**What Is an Exception?**

**Definition:** An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

The term exception is shorthand for the phrase "exceptional event."

**2. How Exception Handling Works in Java**

This topic is covered in a separate post on **How Exception Handling Works in Java**

**3. Three Kinds of Exceptions**

Let's look at the three basic categories of exceptions with examples.

1. Checked Exception or Compile Time Exception
2. Error
3. Un-Checked Exception or Runtime Exception

Read more detail on The **Three Kinds of Exceptions in Java**

**4. Exception Handling Keywords in Java**

Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.

Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block.

If an exception occurs within the *try*block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java runtime system.

To manually throw an exception, use the keyword *throw*. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a *finally*block.

This is the general form of an exception-handling block:

try {

// block of code to monitor for errors

}

catch (ExceptionType1 exOb) {

// exception handler for ExceptionType1

}

catch (ExceptionType2 exOb) {

// exception handler for ExceptionType2

}

// ...

finally {

// block of code to be executed after try block ends

}

Here, *ExceptionType* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

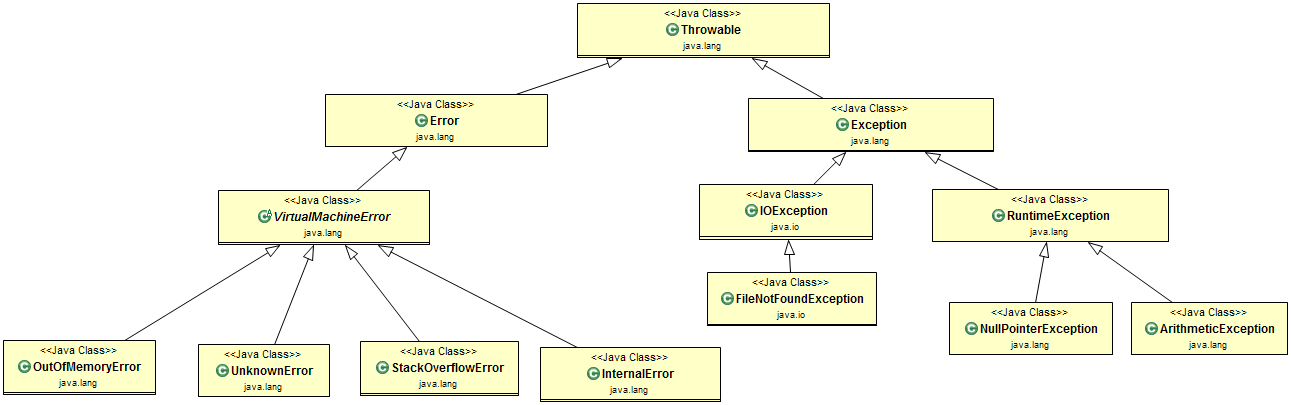
All exception handling keywords are nicely explained on Exception Handling Keywords in Java article.

**5. Java Exception Hierarchy in Java**

The objects that inherit from the **Throwable** class include direct descendants (objects that inherit directly from the **Throwable** class) and indirect descendants (objects that inherit from children or grandchildren of the **Throwable** class).

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. **Throwable** are two subclasses that partition exceptions into two distinct branches that are:

1. Error Class
2. Exception Class

****

Read more about Exceptions Hierarchy on **Exceptions Hierarchy in Java**

**6. The try with resources Statement**

The try-with-resources statement is a try statement that declares one or more resources.

Let's learn below topics about try-with-resources on **The try-with-resources Statement with Examples**

* What is a try-with-resources statement?
* Overview of java.lang.AutoCloseable Interface
* Working on a try-with-resources statement with *BufferedReader* Example
* Resource closing using finally block(Prior to Java SE 7)
* Declare one or more resources in a try-with-resources statement
* Custom Resource object close examples
* try-with-resources Statement with File IO Examples

**7. How to Create Custom Exceptions**

Although Java’s built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable).

Let's learn below topics about creating our own exception on **Guide to Create Custom Exceptions**

* Steps to create custom exceptions
* How to create checked custom exceptions with examples?
* How to create an unchecked custom exception with examples?
* More examples

Read more on **Guide to Create Custom Exceptions**.

**8. Java Chained Exceptions**

The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception.

Read more in detail with examples on **Java Chained Exceptions**

**9. Advantages of Java Exceptions**

* Advantage 1: Separating Error-Handling Code from "Regular" Code
* Advantage 2: Propagating Errors Up the Call Stack
* Advantage 3: Grouping and Differentiating Error Types

Let's understand each advantage of Exceptions on a separate article on **Advantages of Java Exceptions**

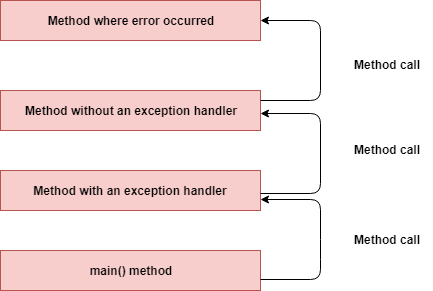
**10. Java Exception Handling Best Practices**

1. Clean up Resources in a Finally Block or Use a Try-With-Resource Statement
2. Throw Specific Exception
3. Do not catch the Exception class rather catch specific subclasses
4. Never catch a Throwable class
5. Always correctly wrap the exceptions in custom exceptions so that stack trace is not lost
6. Catch the most specific exception first
7. Don’t ignore exceptions rather log the exceptions
8. Never throw any exception from finally block
9. Don’t use printStackTrace() statement or similar methods
10. Use finally blocks instead of catch blocks if you are not going to handle the exception
11. Validate user input to catch adverse conditions very early in request processing
12. Throw Exceptions With Descriptive Messages

**How the Exception Handling Works in Java**

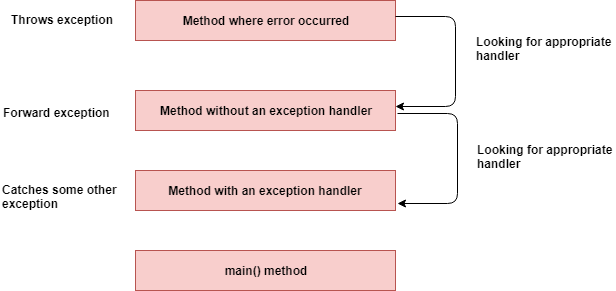
First, we will discuss conceptually how the exception handling works in Java. In the next section, we will demonstrate the same with programming example.  
  
**Step 1:**When an error occurs within a method, the method creates an object and hands it off to the runtime system this object is called an *exception object*. The *exception object*contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an *exception*.

**Step 2:** After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack*. The following diagram shows the call stack of three method calls, where the first method called has the exception handler.

****

**Step 3:** The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler.

An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.  
  
**Step 4:**The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the following diagram, the runtime system (and, consequently, the program) terminates.

****

Let's demonstrate above how exception handling works in Java with a programmatic example.

Let's develop a program for reading a file example, Now we will create three methods which don't have exception handling code.  
  
**1. exceptionWithoutHandler()**  
**2. exceptionWithoutHandler1()**  
**3. exceptionWithoutHandler2()**

Let's create an *exceptionHandler()* method which contains an exception handling code that is which has *catch*block to handle the exception.

1. Read file from some location, if a file does not exist in a given location this code throws *IOException*or *FileNotFoundException*.
2. The *exceptionWithoutHandler()* method don't have exception handler(catch block) to handle this exception.
3. The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code exists in *exceptionHandler()* method.

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

/\*\*

\* This class demonstrate how Exception Handling Works

\* @author javaguides.net

\*

\*/

public class ExceptionHandlingWorks {

public static void main(String[] args) {

exceptionHandler();

}

private static void exceptionWithoutHandler() throws IOException {

try (BufferedReader reader = new BufferedReader(new FileReader(new File("/invalid/file/location")))) {

int c;

// Read and display the file.

while ((c = reader.read()) != -1) {

System.out.println((char) c);

}

}

}

private static void exceptionWithoutHandler1() throws IOException {

exceptionWithoutHandler();

}

private static void exceptionWithoutHandler2() throws IOException {

exceptionWithoutHandler1();

}

private static void exceptionHandler() {

try {

exceptionWithoutHandler2();

} catch (IOException e) {

System.out.println("IOException caught!");

}

}

}

Output:

IOException caught!

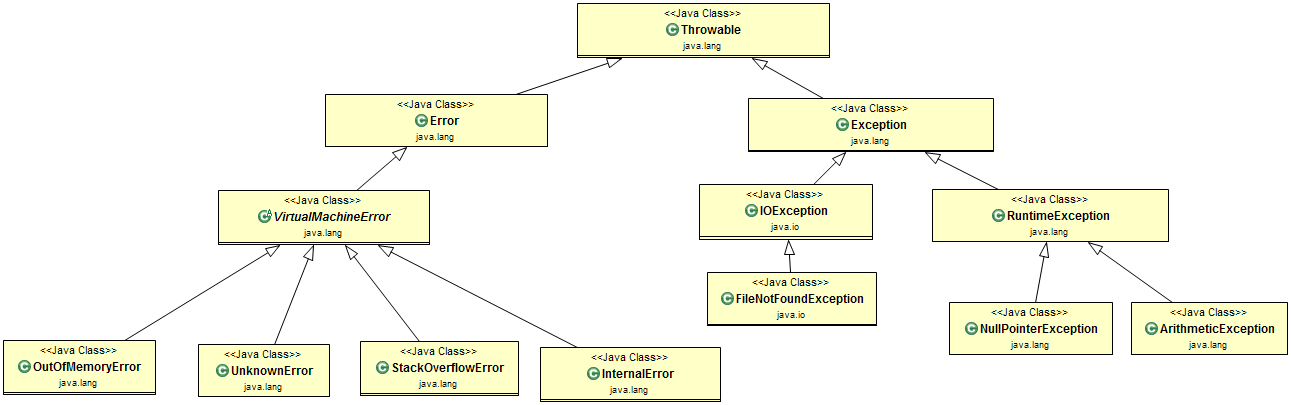
### Exceptions Hierarchy in Java

In this article, we will learn the exceptions class hierarchy in *java.lang* package. The objects that inherit from the **Throwable** class include direct descendants (objects that inherit directly from the **Throwable** class) and indirect descendants (objects that inherit from children or grandchildren of the **Throwable** class).

**Throwable** has two direct descendants:

1. Error Class
2. Exception Class

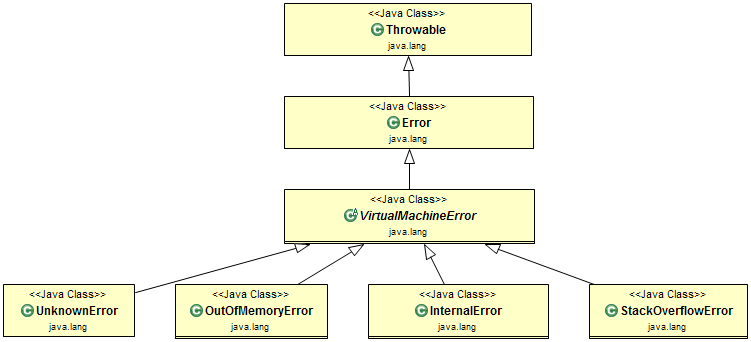
The figure below illustrates the class hierarchy of the **Throwable** class and its most significant subclasses.

****

## 1. Error Class

When a dynamic linking failure or other hard failures in the Java virtual machine occurs, the virtual machine throws an Error. Simple programs typically do not catch or throw Errors.

The figure below illustrates the class hierarchy of *Error* Class.

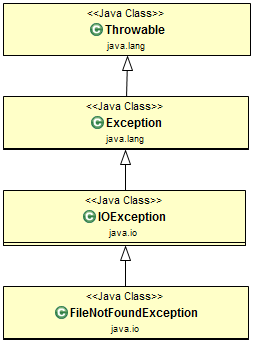
****

## 2. Exception Class

Most programs **throw** and **catch** objects that derive from the Exception class. An Exception indicates that a problem occurred, but it is not a serious system problem. Most programs you write will **throw** and **catch** Exceptions as opposed to Errors.

The Java platform defines the many descendants of the *Exception*class. These descendants indicate various types of exceptions that can occur.

The figure below illustrates the class hierarchy of the Exception Class.

****

**Exception Class Example:**

*FileNotFoundException*exceptions are good examples for *Exception*class. In this example, we will use the FileNotFoundException exception type to demonstrate the usage of the Exception class.

In this example, *FileReader* class try to read a file from an invalid location will throw *FileNotFoundException*exception.

public class FileNotFoundExceptionExample {

public static void main(String[] args) {

BufferedReader reader = null;

try {

reader = new BufferedReader(new FileReader(new File("/invalid/file/location")));

} catch (FileNotFoundException e) {

System.err.println("FileNotFoundException caught!");

}

}

}

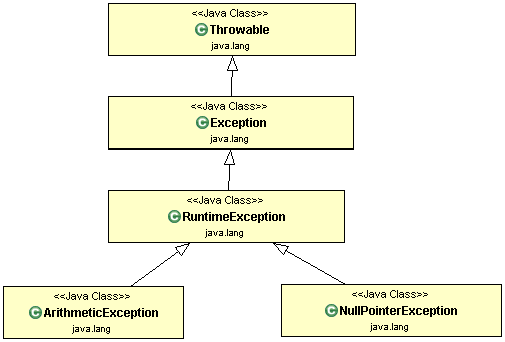
Output:

FileNotFoundException caught!

### RuntimeException Class

One Exception subclass, RuntimeException, is reserved for exceptions that indicate an incorrect use of an API. An example of a runtime exception is NullPointerException, which occurs when a method tries to access a member of an object through a null reference.

The figure below illustrates the class hierarchy of the RuntimeException Class.

****

**RuntimeException Example:**

Let's consider *NullPointerException*is the best example for runtime exception. In below example, Person class object is not created using the new keyword but it is just declared with a null value. Now we are trying to access personName field value on null reference so JVM will throw NullPointerException exception here.

public class NullPointerExceptionExample {

public static void main(String[] args) {

Person personObj = null;

try {

String name = personObj.personName; // Accessing the field of a null object

personObj.personName = "Jon Doe"; // Modifying the field of a null object

} catch (NullPointerException e) {

System.err.println("NullPointerException caught!");

}

}

}

class Person {

public String personName;

public String getPersonName() {

return personName;

}

public void setPersonName(String personName) {

this.personName = personName;

}

}

Output:

NullPointerException caught!

**Java Chained Exceptions Overview**

The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception.

For example, imagine a situation in which a method throws an *ArithmeticException*because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an *ArithmeticException*, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist. This concept was introduced in JDK 1.4.

**Throwable Class**

**Throwable** class has some constructors and methods to support chained exceptions. Firstly, let’s look at the constructors.

1. Throwable(Throwable cause) – Throwable has a single parameter, which specifies the actual cause of an Exception.
2. Throwable(String desc, Throwable cause) – this constructor accepts an Exception description with the actual cause of an Exception as well.

Next, let’s have a look at the methods this class provides:

* getCause() method – This method returns the actual cause associated with the current Exception.
* initCause() method – It sets an underlying cause with invoking Exception.

Read more about Throwable Class on **java.lang.Throwable Class in Java**

**Java Chained Exceptions Example**

Here is an example that illustrates the mechanics of handling chained exceptions:

// Demonstrate exception chaining.

class ChainExcDemo {

static void demoproc() {

// create an exception

NullPointerException e = new NullPointerException("top layer");

// add a cause

e.initCause(new ArithmeticException("cause"));

throw e;

}

public static void main(String args[]) {

try {

demoproc();

} catch (NullPointerException e) {

// display top level exception

System.out.println("Caught: " + e);

// display cause exception

System.out.println("Original cause: " + e.getCause());

}

}

}

Output:

Caught: java.lang.NullPointerException: top layer

Original cause: java.lang.ArithmeticException: cause

In this example, the top-level exception is *NullPointerException*. To it is added a cause exception, *ArithmeticException*. When the exception is thrown out of *demoproc( )*method, it is caught by *main(*). There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling *getCause(*).

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design.

Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

### java.lang.Throwable Class in Java

The *Throwable* class is the superclass of all *errors* and *exceptions* in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the *Java Virtual Machine* or can be thrown by the Java *throw* statement. Similarly, only this class or one of its subclasses can be the argument type in a *catch* clause.

Instances of two subclasses, *Error* and *Exception*, are conventionally used to indicate that exceptional situations have occurred. Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information (such as stack trace data).

## *Throwable* class Declaration

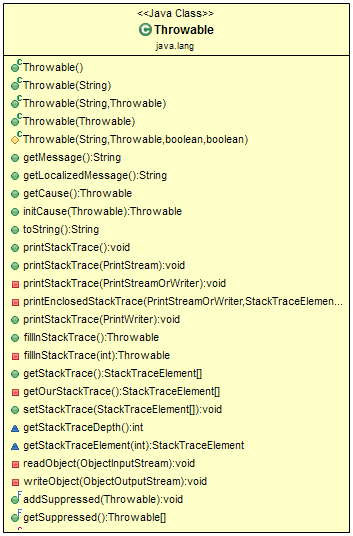
Following is the declaration for *java.lang.Throwable* class −

public class Throwable

extends Object

implements Serializable

## *Throwable* class Constructors

****

* *Throwable()* - Constructs a new throwable with null as its detail message.
* *Throwable(String message)* - Constructs a new throwable with the specified detail message.
* *Throwable(String message, Throwable cause)* - Constructs a new throwable with the specified detail message and cause.
* *protected Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)* - Constructs a new throwable with the specified detail message, cause, suppression enabled or disabled, and writable stack trace enabled or disabled.
* *Throwable(Throwable cause)* - Constructs a new throwable with the specified cause and a detail message of *(cause==null ? null : cause.toString())* (which typically contains the class and detail message of cause).

## *Throwable* class Methods

* *void addSuppressed(Throwable exception)* - Appends the specified exception to the exceptions that were suppressed in order to deliver this exception.
* *Throwable fillInStackTrace()* - Fills in the execution stack trace.
* *Throwable getCause()* - Returns the cause of this throwable or null if the cause is nonexistent or unknown.
* *String getLocalizedMessage()* - Creates a localized description of this throwable.
* *String getMessage()* - Returns the detail message string of this throwable.
* *StackTraceElement[] getStackTrace()* - Provides programmatic access to the stack trace information printed by printStackTrace().
* *Throwable[] getSuppressed()* - Returns an array containing all of the exceptions that were suppressed, typically by the try-with-resources statement, in order to deliver this exception.
* **Throwable initCause(Throwable cause)** - Initializes the cause of this throwable to the specified value.
* *void printStackTrace()* - Prints this throwable and its backtrace to the standard error stream.
* *void printStackTrace(PrintStream s)* - Prints this throwable and its backtrace to the specified print stream.
* *void printStackTrace(PrintWriter s)* - Prints this throwable and its backtrace to the specified print writer.
* *void setStackTrace(StackTraceElement[] stackTrace)* - Sets the stack trace elements that will be returned by getStackTrace() and printed by printStackTrace() and related methods.
* *String toString()* - Returns a short description of this throwable.

## *Throwable* class Example

public class ThrowableExample {

public static void main(String[] args) {

try{

throwException(null);

}catch (Throwable e) {

System.out.println(e.getMessage());

System.out.println(e.getLocalizedMessage());

System.out.println(e.getClass());

}

}

private static void throwException(String str) throws Throwable{

if(str == null){

throw new Throwable("String is null.");

}

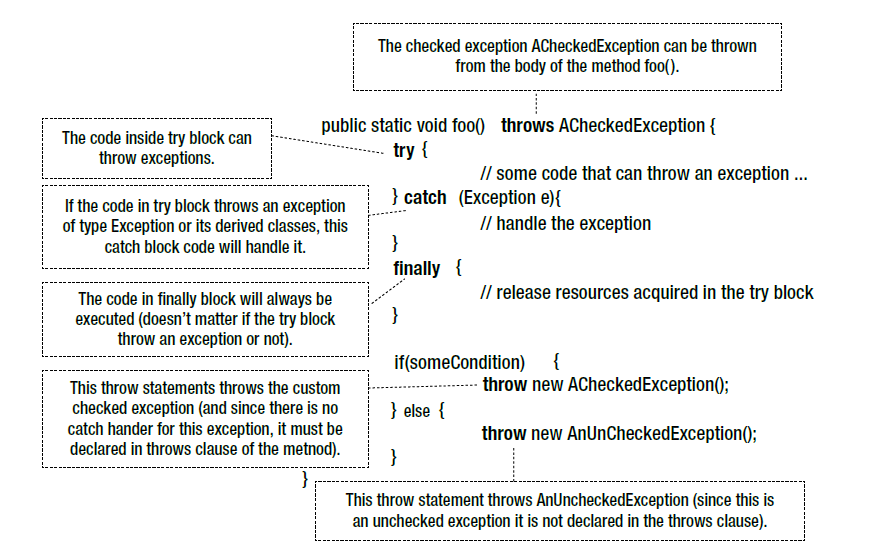
}

}

### Exception Handling Keywords in Java

Java exception handling is managed via five keywords, in this article, we will use all these five keywords with examples.

Let's to understand the basic syntax of try, catch, throw, throws and finally keywords. This diagram summaries usage of these keywords.

****

Let's list five exception handling keywords and we will discuss each keyword with an example.

**[](https://4.bp.blogspot.com/-FUBbD4YrtrM/W5zOgl37zdI/AAAAAAAAD1k/bxxnQg3Tang_XnL1GJytagao0xvjDItegCLcBGAs/s1600/exception-handling-keywords.png)**

## 1. try Block

Enclose the code that might throw an exception within a try block. If an exception occurs within the *try*block, that exception is handled by an exception handler associated with it. The try block contains at least one catch block or finally block.

### The syntax of java try-catch

try{

//code that may throw exception

}catch(Exception\_class\_Name ref){}

### The syntax of a try-finally block

try{

//code that may throw exception

}finally{}

### Nested try block

The *try*block within a *try*block is known as nested *try*block in java.

**Why use nested try block?**

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

public class NestedTryBlock {

public static void main(String args[]) {

try {

try {

System.out.println(" This gives divide by zero error");

int b = 39 / 0;

} catch (ArithmeticException e) {

System.out.println(e);

}

try {

System.out.println(" This gives Array index out of bound exception");

int a[] = new int[5];

a[5] = 4;

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println(e);

}

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

} catch (Exception e) {

System.out.println("handeled");

}

System.out.println("normal flow..");

}

}

## 2. catch Block

Java *catch*block is used to handle the Exception. It must be used after the *try*block only. You can use multiple *catch*block with a single try.

Syntax:

try

{

//code that cause exception;

}

catch(Exception\_type e)

{

//exception handling code

}

### catch Block Examples

Let's demonstrate the usage of catch block using *ArithmeticException*type.

**Example 1:** *ArithmeticException*exception type example

public class Arithmetic {

public static void main(String[] args) {

try {

int result = 30 / 0; // Trying to divide by zero

} catch (ArithmeticException e) {

System.out.println("ArithmeticException caught!");

}

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

}

}

Output:

ArithmeticException caught!

rest of the code executes

**Example 2:** *ArrayIndexOutOfBoundsException*exception type example

public class ArrayIndexOutOfBounds {

public static void main(String[] args) {

int[] nums = new int[] { 1, 2, 3 };

try {

int numFromNegativeIndex = nums[-1]; // Trying to access at negative index

int numFromGreaterIndex = nums[4]; // Trying to access at greater index

int numFromLengthIndex = nums[3]; // Trying to access at index equal to size of the array

} catch (ArrayIndexOutOfBoundsException e) {

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

}

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

}

}

If you have to perform different tasks at the occurrence of different Exceptions, use java multi-catch block.

### Multi-catch Block

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more *catch*clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

public class TestMultipleCatchBlock {

public static void main(String args[]) {

try {

int a = args.length;

System.out.println("a = " + a);

int b = 42 / a;

int c[] = { 1 };

c[42] = 99;

} catch (ArithmeticException e) {

System.out.println("Divide by 0: " + e);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index oob: " + e);

}

System.out.println("After try/catch blocks.");

}

}

This program will cause a division-by-zero exception if it is started with no command line arguments, since a will equal zero. It will survive the division if you provide a command-line argument, setting a to something larger than zero. But it will cause an ArrayIndexOutOfBoundsException, since the *int array c* has a length of 1, yet the program attempts to assign a value to c[42].

Here is the output generated by running it both ways: Output:

C:\>java MultipleCatches

a = 0

Divide by 0: java.lang.ArithmeticException: / by zero

After try/catch blocks.

C:\>java MultipleCatches TestArg

a = 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException:42

After try/catch blocks.

When you use multiple *catch*statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

### Catching More Than One Type of Exception with One Exception Handler

In Java SE 7 and later, a single *catch*block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to *catch*an overly broad exception.

In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

catch (IOException|SQLException ex) {

logger.log(ex);

throw ex;

}

## 3. throw Keyword

It is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

throw ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

Primitive types, such as int or char, as well as non-Throwable classes, such as **String** and *Object*, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter in a *catch*clause or creating one with the *new*operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing *try* block is inspected to see if it has a *catch*statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

### Using throw Keyword Example 1

Let's create custom exception ResourceNotFoundException and use throw keyword is used to explicitly throw an exception.

package com.javaguides.exceptions.examples;

public class TestResourceNotFoundException {

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

ResourceManager manager = new ResourceManager();

manager.getResource(0);

}

}

class Resource {

private int id;

public Resource(int id) {

super();

this.id = id;

}

}

class ResourceManager {

public Resource getResource(int id) throws ResourceNotFoundException {

if (id == 10) {

new Resource(id);

} else {

throw new ResourceNotFoundException("Resource not found with id ::" + id);

}

return null;

}

}

class ResourceNotFoundException extends Exception {

private static final long serialVersionUID = 1L;

public ResourceNotFoundException(Object resourId) {

super(resourId != null ? resourId.toString() : null);

}

}

Output:

Exception in thread "main" com.javaguides.exceptions.examples.ResourceNotFoundException: Resource not found with id ::0

at com.javaguides.exceptions.examples.ResourceManager.getResource(TestResourceNotFoundException.java:26)

at com.javaguides.exceptions.examples.TestResourceNotFoundException.main(TestResourceNotFoundException.java:6)

Note that we have used below code to demonstrate usage of throw keyword.

throw new ResourceNotFoundException("Resource not found with id ::" + id);

### Using throw Keyword Example 2

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

package com.javaguides.exceptions.examples;

//Demonstrate throw.

class ThrowDemo {

static void demoproc() {

try {

throw new NullPointerException("demo");

} catch (NullPointerException e) {

System.out.println("Caught inside demoproc.");

throw e; // rethrow the exception

}

}

public static void main(String args[]) {

try {

demoproc();

} catch (NullPointerException e) {

System.out.println("Recaught: " + e);

}

}

}

This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls demoproc( ). The demoproc( ) method then sets up another exception-handling context and immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown. output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

## 4. throws Keyword

The Java throws keyword is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as *NullPointerException*, it is programmers fault that he is not performing checkup before the code is used.

### The syntax of java throws

return\_type method\_name() throws exception\_class\_name{

//method code

}

We declare only checked exception using a throws keyword. Let's see an example to demonstrate the usage of a throws keyword.

Basically, whenever exception arises there two cases, either you should handle the exception using try/catch or you declare the exception i.e. specifying throws with the method.

### throws Keyword Example

Note that exceptionWithoutHandler(), exceptionWithoutHandler1() and exceptionWithoutHandler2() methods uses **throws**keyword to declare exception.

public class ExceptionHandlingWorks {

public static void main(String[] args) {

exceptionHandler();

}

private static void exceptionWithoutHandler() throws IOException {

try (BufferedReader reader = new BufferedReader(new FileReader(new File("/invalid/file/location")))) {

int c;

// Read and display the file.

while ((c = reader.read()) != -1) {

System.out.println((char) c);

}

}

}

private static void exceptionWithoutHandler1() throws IOException {

exceptionWithoutHandler();

}

private static void exceptionWithoutHandler2() throws IOException {

exceptionWithoutHandler1();

}

private static void exceptionHandler() {

try {

exceptionWithoutHandler2();

} catch (IOException e) {

System.out.println("IOException caught!");

}

}

}

## 5. finally Block

* Java *finally*block is a block that is used to execute important code such as closing connection, stream etc.
* Java *finally*block is always executed whether an exception is handled or not.
* Java *finally*block follows try or catch block.
* For each try block, there can be zero or more catch blocks, but only one *finally*block.
* The *finally*block will not be executed if program exits(either by calling *System.exit()* or by causing a fatal error that causes the process to abort).

### finally Block Different Scenario Examples

Here is an example program that shows three methods that exit in various ways, none without executing their *finally*clauses:

// Demonstrate finally.

class FinallyDemo {

// Throw an exception out of the method.

static void procA() {

try {

System.out.println("inside procA");

throw new RuntimeException("demo");

} finally {

System.out.println("procA's finally");

}

}

// Return from within a try block.

static void procB() {

try {

System.out.println("inside procB");

return;

} finally {

System.out.println("procB's finally");

}

}

// Execute a try block normally.

static void procC() {

try {

System.out.println("inside procC");

} finally {

System.out.println("procC's finally");

}

}

public static void main(String args[]) {

try {

procA();

} catch (Exception e) {

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

}

procB();

procC();

}

}

Output:

inside procA

procA's finally

Exception caught

inside procB

procB's finally

inside procC

procC's finally

In this example, procA( ) prematurely breaks out of the try by throwing an exception. The *finally*clause is executed on the way out.

procB( )’s try statement is exited via a return statement. The *finally*clause is executed before procB( ) returns.

In procC( ), the try statement executes normally, without error. However, the *finally*block is still executed.

### finally Block Examples

**Example 1:** In this example, we have used **FileInputStream** to read the simple.txt file. After reading a file the resource **FileInputStream** should be closed by using finally block.

public class FileInputStreamExample {

public static void main(String[] args) {

FileInputStream fis = null;

try {

File file = new File("sample.txt");

fis = new FileInputStream(file);

int content;

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

// convert to char and display it

System.out.print((char) content);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

if(fis != null){

try {

fis.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

}

**Example 2:** When we use JDBC connection to communicate with a database. The resource that we have used are ResultSet, Statement, and Connection (in that order) should be closed in a finally block when you are done with them, something like that:

Connection conn = null;

PreparedStatement ps = null;

ResultSet rs = null;

try {

// Do stuff

...

} catch (SQLException ex) {

// Exception handling stuff

...

} finally {

if (rs != null) {

try {

rs.close();

} catch (SQLException e) { /\* ignored \*/}

}

if (ps != null) {

try {

ps.close();

} catch (SQLException e) { /\* ignored \*/}

}

if (conn != null) {

try {

conn.close();

} catch (SQLException e) { /\* ignored \*/}

}

}

### Java try/catch Block

we will learn in depth about try/catch block and how to use try/catch block to handle exceptions.  
What will we learn in this articles?

* *try*Block
* The syntax of java *try-catch*
* The syntax of *a try-finally* block
* Nested *try*block
* *catch*Block
* *try/catch* Block Examples
* Multi-*catch*Block
* Catching More Than One Type of Exception with One Exception Handler

## try Block

Enclose the code that might throw an exception within a try block. If an exception occurs within the *try* block, that exception is handled by an exception handler associated with it. The *try* block contains at least one *catch* block or *finally* block.

### The syntax of java try-catch

try{

//code that may throw exception

}catch(Exception\_class\_Name ref){}

### The syntax of a try-finally block

try{

//code that may throw exception

}finally{}

### Nested try block

The *try* block within a try block is known as nested *try* block in java.

**Why use nested try block?**

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

public class NestedTryBlock {

public static void main(String args[]) {

try {

try {

System.out.println(" This gives divide by zero error");

int b = 39 / 0;

} catch (ArithmeticException e) {

System.out.println(e);

}

try {

System.out.println(" This gives Array index out of bound exception");

int a[] = new int[5];

a[5] = 4;

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println(e);

}

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

} catch (Exception e) {

System.out.println("handeled");

}

System.out.println("normal flow..");

}

}

## catch Block

Java *catch* block is used to handle the *Exception*. It must be used after the *try* block only. You can use multiple *catch* block with a single try.

Syntax:

try

{

//code that cause exception;

}

catch(Exception\_type e)

{

//exception handling code

}

### try/catch Block Examples

Let's demonstrate the usage of *catch* block using *ArithmeticException* type.

**Example 1:** *ArithmeticException*exception type example.

public class Arithmetic {

public static void main(String[] args) {

try {

int result = 30 / 0; // Trying to divide by zero

} catch (ArithmeticException e) {

System.out.println("ArithmeticException caught!");

}

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

}

}

Output:

ArithmeticException caught!

rest of the code executes

**Example 2:** *ArrayIndexOutOfBoundsException*exception type example.

public class ArrayIndexOutOfBounds {

public static void main(String[] args) {

int[] nums = new int[] { 1, 2, 3 };

try {

int numFromNegativeIndex = nums[-1]; // Trying to access at negative index

int numFromGreaterIndex = nums[4]; // Trying to access at greater index

int numFromLengthIndex = nums[3]; // Trying to access at index equal to size of the array

} catch (ArrayIndexOutOfBoundsException e) {

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

}

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

}

}

If you have to perform different tasks at the occurrence of different Exceptions, use java *multi-catch* block.

### Multi-catch Block

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more *catch* clauses, each catching a different type of exception. When an exception is thrown, each *catch* statement is inspected in order, and the first one whose type matches that of the exception is executed.

After one *catch*statement executes, the others are bypassed, and execution continues after the *try/catch* block.

public class TestMultipleCatchBlock {

public static void main(String args[]) {

try {

int a = args.length;

System.out.println("a = " + a);

int b = 42 / a;

int c[] = { 1 };

c[42] = 99;

} catch (ArithmeticException e) {

System.out.println("Divide by 0: " + e);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index oob: " + e);

}

System.out.println("After try/catch blocks.");

}

}

This program will cause a **division-by-zero** exception if it is started with no command line arguments, since a will equal zero. It will survive the division if you provide a command-line argument, setting a to something larger than zero.

But it will cause an ArrayIndexOutOfBoundsException, since the *int array c* has a length of 1, yet the program attempts to assign a value to *c[42].*

Here is the output generated by running it both ways.

Output:

C:\>java MultipleCatches

a = 0

Divide by 0: java.lang.ArithmeticException: / by zero

After try/catch blocks.

C:\>java MultipleCatches TestArg

a = 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException:42

After try/catch blocks.

When you use multiple *catch*statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will *catch*exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

### Catching More Than One Type of Exception with One Exception Handler

In Java SE 7 and later, a single *catch*block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

In the *catch*clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

catch (IOException|SQLException ex) {

logger.log(ex);

throw ex;

}

### Java throw Keyword

In the previous article **Java try/catch Block**, we have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the *throw*statement. The general form of throw:

throw ThrowableInstance;

Here, ThrowableInstance must be an object of type **Throwable** or a subclass of **Throwable**.

Primitive types, such as *int* or *char*, as well as non-Throwable classes, such as **String** and *Object*, cannot be used as exceptions.

There are two ways you can obtain a **Throwable** object:

* using a parameter in a *catch*clause.
* creating one with the *new*operator.

The flow of execution stops immediately after the *throw*statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a *catch*statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching *catch*is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and *throws*an exception. The handler that catches the exception rethrows it to the outer handler.

## Java throw Keyword Examples

### Using throw Keyword Example 1

Let's create custom exception ResourceNotFoundException and use throw keyword is used to explicitly throw an exception.

package com.javaguides.exceptions.examples;

public class TestResourceNotFoundException {

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

ResourceManager manager = new ResourceManager();

manager.getResource(0);

}

}

class Resource {

private int id;

public Resource(int id) {

super();

this.id = id;

}

}

class ResourceManager {

public Resource getResource(int id) throws ResourceNotFoundException {

if (id == 10) {

new Resource(id);

} else {

throw new ResourceNotFoundException("Resource not found with id ::" + id);

}

return null;

}

}

class ResourceNotFoundException extends Exception {

private static final long serialVersionUID = 1L;

public ResourceNotFoundException(Object resourId) {

super(resourId != null ? resourId.toString() : null);

}

}

Output:

Exception in thread "main" com.javaguides.exceptions.examples.ResourceNotFoundException: Resource not found with id ::0

at com.javaguides.exceptions.examples.ResourceManager.getResource(TestResourceNotFoundException.java:26)

at com.javaguides.exceptions.examples.TestResourceNotFoundException.main(TestResourceNotFoundException.java:6)

Note that we have used below code to demonstrate usage of throw keyword.

throw new ResourceNotFoundException("Resource not found with id ::" + id);

### Using throw Keyword Example 2

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

package com.javaguides.exceptions.examples;

//Demonstrate throw.

class ThrowDemo {

static void demoproc() {

try {

throw new NullPointerException("demo");

} catch (NullPointerException e) {

System.out.println("Caught inside demoproc.");

throw e; // rethrow the exception

}

}

public static void main(String args[]) {

try {

demoproc();

} catch (NullPointerException e) {

System.out.println("Recaught: " + e);

}

}

}

This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls demoproc( ). The demoproc( ) method then sets up another exception-handling context and immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown. output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

### Java throws Keyword

The Java *throws*keyword is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as *NullPointerException*, it is programmers fault that he is not performing checkup before the code is used.

### The syntax of Java throws Keyword

return\_type method\_name() throws exception\_class\_name{

//method code

}

We declare only checked exception using a throws keyword. Let's see an example to demonstrate the usage of a throws keyword.

Basically, whenever exception arises there two cases, either you should handle the exception using **try/catch** or you declare the exception i.e. specifying *throws*with the method.

### throws Keyword Example

In this example, the exceptionWithoutHandler(), exceptionWithoutHandler1() and exceptionWithoutHandler2() methods uses *throws*keyword to declare exception.

public class ExceptionHandlingWorks {

public static void main(String[] args) {

exceptionHandler();

}

private static void exceptionWithoutHandler() throws IOException {

try (BufferedReader reader = new BufferedReader(new FileReader(new File("/invalid/file/location")))) {

int c;

// Read and display the file.

while ((c = reader.read()) != -1) {

System.out.println((char) c);

}

}

}

private static void exceptionWithoutHandler1() throws IOException {

exceptionWithoutHandler();

}

private static void exceptionWithoutHandler2() throws IOException {

exceptionWithoutHandler1();

}

private static void exceptionHandler() {

try {

exceptionWithoutHandler2();

} catch (IOException e) {

System.out.println("IOException caught!");

}

}

}

### Java finally Block

In the previous article, we have learned how to use **Java try/catch Block** to handle exceptions in Java. In this article, we will how and why to use finally block with examples.

## finally, Block Overview

* Java *finally*block is a block that is used to execute important code such as closing *connection*, *stream*etc.
* Java *finally*block is always executed whether an exception is handled or not.
* Java *finally*block follows **try/catch** block.
* For each try block, there can be zero or more catch blocks, but only one *finally*block.
* The *finally*block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

### finally Block Different Scenario Examples

Here is an example program that shows three methods that exit in various ways, none without executing their *finally*clauses:

Below example covers the below scenarios:

* Throw an exception out of the method
* Return from within a try block
* Execute a try block normally

// Demonstrate finally.

class FinallyDemo {

// Throw an exception out of the method.

static void procA() {

try {

System.out.println("inside procA");

throw new RuntimeException("demo");

} finally {

System.out.println("procA's finally");

}

}

// Return from within a try block.

static void procB() {

try {

System.out.println("inside procB");

return;

} finally {

System.out.println("procB's finally");

}

}

// Execute a try block normally.

static void procC() {

try {

System.out.println("inside procC");

} finally {

System.out.println("procC's finally");

}

}

public static void main(String args[]) {

try {

procA();

} catch (Exception e) {

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

}

procB();

procC();

}

}

Output:

inside procA

procA's finally

Exception caught

inside procB

procB's finally

inside procC

procC's finally

In this example, procA( ) prematurely breaks out of the *try*by throwing an exception. The *finally*clause is executed on the way out.

procB( )’s *try*statement is exited via a return statement. The *finally*clause is executed before procB( ) returns.

In procC( ), the try statement executes normally, without error. However, the *finally*block is still executed.

### finally, Block Examples

**Example 1:** In this example, we have used **FileInputStream** to read the simple.txt file. After reading a file the resource **FileInputStream** should be closed by using *finally*block.

public class FileInputStreamExample {

public static void main(String[] args) {

FileInputStream fis = null;

try {

File file = new File("sample.txt");

fis = new FileInputStream(file);

int content;

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

// convert to char and display it

System.out.print((char) content);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

if(fis != null){

try {

fis.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

}

**Example 2:** When we use JDBC connection to communicate with a database. The resource that we have used are *ResultSet*, *Statement*, and *Connection*(in that order) should be closed in a *finally*block when you are done with them, something like that:

Connection conn = null;

PreparedStatement ps = null;

ResultSet rs = null;

try {

// Do stuff

...

} catch (SQLException ex) {

// Exception handling stuff

...

} finally {

if (rs != null) {

try {

rs.close();

} catch (SQLException e) { /\* ignored \*/}

}

if (ps != null) {

try {

ps.close();

} catch (SQLException e) { /\* ignored \*/}

}

if (conn != null) {

try {

conn.close();

} catch (SQLException e) { /\* ignored \*/}

}

}