In this article, 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;

}