;*

*try{*

*ans=a/0;*

*}*

*catch (Exception e)*

*{*

*System.out.println("Denominator not be zero");*

*}*

*}*

*}*

### ArrayIndexOutOfBoundsException

This exception will be raised whenever given index value of an array is out of range.

The index is either negative or greater than or equal to the size of the array.

## Example

int a[]=new int[5];

a[10]=100; //ArrayIndexOutOfBoundsException

### StringIndexOutOfBoundsException

This exception will be raised whenever given index value of string is out of range.

The index is either negative or greater than or equal to the size of the array.

## Example

*String s="Hello";*

*s.charAt(3);*

*s.charAt(10); //* ***Exception raised***

chatAt() is a predefined method of string class used to get the individual characters based on index value.

### NumberFormateException

This exception will be raised whenever you trying to store any input value in the un-authorized datatype.

**Example**:

Storing string value into int datatype.

## Example

int a;

a="Hello";

## Example

String s="hello";

int i=Integer.parseInt(s);//NumberFormatException

### NoSuchMethodException

This exception will be raised whenever calling method is not existing in the program.

### NullPointerException

A NullPointerException is thrown when an application is trying to use or access an object whose reference equals to null.

## Example

String s=null;

System.out.println(s.length());//NullPointerException

### StackOverFlowException

This exception throw when full the stack because the recursion method are stored in stack area.

## throw

throw is a keyword in java language which is used to throw any user defined exception to the same signature of method in which the exception is raised.

**Note:**

throw keyword always should exist within method body.

Whenever method body contains throw keyword than the call method should be followed by throws keyword.

## Syntax

*class className{*

*returntype method(...)* ***throws*** *Exception\_class{*

***throw****(Exception obj)*

*}*

*}*

## throws

throws is a keyword in java language which is used to throw the exception which is raised in the called method to it's calling method. Throws keyword always followed by method signature.

## Example

*returnType methodName(parameter)throws Exception\_class....*

*{*

*.....*

*}*

### Difference between throw and throws

|  |  |
| --- | --- |
| **throw** | **throws** |
| throw is a keyword used for hitting and generating the exception which are occurring as a part of method body | throws is a keyword which gives an indication to the specific method to place the common exception methods as a part of try and catch block for generating user friendly error messages |
| The place of using throw keyword is always as a part of method body. | The place of using throws is a keyword is always as a part of method heading |
| When we use throw keyword as a part of method body, it is mandatory to the java programmer to write throws keyword as a part of method heading | When we write throws keyword as a part of method heading, it is optional to the java programmer to write throw keyword as a part of method body. |

|  |
| --- |
|  |
|  |
|  |
|  |

### Example of throw and throws

**// Save by DivZero.java**

*package pack;*

*public class DivZero{*

*public void division(int a, int b)throws ArithmeticException{*

*if(b==0){*

*ArithmeticException ae=new ArithmeticException("Does not enter zero for Denominator");*

*throw ae;*

*}else*

*{*

*int c=a/b;*

*System.out.println("Result: "+c);*

*}*

*}*

*}*

## Compile: javac -d . DivZero.java

## Example

**// Save by ArthException.java**

*import pack.DivZero;*

*import java.util.\*;*

*class ArthException{*

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

*{*

*System.out.println("Enter any two number: ");*

*Scanner s=new Scanner(System.in);*

*try*

*{*

*int a=s.nextInt();*

*int b=s.nextInt();*

*DivZero dz=new DivZero();*

*dz.division(a, b);*

*}*

*catch(Exception e)*

*{*

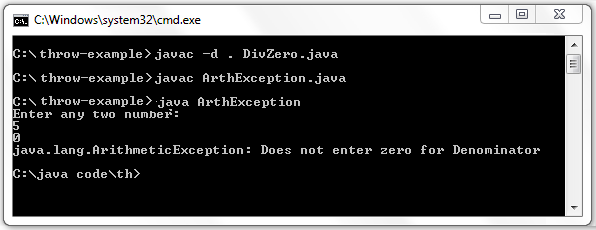
*System.err.println(e);*

*}*

*}*

*}*

## Compile: javac ArthException.java



### Steps to Compile and Run code

* javac -d . DivZero.java
* javac ArthException.java

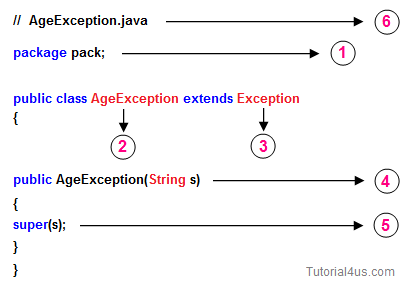
**Note:** First compile DivZero.java code then compile ArthException.java code.

## Custom Exception

If any exception is designed by the user, then it is known as user defined or Custom Exception.

### Rules to design user defined Exception

1. Create a package with valid user defined name.
2. Create any user defined class.
3. Make that user defined class as derived class of Exception or RuntimeException class.
4. Declare parametrized constructor with string variable.
5. call super class constructor by passing string variable within the derived class constructor.
6. Save the program with public class name.java



## Example

// Save by AgeException.java

*package nage;*

*public class AgeException extends Exception{*

*public AgeException(String s) {*

*super(s);*

*}*

*}*

## Compile: javac -d . AgeException.java

## Example

// **Save by CheckAge.java**

*package nage;*

*public class CheckAge{*

*public void verify(int age)throws AgeException{*

*if (age>0)*

*{*

*System.err.print("valid age");*

*}*

*else*

*{*

*AgeException ae=new AgeException("Invalid age");*

*throw(ae);*

*}*

*}*

*}*

## Compile: javac -d . CheckAge.java

## Example

// **Save by VerifyAgeException**

*import nage.AgeException;*

*import nage.CheckAge;*

*import java.util.\*;*

*public class VerifyAgeException{*

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

*int a;*

*System.out.println("Enter your age");*

*Scanner s=new Scanner(System.in);*

*a=s.nextInt();*

*try{*

*CheckAge ca=new CheckAge();*

*ca.verify(a);*

*}*

*catch(AgeException ae){*

*System.err.println("Age should not be -ve");*

*}*

*catch(Exception e){*

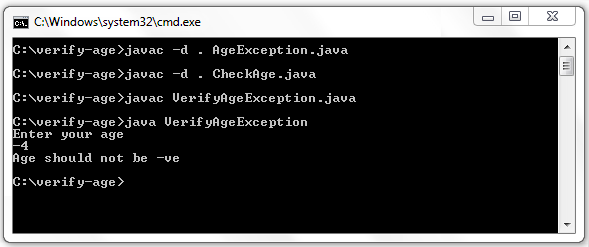
*System.err.println(e);*

*}*

*}*

*}*

## Compile: javac VerifyAgeException.java



### Steps to compile and run above program

First you save verify-age files into you PC in any where, here i will save this file in C:\>

* javac -d . AgeException.java
* javac -d . CheckAge.java
* javac VerifyAgeException.java

**Note:** First compile AgeException.java code then CheckAge.java and at last compile VerifyAgeException.java code.

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 predefined **Exception** class to create your own Exception. 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 the 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)

## Common Exceptions:

In Java, it is possible to define two catergories of Exceptions and Errors.

* **JVM Exceptions:** - These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples : NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException,
* **Programmatic exceptions:** - These exceptions are thrown explicitly by the application or the API programmers Examples: IllegalArgumentException, IllegalStateException.

## Example of User defined exception in Java

*class MyException extends Exception{*

*String str1;*

*MyException(String str2) {*

*str1=str2;*

*}*

*public String toString(){*

*return ("Output String = "+str1) ;*

*}*

*}*

*class CustomException{*

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

*try{*

*throw new MyException("Custom");*

*// I'm throwing user defined custom exception above*

*}*

*catch(MyException exp){*

*System.out.println("Hi this is my catch block") ;*

*System.out.println(exp) ;*

*}*

*}*

*}*

**Output:**

Hi this is my catch block

Output String = Custom

**Key-points from above example**:  
You can see that while throwing my custom exception I gave a string in parenthesis ( throw new MyException("Custom");). That’s the reason we have a parametric constructor (with a String parameter) in my custom exception class.

**Notes:**

* User defined exception needs to [inherit (extends)](http://beginnersbook.com/2013/03/inheritance-in-java/) Exception class in order to [act as an exception](http://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/).
* throw keyword is used to throw such exceptions.

### Another example wherein we will modify the error message of Exception Class

**Part 1**:

I have created my own exception class- **MyException** by inheriting the parent class **Exception** then I have [defined a parametric constructor](http://beginnersbook.com/2013/03/constructors-in-java/) of my class with a String parameter. In the constructor I called super(), super refers to the super class { My class has inherited Exception class so Exception class is my superclass}.

In this way I have modified the system generated message by my own message.

*public class MyException extends Exception*

*{*

*public MyException(String mymsg)*

*{*

*super(mymsg);*

*}*

*}*

**Part2**:

*public class ExceptionSample{*

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

*ExceptionSample es = new ExceptionSample();*

*es.displayMymsg();*

*}*

*public void displayMymsg() throws MyException{*

*for(int j=8;j>0;j--){*

*System.out.println("j= "+j);*

*if(j==7){*

*throw new MyException("This is my own Custom Message");*

*}*

*}*

*}*

*}*

**Output:**

j = 8

j = 7

Exception in thread "main" MyException: This is my own Custom Message

at ExceptionSample.displayMymsg( ExceptionSample.java.19)

...

## Throw vs Throws in java

1. **Throws clause** in used to declare an exception and **thow** keyword is used to throw an exception explicitly.

2. If we see syntax wise than **throw** is followed by an instance variable and **throws** is followed by exception class names.

3. The keyword **throw** is used inside method body to invoke an exception and **throws clause** is used in method declaration (signature).

**for e.g.**

**Throw:**

....

static{

try {

throw new Exception("Something went wrong!!");

} catch (Exception exp) {

System.out.println("Error: "+exp.getMessage());

}

}

....

**Throws**:

public void sample() throws ArithmeticException{

//Statements

.....

//if (Condition : There is an error)

ArithmeticException exp = new ArithmeticException();

throw exp;

...

}

4. By using **Throw keyword** in java you cannot throw more than one exception but using **throws** you can declare multiple exceptions. PFB the examples.

**Example**:

**Throw:**

throw new ArithmeticException("An integer should not be divided by zero!!")

throw new IOException("Connection failed!!")

**Throws:**

throws IOException, ArithmeticException, NullPointerException,

ArrayIndexOutOfBoundsException

# How Exception Terminates Java Program?

Whenever exception arises, it terminates the program execution, means it stops the execution of the current java program. Below example shows what happens when an exception occurs.

|  |
| --- |
| **Code:** |
| *package com.myjava.exceptions;*  *public class MyException {*  *public static void main(String a[]){*  *for(int i=5;i>=0;i--){*  *System.out.println(10/i);*  *}*  *System.out.println("After for loop...");*  *}*  *}* |

#### Example Output

2

2

3

5

10

Exception in thread "main" java.lang.ArithmeticException: / by zero

at com.myjava.exceptions.MyException.main(MyException.java:6)

Java Result: 1

### Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur.

They are as follows:

### 1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

**Example:** int a=50/0;//ArithmeticException

### 2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

*String s=null;*

*System.out.println(s.length());//NullPointerException*

### 3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

*String s="abc";*

*int i=Integer.parseInt(s);//NumberFormatException*

### 4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

*int a[]=new int[5];*

*a[10]=50; //ArrayIndexOutOfBoundsException*

# Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  |  | | --- | --- | --- | --- | | **No.** | **final** | **finally** | **finalize** | | 1) | Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. | | 2) | Final is a keyword. | Finally is a block. | Finalize is a method. | | |

## Java final example

*class FinalExample{*

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

*final int x=100;*

*x=200;//Compile Time Error*

*}*

*}*

## Java finally example

**c*lass*** *FinallyExample{*

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

***try****{*

***int*** *x=300;*

*}****catch****(Exception e){System.out.println(e);}*

***finally****{System.out.println("finally block is executed");}*

*}*

*}*

## Java finalize example

***class*** *FinalizeExample{*

***public******void*** *finalize(){System.out.println("finalize called");}*

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

*FinalizeExample f1=****new*** *FinalizeExample();*

*FinalizeExample f2=****new*** *FinalizeExample();*

*f1=****null****;*

*f2=****null****;*

*System.gc();*

*}*

*}*

# ExceptionHandling with MethodOverriding in Java

|  |
| --- |
| There are many rules, if we talk about methodoverriding with exception handling.  The Rules are as follows:   * **If the superclass method does not declare an exception**   + If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception. * **If the superclass method declares an exception**   + If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception. |

### If the superclass method does not declare an exception

#### 1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

***import*** *java.io.\*;*

***class*** *Parent{*

***void*** *msg(){System.out.println("parent");}*

*}*

***class*** *TestExceptionChild* ***extends*** *Parent{*

***void*** *msg()****throws*** *IOException{*

*System.out.println("TestExceptionChild");*

*}*

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

*Parent p=****new*** *TestExceptionChild();*

*p.msg();*

*}*

*}*

**Output**:

Compile Time Error: Exception IOException is not compatible with throws clause in Parent.msg()

#### 2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

***import*** *java.io.\*;*

***class*** *Parent{*

***void*** *msg(){System.out.println("parent");}*

*}*

***class*** *TestExceptionChild1* ***extends*** *Parent{*

***void*** *msg()****throws*** *ArithmeticException{*

*System.out.println("child");*

*}*

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

*Parent p=****new*** *TestExceptionChild1();*

*p.msg();*

*}*

*}*

**Output**:

child

### If the superclass method declares an exception

#### 1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

### Example in case subclass overridden method declares parent exception

***import*** *java.io.\*;*

***class*** *Parent{*

***void*** *msg()****throws*** *ArithmeticException{System.out.println("parent");}*

*}*

***class*** *TestExceptionChild2* ***extends*** *Parent{*

***void*** *msg()****throws*** *Exception{System.out.println("child");}*

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

*Parent p=****new*** *TestExceptionChild2();*

***try****{*

*p.msg();*

*}****catch****(Exception e){}*

*}*

*}*

**Output**:

Compile Time Error: Exception Exception is not compatible with throws clause in Parent.msg()

### Example in case subclass overridden method declares same exception

***import*** *java.io.\*;*

***class*** *Parent{*

***void*** *msg()****throws*** *Exception{System.out.println("parent");}*

*}*

***class*** *TestExceptionChild3* ***extends*** *Parent{*

***void*** *msg()****throws*** *Exception{System.out.println("child");}*

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

*Parent p=****new*** *TestExceptionChild3();*

***try****{*

*p.msg();*

*}****catch****(Exception e){}*

*}*

*}*

**Output**:

child

### Example in case subclass overridden method declares subclass exception

***import*** *java.io.\*;*

***class*** *Parent{*

***void*** *msg()****throws*** *Exception{System.out.println("parent");}*

*}*

***class*** *TestExceptionChild4* ***extends*** *Parent{*

***void*** *msg()****throws*** *ArithmeticException{System.out.println("child");}*

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

*Parent p=****new*** *TestExceptionChild4();*

***try****{*

*p.msg();*

*}****catch****(Exception e){}*

*}*

*}*

**Output**:

child

### Example in case subclass overridden method declares no exception

***import*** *java.io.\*;*

***class*** *Parent{*

***void*** *msg()****throws*** *Exception{*

*System.out.println("parent");*

*}*

*}*

***class*** *TestExceptionChild5* ***extends*** *Parent{*

***void*** *msg(){*

*System.out.println("child");*

*}*

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

*Parent p=****new*** *TestExceptionChild5();*

***try****{*

*p.msg();*

*}****catch****(Exception e){}*

*}*

*}*

**Output**:

child

**Exceptions Methods:**

Following is the list of important methods available in the Throwable class.

|  |  |
| --- | --- |
|  | **Methods with Description** |
|  | **public String getMessage()**  Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. |
|  | **public Throwable getCause()**  Returns the cause of the exception as represented by a Throwable object. |
|  | **public String toString()**  Returns the name of the class concatenated with the result of getMessage() |
|  | **public void printStackTrace()**  Prints the result of toString() along with the stack trace to System.err, the error output stream. |
|  | **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. |
|  | **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. |

## The try-with-resources

Generally when we use any resources like streams, connections etc..

We have to close them explicitly using finally block.

In the program given below we are reading data from a file using **FileReader** and we are closing it using finally block.

*import java.io.File;*

*import java.io.FileReader;*

*import java.io.IOException;*

*public class ReadData\_Demo {*

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

*FileReader fr=null;*

*try{*

*File file=new File("file.txt");*

*fr = new FileReader(file); char [] a = new char[50];*

*fr.read(a); // reads the content to the array*

*for(char c : a)*

*System.out.print(c); //prints the characters one by one*

*}catch(IOException e){*

*e.printStackTrace();*

*}*

*finally{*

*try{*

*fr.close();*

*}catch(IOException ex){*

*ex.printStackTrace();*

*}*

*}*

*}*

*}*

**try-with-resources**, also referred as **automatic resource management**. is a new exception handling mechanism that was introduced in Java7, which automatically closes the resources used within the try catch block.

To use this statement you simply need to declare the required resources within the parenthesis, the created resource will be closed automatically at the end of the block, below given is the syntax of try-with-resources statement.

*try(FileReader fr=new FileReader("file path"))*

*{*

*//use the resource*

*}catch(){*

*//body of catch*

*}*

*}*

Below given is the program that reads the data in a file using try-with-resources statement.

*import java.io.FileReader;*

*import java.io.IOException;*

*public class Try\_withDemo {*

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

*try(FileReader fr=new FileReader("E://file.txt")){*

*char [] a = new char[50];*

*fr.read(a); // reads the contentto the array*

*for(char c : a)*

*System.out.print(c); //prints the characters one by one*

*}catch(IOException e){*

*e.printStackTrace();*

*}*

*}*

*}*

Following points are to be kept in mind while working with try-with resources statement.

* To use a class with try-with-resources statement it should implement **AutoCloseable** interface and the **close()** method of it gets invoked automatically at runtime.
* You can declare more than one class in try-with-resources statement.
* while you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
* Except the deceleration of resources within the parenthesis everything is same as normal try/catch block of a try block.
* The resource declared in try gets instantiated just before the start of the try-block.
* The resource declared at the try block is implicitly declared as final.

# [Chained Exceptions In Java](http://javaconceptoftheday.com/chained-exceptions-java/)

In an application, one exception throws many exceptions. i.e one exception causes another exception and that exception causes another exception thus forming chain of exceptions. It is better to know where the actual cause of the exception lies. This is possible with chained exceptions feature of the Java.

**Chained exceptions are introduced from JDK 1.4.** To implement chained exceptions in java, two new constructors and two new methods are added in the Throwable class. They are,

## Constructors Of Throwable class Which support chained exceptions in java :

**1) Throwable(Throwable cause)**  —-> where cause is the exception that causes the current exception.

**2) Throwable(String msg, Throwable cause)** —-> where msg is the exception message and cause is the exception that causes the current exception.

## Methods Of Throwable class Which support chained exceptions in java :

1) **getCause() method** : This method returns actual cause of an exception.

2) **initCause(Throwable cause) method** : This method sets the cause for the calling exception.

Let’s see one example for how to set and get the actual cause of an exception.

**public** **class** ExceptionHandling

{

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

{

**try**

{

//creating an exception

NumberFormatException ex = **new** NumberFormatException("Exception");

**//setting a cause of the exception**

ex.initCause(**new** NullPointerException("This is actual cause of the exception"));

**throw** ex;

}

**catch**(NumberFormatException ex)

{

System.*out*.println(ex); //**displaying the exception**

System.*out*.println(ex.getCause()); //**getting the actual cause of the exception**

}

}

}

## What is the difference between throw, throws and Throwable in java?

## throw In Java :

**throw** is a keyword in java which is used to throw an exception manually. Using throw keyword, you can throw an exception from any method or block. But, that exception must be of type **java.lang.Throwable** class or it’s sub classes. Below example shows how to throw an exception using throw keyword.

*class ThrowAndThrowsExample*

*{*

*void method() throws Exception*

*{*

*Exception e = new Exception();*

*throw e;*

*//throwing an exception using 'throw'*

*}*

*}*

## throws In Java :

**throws** is also a keyword in java which is used in the method signature to indicate that this method may throw mentioned exceptions. The caller to such methods must handle the mentioned exceptions either using try-catch blocks or using throws keyword. Below is the syntax for using throws keyword.

*return\_type method\_name(parameter\_list) throws exception\_list*

*{*

*//some statements*

*}*

Below is the example which shows how to use throws keyword.

*class ThrowsExample{*

*void methodOne() throws SQLException{*

*//This method may throw SQLException*

*}*

*void methodTwo() throws IOException{*

*//This method may throw IOException*

*}*

*void methodThree() throws ClassNotFoundException{*

*//This method may throw ClassNotFoundException*

*}*

*}*

## Throwable In Java :

**Throwable** is a super class for all types of errors and exceptions in java. This class is a member of **java.lang** package. Only instances of this class or it’s sub classes are thrown by the java virtual machine or by the throw statement. The only argument of catch block must be of this type or it’s sub classes. If you want to create your own customized exceptions, then your class must extend this class. Click [here](http://javaconceptoftheday.com/hierarchy-exceptions-java/) to see the hierarchy of exception classes in java.

Below example shows how to create customized exceptions by extending **java.lang.Throwable** class.

*class MyException extends Throwable{*

*//Customized Exception class*

*}*

*class ThrowAndThrowsExample{*

*void method() throws MyException{*

*MyException e = new MyException();*

*throw e;*

*}*

*}*

# [Difference Between ClassNotFoundException Vs NoClassDefFoundError In Java](http://javaconceptoftheday.com/classnotfoundexception-vs-noclassdeffounderror-in-java/)

In Java, both **ClassNotFoundException** and **NoClassDefFoundError** occur when a particular class is not found at run time. But, they occur at different scenarios. **ClassNotFoundException** is an exception which occurs when you try to load a class at run time using **Class.forName()** or **loadClass()** methods and mentioned classes are not found in the classpath. On the other hand, **NoClassDefFoundError** is an error which occurs when a particular class is present at compile time but it was missing at run time. In this tutorial, we will see the differences between ClassNotFoundException Vs NoClassDefFoundError in java and when they occur.

## ClassNotFoundException In Java :

ClassNotFoundException is a run time exception which is thrown when an application tries to load a class at run time using **Class.forName()** or **loadClass()** or **findSystemClass()** methods and the class with specified name are not found in the classpath. For example, you may have come across this exception when you try to connect to MySQL or Oracle databases and you have not updated the classpath with required JAR files. In most of time, this exception occurs when you try to run an application without updating the classpath with required JAR files.

For example, below program will throw ClassNotFoundException if the mentioned class **“oracle.jdbc.driver.OracleDriver”** is not found in the classpath.

*public class MainClass{*

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

*try{*

*Class.forName("oracle.jdbc.driver.OracleDriver");*

*}*

*catch (ClassNotFoundException e){*

*e.printStackTrace();*

*}*

*}*

*}*

If you run the above program without updating the classpath with required JAR files, you will get the exception like below,

***java.lang.ClassNotFoundException****: oracle.jdbc.driver.OracleDriver*

*at java.net.URLClassLoader.findClass(Unknown Source)*

*at java.lang.ClassLoader.loadClass(Unknown Source)*

*at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)*

*at java.lang.ClassLoader.loadClass(Unknown Source)*

*at java.lang.Class.forName0(Native Method)*

*at java.lang.Class.forName(Unknown Source)*

*at pack1.MainClass.main(MainClass.java:17)*

## NoClassDefFoundError In Java :

NoClassDefFoundError is an error which is thrown when Java Runtime System tries to load the definition of a class and class definition is no longer available. The required class definition was present at compile time but it was missing at run time. For example, compile the below program.

*class A{*

*}*

*public class B{*

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

*{*

*A a = new A();*

*}*

*}*

When you compile the above program, two .class files will be generated. One is **A.class** and another one is **B.class**. If you remove the **A.class** file and run the **B.class** file, Java Runtime System will throw NoClassDefFoundError like below,

*Exception in thread "main"* ***java.lang.NoClassDefFoundError****: A*

*at MainClass.main(MainClass.java:10)*

*Caused by: java.lang.ClassNotFoundException: A*

*at java.net.URLClassLoader.findClass(URLClassLoader.java:381)*

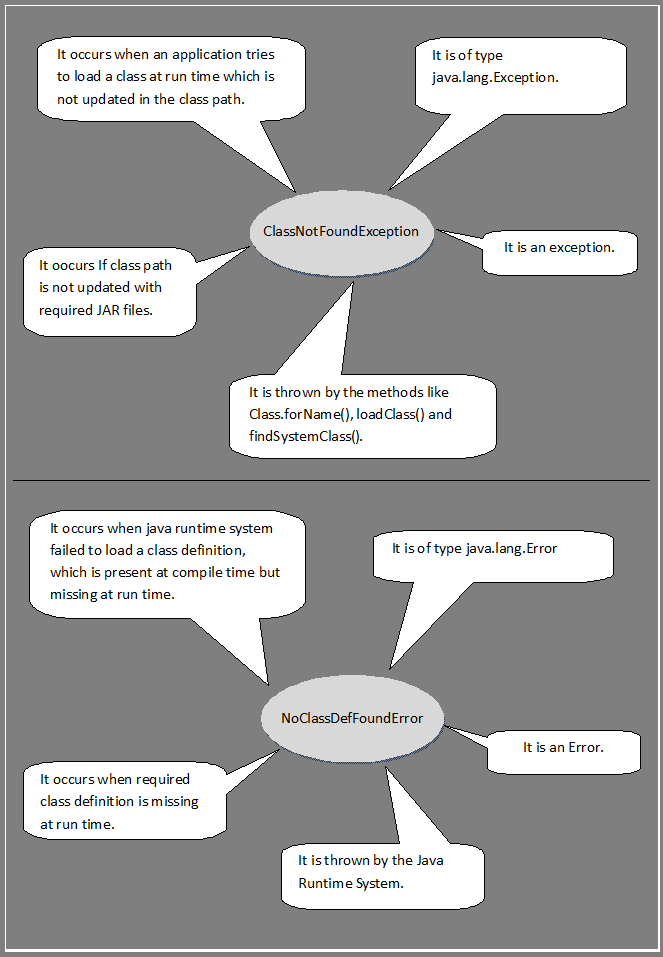
*at java.lang.ClassLoader.loadClass(ClassLoader.java:424)*

*at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)*

*at java.lang.ClassLoader.loadClass(ClassLoader.java:357)*

**Difference Between ClassNotFoundException Vs NoClassDefFoundError In Java :**

|  |  |
| --- | --- |
| **ClassNotFoundException** | **NoClassDefFoundError** |
| It is an exception. It is of type java.lang.Exception. | It is an error. It is of type java.lang.Error. |
| It occurs when an application tries to load a class at run time which is not updated in the classpath. | It occurs when java runtime system doesn’t find a class definition, which is present at compile time, but missing at run time. |
| It is thrown by the application itself. It is thrown by the methods like Class.forName(), loadClass() and findSystemClass(). | It is thrown by the Java Runtime System. |
| It occurs when classpath is not updated with required JAR files. | It occurs when required class definition is missing at run time. |



## Java Exception Handling Interview Questions And Answers

**1) What is an exception?**

Exception is an abnormal condition which occurs during the execution of a program and disrupts normal flow of the program. This exception must be handled properly. If it is not handled, program will be terminated abruptly.

**2) How the exceptions are handled in java? OR Explain exception handling mechanism in java?**

Exceptions in java are handled using try, catch and finally blocks.

**try block** : The code or set of statements which are to be monitored for exception are kept in this block.

**catch block** : This block catches the exceptions occurred in the try block.

**finally block** : This block is always executed whether exception is occurred in the try block or not and occurred exception is caught in the catch block or not.

**3) What is the difference between error and exception in java?**

Errors are mainly caused by the environment in which an application is running. For example, OutOfMemoryError happens when JVM runs out of memory. Where as exceptions are mainly caused by the application itself. For example, NullPointerException occurs when an application tries to access null object.

**4) Can we keep other statements in between try, catch and finally blocks?**

No. We shouldn’t write any other statements in between try, catch and finally blocks. They form a one unit.

*try*

*{*

*// Statements to be monitored for exceptions*

*}*

***//You can't keep statements here***

*catch(Exception ex)*

*{*

***//Cathcing the exceptions here***

*}*

*//You can't keep statements here*

*finally*

*{*

***// This block is always executed***

*}*

**5) Can we write only try block without catch and finally blocks?**

No, It shows compilation error. The try block must be followed by either catch or finally block. You can remove either catch block or finally block but not both.

**6) There are three statements in a try block – statement1, statement2 and statement3. After that there is a catch block to catch the exceptions occurred in the try block. Assume that exception has occurred in statement2. Does statement3 get executed or not?**

No. Once a try block throws an exception, remaining statements will not be executed. control comes directly to catch block.

**7) What is unreachable catch block error?**

When you are keeping multiple catch blocks, the order of catch blocks must be from most specific to most general ones. i.e sub classes of Exception must come first and super classes later. If you keep super classes first and sub classes later, compiler will show unreachable catch block error.

*public class ExceptionHandling{*

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

*try {*

*int i = Integer.parseInt("abc");*

***//This statement throws NumberFormatException***

*}*

*catch(Exception ex)*

*{*

*System.out.println("This block handles all exception types");*

*}*

*catch(NumberFormatException ex)*

*{*

***//Compile time error***

***//This block becomes unreachable as***

***//exception is already caught by above catch block***

*}*

*}*

*}*

8) **What are run time exceptions in java. Give example?**

The exceptions which occur at run time are called as run time exceptions. These exceptions are unknown to compiler. All sub classes of java.lang.RunTimeException and java.lang.Error are run time exceptions. These exceptions are unchecked type of exceptions. For example, NumberFormatException, NullPointerException, ClassCastException, ArrayIndexOutOfBoundException, StackOverflowError etc.

**9) What is OutOfMemoryError in java?**

OutOfMemoryError is the sub class of java.lang.Error which occurs when JVM runs out of memory.

**10) What are checked and unchecked exceptions in java?**

Checked exceptions are the exceptions which are known to compiler. These exceptions are checked at compile time only. Hence the name checked exceptions. These exceptions are also called compile time exceptions. Because, these exceptions will be known during compile time.

Unchecked exceptions are those exceptions which are not at all known to compiler. These exceptions occur only at run time. These exceptions are also called as run time exceptions. All sub classes of java.lang.RunTimeException and java.lang.Error are unchecked exceptions.

**Can we keep the statements after finally block If the control is returning from the finally block itself?**

No, it gives unreachable code error. Because, control is returning from the finally block itself. Compiler will not see the statements after it. That’s why it shows unreachable code error.

**Does finally block get executed If either try or catch blocks are returning the control?**

Yes, finally block will be always executed no matter whether try or catch blocks are returning the control or not.

**Can we throw an exception manually? If yes, how?**

Yes, we can throw an exception manually using throw keyword. Syntax for throwing an exception manually is

**throw InstanceOfThrowableType;**

Below example shows how to use throw keyword to throw an exception manually.

*try{*

*NumberFormatException ex = new NumberFormatException();    //Creating an object to NumberFormatException explicitly*

*throw ex;*

***//throwing NumberFormatException object explicitly using throw keyword***

***}***

*catch(NumberFormatException ex){*

*System.out.println("explicitly thrown NumberFormatException object will be caught here");*

*}*

**What is Re-throwing an exception in java?**

Exceptions raised in the try block are handled in the catch block. If it is unable to handle that exception, it can re-throw that exception using throw keyword. It is called re-throwing an exception.

*try*

*{*

*String s = null;*

*System.out.println(s.length());*

***//This statement throws NullPointerException***

*}*

*catch(NullPointerException ex)*

*{*

*System.out.println("NullPointerException is caught here");*

*throw ex;* ***//Re-throwing NullPointerException***

*}*

**Why it is always recommended that clean up operations like closing the DB resources to keep inside a finally block?**

Because finally block is always executed whether exceptions are raised in the try block or not and raised exceptions are caught in the catch block or not. By keeping the clean up operations in finally block, you will ensure that those operations will be always executed irrespective of whether exception is occurred or not.

**What is ClassCastException in java?**

ClassCastException is a RunTimeException which occurs when JVM unable to cast an object of one type to another type.

**Example:**

In Java application, when we are trying to copy a super class object reference value in sub-class reference variable, then JVM will raise Class Cast Exception.

*Class A{*

*}*

*Class B extends A{*

*}*

*Class Test{*

*P.S.V.M(){*

*A a=new A();*

*B b=(B)a;*

*}*

*}*

If we execute the above code, the JVM will raise an exception with the following details:

**Exception Name:** java.lang.ClassCastException

**Exception Description:** A cannot be casted to B.

**Exception Location: Test.java**

**Note:**

* + 1. In Java applications, it is possible to assign subclass reference variable to superclass reference variable directly.
    2. In Java applications, it is not possible to assign super class reference variable to sub class reference variable directly. Compiler will raise incompatible type error. If you want to assign super class reference variable to sub class reference variable, then we must use cast operator.
    3. In Java application, is is possible to keep subclass object reference value into superclass reference variable directly.
    4. In Java applications, it is not possible to keep super class object reference value in subclass reference variable, then JVM raise ClassCastException.

**What is StackOverflowError in java?**

StackOverflowError is an error which is thrown by the JVM when stack overflows.

**Can we override a super class method which is throwing an unchecked exception with checked exception in the sub class?**

No. If a super class method is throwing an unchecked exception, then it can be overridden in the sub class with same exception or any other unchecked exceptions but can not be overridden with checked exceptions.

**Which class is the super class for all types of errors and exceptions in java?**

java.lang.Throwable is the super class for all types of errors and exceptions in java.

**What are the legal combinations of try, catch and finally blocks?**

1)

*try*

*{*

*//try block*

*}*

*catch(Exception ex)*

*{*

*//catch block*

*}*

2)

*try*

*{*

*//try block*

*}*

*finally*

*{*

*//finally block*

*}*

3)

*try*

*{*

*//try block*

*}*

*catch(Exception ex)*

*{*

*//catch block*

*}*

*finally*

*{*

*//finally block*

*}*

**What is the use of printStackTrace() method?**

printStackTrace() method is used to print the detailed information about the exception occurred.

**Give some examples to checked exceptions?**

ClassNotFoundException, SQLException, IOException

**Give some examples to unchecked exceptions?**

NullPointerException, ArrayIndexOutOfBoundsException, NumberFormatException

### Most Frequently Occurring Exceptions In Java With Examples :

**java.lang.NullPointerException**

NullPointerException is a RunTimeException which occurs when your application tries to access null object. It happens if you don’t initialize the reference variable and it is pointing to null instead of actual object. Using such reference variable will cause NullpointerException.

*package pack1;*

*class A*

*{*

*static String s;*

*}*

*public class MainClass*

*{*

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

*{*

*System.out.println(A.s.length());*

*}*

*}*

In the above code, we are trying to find the length of string filed **‘s’** of class **A**. But, it is not initialized. It is pointing to null. This will throw java.lang.NullPointerException like below.

*Exception in thread "main" java.lang.NullPointerException*

*at pack1.MainClass.main(MainClass.java:12)*

**java.lang.ArrayIndexOutOfBoundsException**

ArrayIndexOutOfBoundsException is also one of the frequently occurring exception in java. It occurs when you try to access an array element with an invalid index i.e index greater than the array length or with a negative index.

*package pack1;*

*public class MainClass{*

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

*String s[] = new String[] {"ZERO", "ONE", "TWO", "THREE", "FOUR"};*

*System.out.println(s[5]);*

***//This will throw ArrayIndexOutOfBoundsException***

*System.out.println(s[-1****]);   //This will also throw*** *ArrayIndexOutOfBoundsException*

*}*

*}*

In the above example, string array **‘s’** contains only 5 elements. But, we are trying to access sixth element. This will result in java.lang.ArrayIndexOutOfBoundsException.

*Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5*

*at pack1.MainClass.main(MainClass.java:9)*

**java.lang.NumberFormatException**

NumberFormatException is thrown when you are trying to convert a string to numeric value like integer, float, double etc…, but input string is not a valid number. NumberFormatException is also one of IllegalArgumentException.

*package pack1;*

*public class MainClass{*

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

*int i = Integer.parseInt("abc");*

*}*

*}*

In the above code, we are trying to convert a string **“abc”** to number, which is not possible. This will throw java.lang.NumberFormatException.

*Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"*

*at java.lang.NumberFormatException.forInputString(Unknown Source)*

*at java.lang.Integer.parseInt(Unknown Source)*

*at java.lang.Integer.parseInt(Unknown Source)*

*at pack1.MainClass.main(MainClass.java:7)*

**java.lang.ClassNotFoundException**

ClassNotFoundException is a checked type of exception. It is thrown when an application tries to load a class at run time using Class.forName() or loadClass() or findSystemClass() methods, but the class with specified name is not found in the classpath. This frequently occurs when you try to run your application without updating the class path with required *JAR* files.

*package pack1;*

*public class MainClass{*

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

*try{*

*Class.forName("oracle.jdbc.driver.OracleDriver");*

*}*

*catch (ClassNotFoundException e){*

*e.printStackTrace();*

*}*

*}*

*}*

The above code will throw java.lang.ClassNotFoundException if you don’t update the classpath with Oracle JDBC driver class.

*java.lang.ClassNotFoundException: oracle.jdbc.driver.OracleDriver*

*at java.net.URLClassLoader.findClass(Unknown Source)*

*at java.lang.ClassLoader.loadClass(Unknown Source)*

*at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)*

*at java.lang.ClassLoader.loadClass(Unknown Source)*

*at java.lang.Class.forName0(Native Method)*

*at java.lang.Class.forName(Unknown Source)*

*at pack1.MainClass.main(MainClass.java:9)*

**java.lang.ArithmeticException**

ArithmeticException is also a RunTimeException which is thrown when an abnormal arithmetic condition arises in an application. For example, divide by zero exception.

*package pack1;*

*public class MainClass{*

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

*int i = 10/0;*

*System.out.println(i);*

*}*

*}*

The above code will throw java.lang.ArithmeticException like below.

*Exception in thread "main" java.lang.ArithmeticException: / by zero*

*at pack1.MainClass.main(MainClass.java:7)*

**java.sql.SQLException**

SQLException is thrown when an application encounters with an error while interacting with the database. For example, passing the wrong username or password, passing the wrong URL of the database, passing invalid column name or column index etc. SQLException is also a checked exception.

*import java.sql.\*;*

*public class MainClass{*

*static{*

*try{*

*Class.forName("oracle.jdbc.driver.OracleDriver");*

*}*

*catch (ClassNotFoundException e)*

*{*

*System.out.println("Unable To Load The Driver class");*

*}*

*}*

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

*Connection con = null;*

*String URL = "jdbc:oracle:thin:@localhost:1521:XE";*

*String username = "username";*

*String password = "password";*

*try{*

*con = DriverManager.getConnection(URL, username, password);*

*}*

*catch (SQLException e) {*

*e.printStackTrace();*

*}*

*}*

*}*

In the above program, we are trying to connect to the database by passing wrong username and password. This will throw java.sql.SQLException.

*java.sql.SQLException: ORA-01017: invalid username/password; logon denied*

*at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java:112)*

*at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:331)*

*at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:283)*

*at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:278)*

*at oracle.jdbc.driver.T4CTTIoauthenticate.receiveOauth(T4CTTIoauthenticate.java:785)*

*at oracle.jdbc.driver.T4CConnection.logon(T4CConnection.java:362)*

*at oracle.jdbc.driver.PhysicalConnection.<init>(PhysicalConnection.java:414)*

*at oracle.jdbc.driver.T4CConnection.<init>(T4CConnection.java:165)*

*at oracle.jdbc.driver.T4CDriverExtension.getConnection(T4CDriverExtension.java:35)*

*at oracle.jdbc.driver.OracleDriver.connect(OracleDriver.java:801)*

*at java.sql.DriverManager.getConnection(Unknown Source)*

*at java.sql.DriverManager.getConnection(Unknown Source)*

*at MainClass.main(MainClass.java:31)*

**java.lang.ClassCastException**

ClassCastException occurs when an object of one type can not be casted to another type. While casting, an object must satisfy “IS-A” relationship. If it doesn’t satisfy then JVM will throw java.lang.ClassCastException.

*package pack1;*

*class A{*

*}*

*class B extends A{*

*}*

*public class MainClass{*

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

*A a = new A();*

*B b = (B)a;*

*}*

*}*

In the above example, we are trying to cast an object of type A to type B. But, it doesn’t satisfy “IS-A” relationship. i.e A is not of type B. This will throw java.lang.ClassCastException.

*Exception in thread "main" java.lang.ClassCastException: pack1.A cannot be cast to pack1.B*

*at pack1.MainClass.main(MainClass.java:19)*

**java.io.IOException**

IOException occurs when an IO operation fails in your application. IOException is a checked type of exception. This exception is the super class for all type of IO exceptions. Some of the popular IO exceptions are FileNotFoundException, SocketException, SSLException etc.

*package pack1;*

*import java.io.BufferedReader;*

*import java.io.FileNotFoundException;*

*import java.io.FileReader;*

*import java.io.IOException;*

*public class MainClass{*

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

*String filePath = "C:\\Users\\Bablad\\Desktop\\Open.txt";*

*BufferedReader reader = null;*

*try       {*

*reader = new BufferedReader(new FileReader(filePath));*

*}*

*catch (FileNotFoundException e) {*

*e.printStackTrace();*

*}*

*finally {*

*try{*

*if(reader != null){*

*reader.close();*

*}*

*}*

*catch (IOException e){*

*e.printStackTrace();*

*}*

*}*

*}*

*}*

In the above example, we are trying to read a file which doesn’t exist in the path specified. It causes java.io.FileNotFoundException.

*java.io.FileNotFoundException: C:\Users\Bablad\Desktop\Open.txt (The system cannot find the file specified)*

*at java.io.FileInputStream.open0(Native Method)*

*at java.io.FileInputStream.open(Unknown Source)*

*at java.io.FileInputStream.<init>(Unknown Source)*

*at java.io.FileInputStream.<init>(Unknown Source)*

*at java.io.FileReader.<init>(Unknown Source)*

*at pack1.MainClass.main(MainClass.java:18)*

**java.lang.InterruptedException**

You may have come across this exception if you have worked on multithreaded programming. InterruptedException is thrown when a sleeping thread or waiting thread is interrupted. The methods like sleep(), wait() and join() methods throw InterruptedException. InterruptedException is also a checked exception.

*public class MainClass {*

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

*Thread t = new Thread(){*

*public void run() {*

*try{*

*Thread.sleep(10000);   //Thread sleeps for 10s*

*}*

*catch (InterruptedException e) {*

*e.printStackTrace();*

*}*

*}*

*};*

*t.start();*

*try {*

*Thread.sleep(3000);   //main thread sleeping for 3s*

*}*

*catch (InterruptedException e) {*

*e.printStackTrace();*

*}*

*t.interrupt();    //interrupting thread t*

*}*

*}*

In the above example, main thread interrupts thread ***‘t’*** while it is sleeping. This causes java.lang.InterruptedException to arise.

*java.lang.InterruptedException: sleep interrupted*

*at java.lang.Thread.sleep(Native Method)*

*at MainClass$1.run(MainClass.java:11)*

**java.lang.SecurityException**

SecurityException is thrown by the security manager if an application violates the security rules. For example, changing a thread name or thread priority to which you don’t have access or using a package name which is already used. SecurityException indicates that application has violated the security rules and it cannot continue the execution.

*package java.lang;*

*public class MainClass {*

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

*System.out.println("Java Concept Of The Day");*

*}*

*}*

The above code will throw java.lang.SecurityException. Because, we are using package name “java.lang” which is already used in JDK.

*Exception in thread "main" java.lang.SecurityException: Prohibited package name: java.lang*

*at java.lang.ClassLoader.preDefineClass(Unknown Source)*

*at java.lang.ClassLoader.defineClass(Unknown Source)*

*at java.security.SecureClassLoader.defineClass(Unknown Source)*

*at java.net.URLClassLoader.defineClass(Unknown Source)*

*at java.net.URLClassLoader.access$100(Unknown Source)*

*at java.net.URLClassLoader$1.run(Unknown Source)*

*at java.net.URLClassLoader$1.run(Unknown Source)*

*at java.security.AccessController.doPrivileged(Native Method)*

*at java.net.URLClassLoader.findClass(Unknown Source)*

*at java.lang.ClassLoader.loadClass(Unknown Source)*

*at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)*

*at java.lang.ClassLoader.loadClass(Unknown Source)*

*at sun.launcher.LauncherHelper.checkAndLoadMain(Unknown Source)*

**java.lang.StackOverflowError**

StackOverflowError is a run time error which occurs when stack overflows. This happens when you keep calling the methods recursively.

*public class MainClass {*

*static void methodOne(){*

*methodTwo();*

*}*

*static void methodTwo() {*

*methodOne();*

*}*

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

*methodOne();*

*}*

*}*

The above code will throw java.lang.StackOverflowError, because methodOne() and methodTwo() are calling each other recursively.

|  |
| --- |
| *Exception in thread "main" java.lang.StackOverflowError*  *at MainClass.methodTwo(MainClass.java:10)*  *at MainClass.methodOne(MainClass.java:5)*  *at MainClass.methodTwo(MainClass.java:10)*  *at MainClass.methodOne(MainClass.java:5)*  *at MainClass.methodTwo(MainClass.java:10)*  *at MainClass.methodOne(MainClass.java:5)*  *at MainClass.methodTwo(MainClass.java:10)*  *at MainClass.methodOne(MainClass.java:5)*  *......*  ......  ...... |

**java.lang.NoClassDefFoundError**

NoClassDefFoundError is thrown when Java Runtime System tries to load the definition of a class which is no longer available. The required class definition was present at compile time but it was missing at run time.

*class A{*

*}*

*public class B{*

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

*A a = new A();*

*}*

*}*

When you compile the above program, two .class files will be generated. One is A.class and another one is B.class. If you run the B.class file after deleting the A.class file, java.lang.NoClassDefFoundError will be thrown.

|  |
| --- |
| *Exception in thread "main" java.lang.NoClassDefFoundError: A*  *at MainClass.main(MainClass.java:10)*  *Caused by: java.lang.ClassNotFoundException: A*  *at java.net.URLClassLoader.findClass(URLClassLoader.java:381)*  *at java.lang.ClassLoader.loadClass(ClassLoader.java:424)*  *at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)*  *at java.lang.ClassLoader.loadClass(ClassLoader.java:357)* |

**Chain exceptions:**

Sometimes, when you catch an exception, you want to throw a new exception. However, you may want to convey the information in the original exception in the new exception. This can be done by chaining exceptions.

First off, let's look at a code snippet that catches an exception and then throws a new exception without chaining. (Note: in these examples, I write the stacktraces to standard output rather than standard error.)

...

*System.out.println("\*\*\*no chaining example:");*

*try {*

*try {*

*throw new Exception("One");*

*} catch (Exception e) {*

*throw new Exception("Two");*

*}*

*} catch (Exception e) {*

*e.printStackTrace(System.out);*

*}*

...

The output from the above code is shown below. Notice that the information from the inner exception is lost when we display the stacktrace of the outer exception.

**\*\*\*no chaining example:**

*java.lang.Exception: Two*

*at test.ExceptionTest.main(ExceptionTest.java:11)*

Now, let's look at an example of chaining exceptions. In the inner try/catch block, we throw an Exception object with the message "Three". In its catch block, we catch the Exception and then throw a new Exception. However, notice the constructor of this Exception. It takes a message String ("Four") and then a Throwable (e). The Throwable parameter allows us to chain the original exception to the new exception.

...

*System.out.println("\n\*\*\*chaining example 1:");*

*try {*

*try {*

*throw new Exception("Three");*

*} catch (Exception e) {*

*throw new Exception("Four", e);*

*}*

*} catch (Exception e) {*

*e.printStackTrace(System.out);*

*System.out.println("###what was the cause:");*

*e.getCause().printStackTrace(System.out);*

*}*

...

In the outer try/catch block, we catch the exception and display its stacktrace. Notice that the stacktrace includes the new message ("Four") along with the original message ("Three"). In addition, as a bonus, I called e.getCause() on the "Four" exception to get the exception that's the cause for the "Four" exception, which is the "Three" exception. Thus, we see that we can actually get the original exception from the new exception.

**\*\*\*chaining example 1:**

java.lang.Exception: Four

at test.ExceptionTest.main(ExceptionTest.java:22)

Caused by: java.lang.Exception: Three

at test.ExceptionTest.main(ExceptionTest.java:20)

**###what was the cause:**

*java.lang.Exception: Three*

*at test.ExceptionTest.main(ExceptionTest.java:20)*

As another short example of chaining, here we throw an Exception in the inner try/catch block, and we catch the Exception and throw a new Exception. However, notice that this time, when we throw the new Exception, we use a constructor that takes a single argument, a Throwable, so we use our caught Exception e as the parameter to the constructor.

...

*System.out.println("\n\*\*\*chaining example 2:");*

*try {*

*try {*

*throw new Exception("Five");*

*} catch (Exception e) {*

*throw new Exception(e);*

*}*

*} catch (Exception e) {*

*e.printStackTrace(System.out);*

*}*

...

In the outer try/catch block, we catch this exception and display the stacktrace, shown below. Notice that it displays the chained exceptions.

**\*\*\*chaining example 2:**

*java.lang.Exception: java.lang.Exception: Five*

*at test.ExceptionTest.main(ExceptionTest.java:35)*

*Caused by: java.lang.Exception: Five*

*at test.ExceptionTest.main(ExceptionTest.java:33)*

The ExceptionTest class contains all of the code snippets shown above.

[**ExceptionTest.java**](http://www.avajava.com/tutorials/general-java/how-do-i-chain-exceptions/ExceptionTest.java)

*package test;*

*public class ExceptionTest {*

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

*System.out.println("\*\*\*no chaining example:");*

*try {*

*try {*

*throw new Exception("One");*

*} catch (Exception e) {*

*throw new Exception("Two");*

*}*

*} catch (Exception e) {*

*e.printStackTrace(System.out);*

*}*

*System.out.println("\n\*\*\*chaining example 1:");*

*try {*

*try {*

*throw new Exception("Three");*

*} catch (Exception e) {*

*throw new Exception("Four", e);*

*}*

*} catch (Exception e) {*

*e.printStackTrace(System.out);*

*System.out.println("###what was the cause:");*

*e.getCause().printStackTrace(System.out);*

*}*

*System.out.println("\n\*\*\*chaining example 2:");*

*try {*

*try {*

*throw new Exception("Five");*

*} catch (Exception e) {*

*throw new Exception(e);*

*}*

*} catch (Exception e) {*

*e.printStackTrace(System.out);*

*}*

*}*

*}*

The console output from executing ExceptionTest is shown here:

Console output from executing ExceptionTest

**\*\*\*no chaining example:**

*java.lang.Exception: Two*

*at test.ExceptionTest.main(ExceptionTest.java:11)*

**\*\*\*chaining example 1:**

*java.lang.Exception: Four*

*at test.ExceptionTest.main(ExceptionTest.java:22)*

*Caused by: java.lang.Exception: Three*

*at test.ExceptionTest.main(ExceptionTest.java:20)*

**###what was the cause:**

*java.lang.Exception: Three*

*at test.ExceptionTest.main(ExceptionTest.java:20)*

**\*\*\*chaining example 2:**

*java.lang.Exception: java.lang.Exception: Five*

*at test.ExceptionTest.main(ExceptionTest.java:35)*

*Caused by: java.lang.Exception: Five*

*at test.ExceptionTest.main(ExceptionTest.java:33)*