Java try...catch

In this tutorial, we will learn about the try catch statement in Java with the help of examples.

The `try...catch` block in Java is used to handle exceptions and prevents the abnormal termination of the program.

Here's the syntax of a `try...catch` block in Java.

```
try{
  // code
}
catch(exception) {
  // code
}
```

The `try` block includes the code that might generate an exception.

The `catch` block includes the code that is executed when there occurs an exception inside the `try` block.

Example: Java try...catch block

```
class Main {
  public static void main(String[] args) {

    try {
      int divideByZero = 5 / 0;
      System.out.println("Rest of code in try block");
    }

    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
    }
  }
}
```

Output

```
ArithmeticException => / by zero
```

In the above example, notice the line,

```
int divideByZero = 5 / 0;
```

Here, we are trying to divide a number by **zero**. In this case, an exception occurs. Hence, we have enclosed this code inside the `try` block.

When the program encounters this code, `ArithmeticException` occurs. And, the exception is caught by the `catch` block and executes the code inside the `catch` block.

The `catch` block is only executed if there exists an exception inside the `try` block.

> **Note**: In Java, we can use a `try` block without a `catch` block. However, we cannot use a `catch` block without a `try` block.

Java try...finally block

We can also use the `try` block along with a finally block.

In this case, the finally block is always executed whether there is an exception inside the try block or not.

Example: Java try...finally block

```java
class Main {
  public static void main(String[] args) {
    try {
      int divideByZero = 5 / 0;
    }

    finally {
      System.out.println("Finally block is always executed");
    }
  }
}
```

Output

```
Finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:4)
```

In the above example, we have used the `try` block along with the `finally` block. We can see that the code inside the `try` block is causing an exception.

However, the code inside the `finally` block is executed irrespective of the exception.

Java try...catch...finally block

In Java, we can also use the finally block after the `try...catch` block. For example,

```
import java.io.*;

class ListOfNumbers {

  // create an integer array
  private int[] list = {5, 6, 8, 9, 2};

  // method to write data from array to a fila
  public void writeList() {
    PrintWriter out = null;

    try {
      System.out.println("Entering try statement");

      // creating a new file OutputFile.txt
      out = new PrintWriter(new FileWriter("OutputFile.txt"));

      // writing values from list array to Output.txt
      for (int i = 0; i < 7; i++) {
        out.println("Value at: " + i + " = " + list[i]);
      }
    }

    catch (Exception e) {
      System.out.println("Exception => " + e.getMessage());
    }

    finally {
      // checking if PrintWriter has been opened
      if (out != null) {
        System.out.println("Closing PrintWriter");
        // close PrintWriter
        out.close();
      }

      else {
        System.out.println("PrintWriter not open");
      }
    }

  }
}

class Main {
  public static void main(String[] args) {
    ListOfNumbers list = new ListOfNumbers();
    list.writeList();
  }
}
```

Output

```
Entering try statement
Exception => Index 5 out of bounds for length 5
Closing PrintWriter
```

In the above example, we have created an array named `list` and a file named `output.txt`. Here, we are trying to read data from the array and storing to the file.

Notice the code,

```
for (int i = 0; i < 7; i++) {
  out.println("Value at: " + i + " = " + list[i]);
}
```

Here, the size of the array is `5` and the last element of the array is at `list[4]`. However, we are trying to access elements at `a[5]` and `a[6]`.

Hence, the code generates an exception that is caught by the catch block.

Since the `finally` block is always executed, we have included code to close the `PrintWriter` inside the finally block.

It is a good practice to use finally block to include important cleanup code like closing a file or connection.

> **Note**: There are some cases when a `finally` block does not execute:
>
> - Use of `System.exit()` method
> - An exception occurs in the `finally` block
> - The death of a thread

Multiple Catch blocks

For each `try` block, there can be zero or more `catch` blocks. Multiple `catch` blocks allow us to handle each exception differently.

The argument type of each `catch` block indicates the type of exception that can be handled by it. For example,

```
class ListOfNumbers {
  public int[] arr = new int[10];

  public void writeList() {

    try {
      arr[10] = 11;
    }

    catch (NumberFormatException e1) {
      System.out.println("NumberFormatException => " + e1.getMessage());
    }

    catch (IndexOutOfBoundsException e2) {
      System.out.println("IndexOutOfBoundsException => " + e2.getMessage());
    }

  }
}

class Main {
  public static void main(String[] args) {
    ListOfNumbers list = new ListOfNumbers();
    list.writeList();
  }
}
```

Output

```
IndexOutOfBoundsException => Index 10 out of bounds for length 10
```

In this example, we have created an integer array named `arr` of size **10**.

Since the array index starts from **0**, the last element of the array is at `arr[9]`. Notice the statement,

```
arr[10] = 11;
```

Here, we are trying to assign a value to the index **10**. Hence, `IndexOutOfBoundException` occurs.

When an exception occurs in the `try` block,

- The exception is thrown to the first `catch` block. The first `catch` block does not handle an `IndexOutOfBoundsException`, so it is passed to the next `catch` block.
- The second `catch` block in the above example is the appropriate exception handler because it handles an `IndexOutOfBoundsException`. Hence, it is executed.

Catching Multiple Exceptions

From Java SE 7 and later, we can now catch more than one type of exception with one `catch` block.

This reduces code duplication and increases code simplicity and efficiency.

Each exception type that can be handled by the `catch` block is separated using a vertical bar `|`.

Its syntax is:

```
try {
  // code
} catch (ExceptionType1 | Exceptiontype2 ex) {
  // catch block
}
```

Java try-with-resources statement

The **try-with-resources** statement is a try statement that has one or more resource declarations.

Its syntax is:

```
try (resource declaration) {
  // use of the resource
} catch (ExceptionType e1) {
  // catch block
}
```

The resource is an object to be closed at the end of the program. It must be declared and initialized in the try statement.

Let's take an example.

```
try (PrintWriter out = new PrintWriter(new FileWriter("OutputFile.txt")) {
  // use of the resource
}
```

The **try-with-resources** statement is also referred to as **automatic resource management**. This statement automatically closes all the resources at the end of the statement.