**Exception Handling**

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

* A user has entered an invalid data.
* A file that needs to be opened cannot be found.
* A network connection has been lost in the middle of communications or the JVM has run out of memory.

There are three categories of Exceptions. You need to understand them to know how exception handling works in Java.

1. **Checked Exceptions**
2. **Unchecked Exceptions**
3. **Errors**

**Checked exceptions** − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

import java.io.File;

import java.io.FileReader;

public class FilenotFound\_Demo {

public static void main(String args[]) {

File file = new File("E://file.txt");

FileReader fr = new FileReader(file);

}

}

If you try to compile the above program, you will get the following exceptions.

Output

C:\>javac FilenotFound\_Demo.java

FilenotFound\_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown

FileReader fr = new FileReader(file);

^

1 error

* **Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

public class Unchecked\_Demo {

public static void main(String args[]) {

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

System.out.println(num[5]);

}

}

If you compile and execute the above program, you will get the following exception.

Output

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5

at Exceptions.Unchecked\_Demo.main(Unchecked\_Demo.java:8)

* **Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

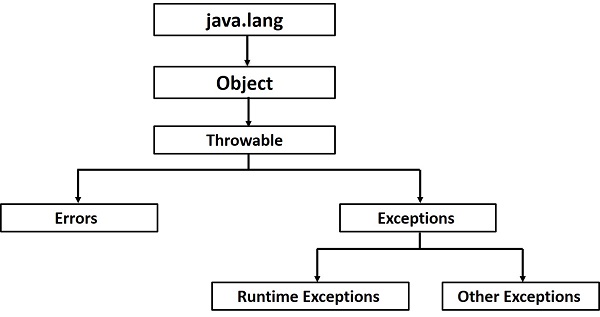
Exception Hierarchy

All exception classes are subtypes of the **java.lang.Exception class**. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses:

1. IOException class
2. RuntimeException Class.



Java being an object oriented programming language, whenever an error occurs while executing a statement, creates an exception object and then the normal flow of the program halts and [JRE](https://www.journaldev.com/546/difference-jdk-vs-jre-vs-jvm) tries to find someone that can handle the raised exception. The exception object contains a lot of debugging information such as method hierarchy, line number where the exception occurred, type of exception etc. When the exception occurs in a method, the process of creating the exception object and handing it over to runtime environment is called “throwing the exception”.

Once runtime receives the exception object, it tries to find the handler for the exception. Exception Handler is the block of code that can process the exception object. The logic to find the exception handler is simple – starting the search in the method where error occurred, if no appropriate handler found, then move to the caller method and so on. So if methods call stack is A->B->C and exception is raised in method C, then the search for appropriate handler will move from C->B->A. If appropriate exception handler is found, exception object is passed to the handler to process it. The handler is said to be “catching the exception”. If there are no appropriate exception handler found then program terminates printing information about the exception.

Note that Java Exception handling is a framework that is used to handle runtime errors only, compile time errors are not handled by exception handling in java.

We use specific keywords in java program to create an exception handler block, we will look into these keywords next.

Java Exception Handling Keywords

Java provides specific keywords for exception handling purposes, we will look after them first and then we will write a simple program showing how to use them for exception handling.

1. throw – We know that if any exception occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometime we might want to generate exception explicitly in our code, for example in a user authentication program we should throw exception to client if the password is null. throw keyword is used to throw exception to the runtime to handle it.
2. throws – When we are throwing any exception in a method and not handling it, then we need to use throws keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to it’s caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with [main()](https://www.journaldev.com/611/exception-in-thread-main-java) method also.
3. try-catch – We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.
4. finally – finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurred or not.

# Usage of try-catch-Finally

## Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

#### Syntax of Java try-catch

try{

//code that may throw exception

}catch(Exception\_class\_Name ref){}

#### Syntax of try-finally block

try{

//code that may throw exception

}finally{}

## Java 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.

## Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

public class Testtrycatch1

{

 public static void main(String args[]){

     int data=50/0;//may throw exception

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

}

}

**Output:**

**Exception in thread main java.lang.ArithmeticException:/ by zero**

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

## Solution by exception handling

Let's see the solution of above problem by java try-catch block.

public class Testtrycatch2{

  public static void main(String args[]){

   try{

      int data=50/0;

   }

catch(ArithmeticException e)

{

System.out.println(e);

}

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

}

}

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=Testtrycatch2)

**Output:**

**Exception in thread main java.lang.ArithmeticException:/ by zero**

**rest of the code...**

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

## 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.

Multiple catch blocks

Exception hierarchy contains many superclass exceptions and their subtypes/subclasses exceptions. Multiple catch blocks are used when we have to catch a specific type of exception, which is subclass of a general exception class. Let's see the use of multiple catch blocks with an example. 

### Example1.

### class MultipleCatch

### {

### public static void main(String... ar)

### {

### try

### {

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

### System.out.println("Value ="+arr[5]); *//ArrayIndexOutOfBoundsException thrown, invalid index 5.*

### }

### catch(ArrayStoreException exp) *//catch block to handle/catch ArrayStoreException*

### {

### System.out.println(exp);

### }

### catch(ArrayIndexOutOfBoundsException exp) *//catch block to handle/catch ArrayIndexOutOfBoundsException*

### {

### System.out.println("Exception Caught - "+ exp);

### }

### catch(Exception exp) *//catch block to handle/catch Exception* System.out.println(exp);

### }

### }

### }

### Output is -:

### Exception Caught - java.lang.ArrayIndexOutOfBoundsException: 5

### In this code, we have three catch blocks associated with one try block. At the runtime, code in try block throws an exception of type ArrayIndexOutOfBoundsException, this exception type is matched with the declared exception type in every catch-block(matching starts with the first catch block).  The catch block that matches with type of exception thrown is executed, while the rest of catch blocks are skipped. Let's see how -

### The first catch block declared an exception of type ArrayStoreException, which doesn't match with the internal working of try-catch blockexception ArrayOutOfBoundsException thrown by the try block, hence this catch-block is not executed.

### ArrayIndexOutOfBoundsException declared in the second catch block matches with the exception ArrayIndexOutOfBoundsException thrown by the try block and hence second catch block will be executed.

### As, we have already found the matching catch block, hence, the third catch block with declared exception of type Exception is skipped and not executed, even though Exception is superclass of ArrayIndexOutOfBoundsException class. Examlpe 2.

class Example2{

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...

**Lab Assignments**

10. Write a java program to implement exception handling using multiple catch statements. Also include code to identify the significance of finally block in handling exceptions.

11. Write a program to implement the concept of Exception Handling by creating user defined exceptions

Rethrowing exceptions

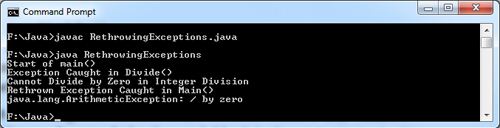
There might be situations in your program where you want to both catch an exception in your code and also want its caller be notified about the exception. This is possible by rethrowing the exception using throw statement.

# How to Rethrow an Exception in Java

An exception can be rethrown in a catch block. This action will cause the exception to be passed to the calling method. If the rethrow operation occurs in the main method then the exception is passed to the JVM and displayed on the console. The purpose of the rethrow operation is to get the attention of the outside world that an exception has occurred and at the same time perform any contingency logic (such as logging) in the catch block.

The following program demonstrates how an exception is rethrown using the throw statement.

*public class RethrowingExceptions   
{   
         static void divide()   
    {   
        int x,y,z;   
        try   
        {   
           x = 6 ;   
           y = 0 ;   
           z = x/y ;   
           System.out.println(x + "/"+ y +" = " + z);   
         }   
         catch(ArithmeticException e)   
         {   
          System.out.println("Exception Caught in Divide()");   
          System.out.println("Cannot Divide by Zero in Integer Division");****throw e; // Rethrows an exception****}   
     }   
      public static void main(String[] args)   
     {   
             System.out.println("Start of main()");   
             try   
            {   
                 divide();   
             }   
             catch(ArithmeticException e)   
            {   
                System.out.println("Rethrown Exception Caught in Main()");   
                System.out.println(e);   
            }   
      }   
 }*

[](http://ecomputernotes.com/images/Rethrowing-Exceptions.jpg)

**Built-in Exceptions**

Java defines several other types of exceptions that relate to its various class libraries.

**Following is the list of Java Unchecked RuntimeException.**

|  |  |
| --- | --- |
| **Sr.No.** | **Exception & Description** |
| 1 | **ArithmeticException**  Arithmetic error, such as divide-by-zero. |
| 2 | **ArrayIndexOutOfBoundsException**  Array index is out-of-bounds. |
| 3 | **ArrayStoreException**  Assignment to an array element of an incompatible type. |
| 4 | **ClassCastException**  Invalid cast. |
| 5 | **IllegalArgumentException**  Illegal argument used to invoke a method. |
| 6 | **IllegalMonitorStateException**  Illegal monitor operation, such as waiting on an unlocked thread. |
| 7 | **IllegalStateException**  Environment or application is in incorrect state. |
| 8 | **IllegalThreadStateException**  Requested operation not compatible with the current thread state. |
| 9 | **IndexOutOfBoundsException**  Some type of index is out-of-bounds. |
| 10 | **NegativeArraySizeException**  Array created with a negative size. |
| 11 | **NullPointerException**  Invalid use of a null reference. |
| 12 | **NumberFormatException**  Invalid conversion of a string to a numeric format. |
| 13 | **SecurityException**  Attempt to violate security. |
| 14 | **StringIndexOutOfBounds**  Attempt to index outside the bounds of a string. |
| 15 | **UnsupportedOperationException**  An unsupported operation was encountered. |

**Checked Exceptions**

**Following is the list of Java Checked Exceptions Defined in java.lang.**

|  |  |
| --- | --- |
| **Sr.No.** | **Exception & Description** |
| 1 | **ClassNotFoundException**  Class not found. |
| 2 | **CloneNotSupportedException**  Attempt to clone an object that does not implement the Cloneable interface. |
| 3 | **IllegalAccessException**  Access to a class is denied. |
| 4 | **InstantiationException**  Attempt to create an object of an abstract class or interface. |
| 5 | **InterruptedException**  One thread has been interrupted by another thread. |
| 6 | **NoSuchFieldException**  A requested field does not exist. |
| 7 | **NoSuchMethodException**  A requested method does not exist. |

**Creating own exception subclasses (user defined Exceptions)**

**Steps to write own exception class**

* Create a new class whose name should end with Exception like ClassNameException. This is a convention to differentiate an exception class from regular ones.
* Make the class extends one of the exceptions which are subtypes of the java.lang.Exception class. Generally, a custom exception class always extends directly from the Exception class.
* Create a constructor with a String parameter which is the detail message of the exception. In this constructor, simply call the super constructor and pass the message.

**The following is a custom exception class which is created by following the above steps:**

|  |  |
| --- | --- |
|  | public class StudentNotFoundException extends Exception {        public StudentNotFoundException(String message) {          super(message);      }  } |

**And the following example shows the way a custom exception is used is nothing different than built-in exception:**

|  |  |
| --- | --- |
|  | public class StudentManager {        public Student find(String studentID) throws StudentNotFoundException  {          if (studentID.equals("123456"))  {              return new Student();          } else  {              throw new StudentNotFoundException( "Could not find student with ID " + studentID);          }      }  } |

**And the following test program handles that exception:**

|  |  |
| --- | --- |
|  | public class StudentTest {      public static void main(String[] args) {          StudentManager manager = new StudentManager();            try {                Student student = manager.find("0000001");            } catch (StudentNotFoundException ex) {              System.err.print(ex);          }      }  } |

Run this program and you will see this output:

|  |  |
| --- | --- |
|  | StudentNotFoundException: Could not find student with ID 0000001 |

**Multithreading**

Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

Thread

A single thread is basically a lightweight and the smallest unit of processing. Java uses threads by using a "Thread Class".

There are two types of thread – **user thread and daemon thread**(daemon threads are used when we want to clean the application and are used in the background).

When an application first begins, user thread is created. Post that, we can create many user threads and daemon threads.

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

**Single Thread Example:**

package demotest;

public class Thread\_demo

{

public static void main(String[] args) {

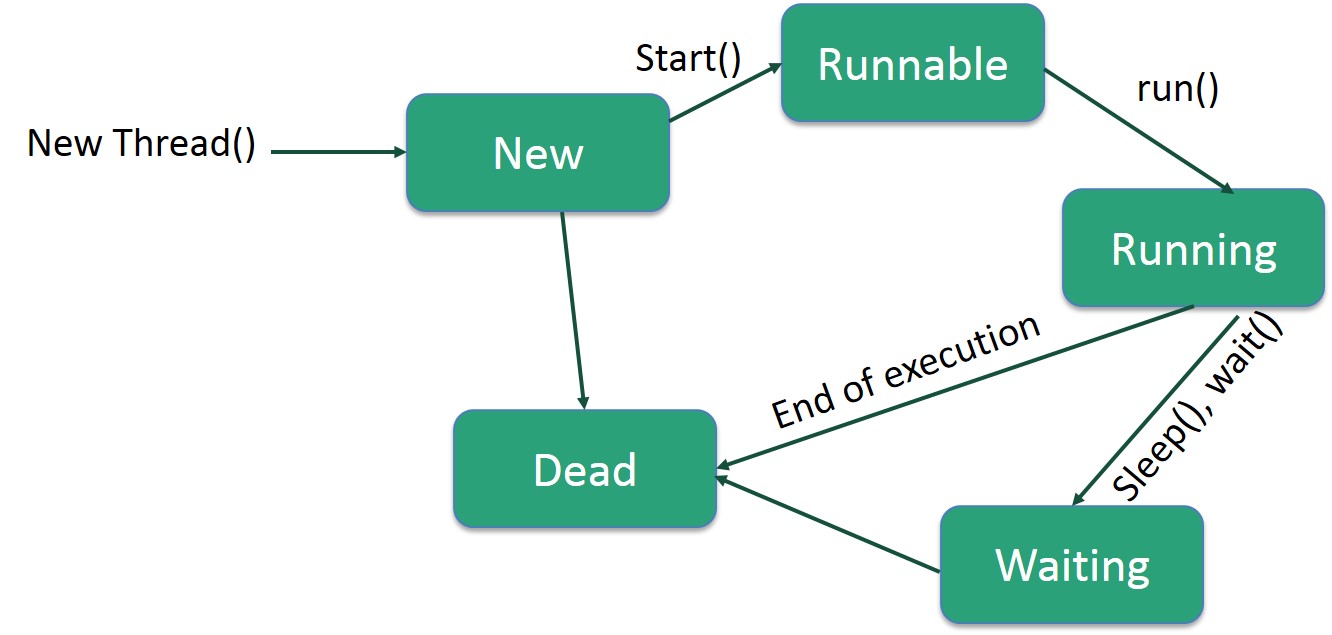
System.out.println("Single Thread");

}

}

**Life Cycle of a Thread**

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle −

* **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
* **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
* **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
* **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
* **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.