**Try, Catch, Finally In Java**

**The below keywords are used in Java for Exception handling.**

* Try
* Catch
* Finally
* Throw
* Throws

**The following table briefly describes these keywords.**

| **Keyword** | **Description** |
| --- | --- |
| **Try** | We specify the block of code that might give rise to the exception in a special block with a “Try” keyword. |
| **Catch** | When the exception is raised it needs to be caught by the program. This is done using a “catch” keyword. So a catch block follows the try block that raises an exception. The keyword catch should always be used with a try. |
| **Finally** | Sometimes we have an important code in our program that needs to be executed irrespective of whether or not the exception is thrown. This code is placed in a special block starting with the “Finally” keyword. The Finally block follows the Try-catch block. |
| **Throw** | The keyword “throw” is used to throw the exception explicitly. |
| **Throws** | The keyword “Throws” does not throw an exception but is used to declare exceptions. This keyword is used to indicate that an exception might occur in the program or method. |

In this tutorial, we will discuss all the above keywords in detail along with the programming examples.

**Try Block In Java**

Whenever we are writing a program there could be a code that we suspect might throw an exception. **For example,** we might suspect that there might be a “division by zero” operation in the code that will throw an exception.

This code that might raise an exception is enclosed in a block with the keyword “try”. So the try block contains the code or set of statements that can raise an exception.

**The general syntax of the try block is as follows:**

*try{*

*//set of statements that can raise exception*

*}*

Hence, if a programmer thinks that certain statements will raise exceptions, then enclose these statements in a try block. Note that when an exception occurs at a specific statement in a try block, then the rest of the code is not executed.

When an exception occurs in a try block at a particular statement, then the control comes out and the program terminates abruptly. To prevent this abrupt termination of the program, we should “handle” this exception. This handling is done using the “catch” keyword. So a try block always has a catch block following it.

**Catch Block In Java**

We use a catch block to handle exceptions. This is the block with the “catch” keyword. The catch block follows the try block.

Whenever an exception occurs in the try block, then the code in the catch block that corresponds to the exception is executed.

**The general syntax of the catch block is:**

catch (Exception e){

//code to handle exception e

}

Generally, the declared exception must be the parent class of all exceptions, i.e. Exception. But if there is more than one exception, we can also write specific exception types or generated exceptions.

Next, we will discuss the try-catch block. Note that for each try block, we can have multiple catch blocks.

**Try-Catch Java**

**The general syntax of the try-catch block is shown below:**

try{

//code causing exception

}

catch (exception (exception\_type) e (object))

{

//exception handling code

}

The try block can have multiple lines of code that can raise multiple exceptions. Each of these exceptions is handled by an independent catch block.

The generic exception handler, object e of the Exception class can handle all the exceptions but if we want to handle specific exceptions, then it is advisable to specify the generic exception handler as the last catch block.

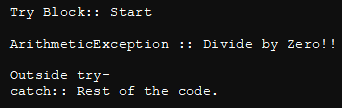
**Java Try Catch Example**

Now let’s demonstrate a try-catch block in Java. Here in the try block, we define a division operation. The divisor is zero. Thus the statement that divides the two numbers raises an Arithmetic exception. We have a catch block that defines a handler for the Arithmetic exceptions.

**Given below is an example Java program.**

|  |
| --- |
| class Main {     public static void main(String args[]) {        int val1, val2;        try {           //this block will contain statements that may raise exceptions           System.out.println("Try Block:: Start");           val1 = 0;           val2 = 25 / val1;           System.out.println(val2);           System.out.println("Try Block:: End");        }        catch (ArithmeticException e) {           //handler for ArithmeticException           System.out.println("ArithmeticException :: Divide by Zero!!");        }        System.out.println("Outside try-catch:: Rest of the code.");     }  } |

**Output**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/06/1-9.png)

**Catching Multiple Exceptions**

As already mentioned, a try block can contain a code that raises more than one exception. In this case, we will need more than one catch block to handle each exception. A single try block can be followed by multiple catch blocks. Each catch block will handle the independent exceptions.

**In the case of multiple catch blocks, we have to remember the below points:**

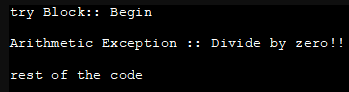
* In a Java program, at any instance of time, only one exception can occur. Also, at any point, only one catch block is executed.
* The multiple catch blocks should be ordered in such a way that the catch block for most specific exceptions should come first and then the general.

**For example,** if we have ArithmeticException and general Exception, then the catch block handling ArithmeticException will come first followed by the catch block handling Exception.

**The below example demonstrates multiple catch blocks.**

|  |
| --- |
| public class Main {      public static void main(String[] args) {          //try block containing exception prone code          try{              System.out.println("try Block:: Begin");              int myArray[]=new int[5];              myArray [5]=10/0;          }          //multiple catch blocks          catch(ArithmeticException e)          {              System.out.println("Arithmetic Exception :: Divide by zero!!");          }          catch(ArrayIndexOutOfBoundsException e)          {              System.out.println("ArrayIndexOutOfBounds :: Accessed index out of bounds");          }          catch(Exception e)          {              System.out.println("Exception :: " + e.getMessage ());          }          System.out.println("rest of the code");      }  } |

**Output**

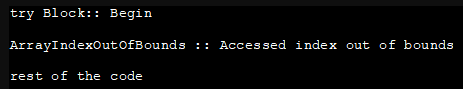
[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/06/2-7.png)

In the above program, an ArithmeticException that is caught in the first catch block is raised. If this catch block was not specified, then the exception would have propagated to the generalized catch block.

Let’s slightly modify the above program so that the try block raises two exceptions. Now let us see the output.

|  |
| --- |
| public class Main {      public static void main(String[] args) {          //try block containing exception prone code          try{              System.out.println("try Block:: Begin");              int myArray[]=new int[5];              System.out.println("Array element 10 : " + myArray[10]);              myArray[5]=10/0;          }          //multiple catch blocks          catch(ArithmeticException e)          {              System.out.println("Arithmetic Exception :: Divide by zero!!");          }          catch(ArrayIndexOutOfBoundsException e)          {              System.out.println("ArrayIndexOutOfBounds :: Accessed index out of bounds");          }          catch(Exception e)          {              System.out.println("Exception :: " + e.getMessage ());          }          System.out.println("rest of the code");      }  } |

**Output**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/06/3-8.png)

If we see this output, it shows ArrayIndexOutOfBoundsException being thrown. This is because the statement that raises ArrayIndexOutOfBoundsException is executed first. The exception is thrown and the control goes to the corresponding catch block.

**Nested Try-Catch**

A try block inside another try block is called a nested try block. We need such structures in certain situations when a piece of code contained in a try code may be such that some lines raise certain exceptions and another piece of code raises a completely different exception.

In the case of nested try blocks, first, the innermost try block is executed and the exception is handled. If the innermost try block does not have a matching catch block, then it is propagated one level up to its parent try block. This way the exception is propagated upwards till a matching exception handler is found.

If there is no exception handler matching the exception, then the program is abruptly terminated with a system-generated message.

**The general syntax of a nested try block is given below:**

try

{

//try\_block 1;

try

{

//try\_block 2;

}

catch(Exception e)

{

//exception handler code

}

}

catch(Exception e)

{

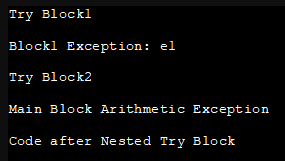
//exception handler code

}

**Let’s implement a program to demonstrate the nested try-catch block.**

|  |
| --- |
| class Main{     public static void main(String args[]){       //Main try block       try{          //try block1           try{              System.out.println("Try Block1");              int num =15/0;              System.out.println(num);           }           catch(ArithmeticException e1){              System.out.println("Block1 Exception: e1");           }           //try block2           try{              System.out.println("Try Block2");              int num =100/0;              System.out.println(num);           }           catch(ArrayIndexOutOfBoundsException e2){              System.out.println("Block2 Exception: e2");           }          System.out.println("General statement after Block1 and Block2");      }      catch(ArithmeticException e3){           System.out.println("Main Block Arithmetic Exception");      }      catch(ArrayIndexOutOfBoundsException e4){          System.out.println("Main Block ArrayIndexOutOfBoundsException");      }      catch(Exception e5){          System.out.println("Main Block General Exception");      }      System.out.println("Code after Nested Try Block");    }  } |

**Output**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/06/4-6.png)

In the program above, we have two try blocks enclosed in the main try block. Both the inner try blocks have a code that raises ArithmeticException. But we have provided a matching catch block only for the first block and not for the second try block.

Hence the second block propagates its exception to the main try block and then handles it. This is evident from the output.

**Finally Block In Java**

So far we have seen the try-catch and nested try block. We know that the code that is expected to raise the exception is put in a try block. When an exception occurs, then the remainder of the code in the try block is not executed.

Either the program terminates abruptly if an exception is not handled or the control is passed to the exception handler.

In such a situation, there arises a need to include a code that is to be executed irrespective of whether an exception occurs or not. This means we will execute a piece of code even when an exception occurs and also when the exception does not occur.

But as try block exits after the exception is raised we cannot put this code in the try block. Similarly, the catch block has an exception handler, so we cannot put this in the catch block too.

Thus we need a separate block that contains a code that executes irrespective of whether an exception occurs or not. Java provides a “finally” block that contains this piece of code.

Hence a finally block in Java can contain critical statements that are to be executed in the program. The execution of these statements should be carried out even when an exception occurs or not.

Therefore, we will put code like closing connections, stream objects, etc. or any cleanup code in the finally block so that they can be executed even if an exception occurs.

The finally block in Java is usually put after a try or catch block. Note that the finally block cannot exist without a try block. When the finally block is included with try-catch, it becomes a “**try-catch-finally**” block.

We can skip the finally block in the exception handling code. This means that finally block is optional.

If the try block does not raise any exception then the finally block will be executed after the try block. If there is an exception in the try block then control will pass to the catch block first and then the finally block.

An exception occurring in finally block behaves in the same way as any other exception. Even if the try block contains a return statement or branching statements like break and continue, then the finally block will still be executed.

Keeping these points in mind, let’s go ahead with the general syntax and examples of finally block.

**The general syntax of the finally block is as follows:**

try {

</em><em>                         //code that might raise exception

</em><em>            }catch {

</em><em>                       //code that handles exception

</em><em>            }finally {

</em><em>                          //mandatory code to be executed

</em><em>            }

Though finally block is always executed, there are certain situations or cases in which it doesn’t execute.

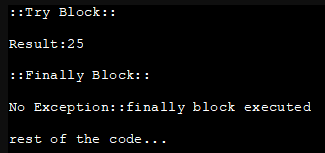
**These are the below cases:**

* When the thread is dead.
* When the System.exit() method is used.
* When an exception occurs in the finally block.

**Let’s implement a couple of programs to demonstrate the finally block.**

|  |
| --- |
| class Main {      public static void main (String args[]) {          //try block          try          {              System.out.println ("::Try Block::");              int data = 125 / 5;              System.out.println ("Result:" + data);          }          //catch block          catch (NullPointerException e) {              System.out.println ("::Catch Block::");              System.out.println (e);          }          //finally block          finally {              System.out.println (":: Finally Block::");              System.out.println ("No Exception::finally block executed");          }          System.out.println ("rest of the code...");      }  } |

**Output**

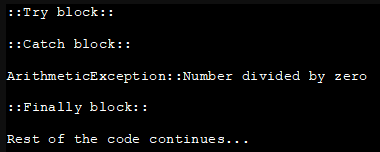
[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/06/5-5.png)

The above program shows a try-catch-finally block. In the try block, a valid operation is performed and hence no exception is thrown. Therefore, the control is not passed to catch from try but to finally block.

The following program is another example of try-catch-finally block but in this case, the exception is thrown in the try block as we perform a divide by zero operation. Thus the finally block is executed after the try the catch block is executed.

|  |
| --- |
| class Main {     public static void main(String args[]) {       //try block       try{              System.out.println("::Try block::");              int num=67/0;              System.out.println(num);        }        //catch block        catch(ArithmeticException e){            System.out.println("::Catch block::");           System.out.println("ArithmeticException::Number divided by zero");        }        //finally block        finally{            System.out.println("::Finally block::");        }        System.out.println("Rest of the code continues...");     }  } |

**Output**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/06/6-5.png)

**Throw An Exception In Java**

Java provides a keyword “throw” using which we can explicitly throw the exceptions in the code. **For example,** if we are checking arithmetic operations and want to raise some exceptions after checking operands we can do so using the ‘throw’ keyword.

Using the throw keyword, we can throw the checked or unchecked exceptions. The throw keyword is also used to throw custom exceptions.

**The general syntax of the throw keyword is:**

*throw exception;*

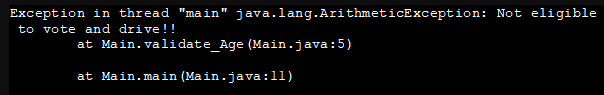
or

*throw new exception\_class("error message");*

**Given below is an example program to demonstrate the throw keyword.**

|  |
| --- |
| public class Main{     static void validate\_Age(int age){       //if specified age is &lt; 18, throw ArithmeticException       if(age&lt;18)        throw new ArithmeticException("Not eligible to vote and drive!!");       else   //print the message        System.out.println("Eligible to vote and drive!!");     }     public static void main(String args[]){        //call validate\_Age method        validate\_Age(10);        System.out.println("rest of the code...");    }  } |

**Output**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/06/7-3.png)

In the above program, we use a method to validate age. If the age is < 18, then an exception is thrown to indicate the age is not valid.

**Throws Clause**

We have seen try block to declare exceptions. It contains the code that may raise exceptions. There is another way to declare an exception and it is using the “throws” keyword.

The declaration of exception using the “throws” keyword tells the programmer that there may be an exception specified after the “throws” keyword and the programmer should provide corresponding handler code for this exception to maintain the normal flow of the program.

However, the question arises as to why we need a “throws” keyword when we have a more reliable try-catch block to declare and handle exceptions.

One reason is as the number of exceptions that might possibly occur increases, the number of catch block that handles exceptions also increases as one catch block can handle only one exception.

Similarly, if there are many methods in a program and each method has numerous exceptions then the code will become unnecessarily long and unmanageable.

Thus declaring an exception with the throws keyword in the method signature and then handling the method call using try-catch seems to be a viable solution.

Another advantage of declaring exceptions using the throws keyword is that we are forced to handle the exceptions. If we do not provide a handler for a declared exception then the program will raise an error.

**The general syntax of the throws keyword is as follows:**

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

*//method code*

*}*

**Let us now implement a Java program to demonstrate the “throws” keyword.**

In this program, we have a class Example\_throw in which we have a method testMethod. In the signature of this testMethod, we declare two exceptions IOException and Arithmetic Exception using the throws keyword. Then in the main function, the exceptions thrown are handled by the catch block.

|  |
| --- |
| import java.io.\*;  class Example\_Throw {      //declare exception using throws in the method signature    void testMethod(int num) throws IOException, ArithmeticException{       if(num==1)          throw new IOException("IOException Occurred");       else          throw new ArithmeticException("ArithmeticException");    }  }class Main{    public static void main(String args[]){     try{         //this try block calls the above method so handle the exception       Example\_Throw obj=new Example\_Throw();       obj.testMethod(1);     }catch(Exception ex){       System.out.println(ex);      }    }  } |

**Output**

[java.io.IOException](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/06/8-4.png)