



Module 5:

Java Exception Handling

Prof. Pramod Bide

1. Java try...catch block

The [try-catch](#) block is used to handle exceptions in Java. Here's the syntax of try...catch block:

```
try {  
    // code  
}  
catch(Exception e) {  
    // code  
}
```

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by a catch block. When an exception occurs, it is caught by the catch block. The catch block cannot be used without the try block.

Example: Exception handling using try...catch

```
class Main { public static void main(String[] args) {  
  
    try {  
        // code that generate exception  
  
        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 example, we are trying to divide a number by 0. Here, this code generates an exception.

To handle the exception, we have put the code, `5 / 0` inside the try block.

Now when an exception occurs, the rest of the code inside the try block is skipped.

The catch block catches the exception and statements inside the catch block is executed.

If none of the statements in the try block generates an exception, the catch block is skipped.

2. Java finally block

In Java, the finally block is always executed no matter whether there is an exception or not.

The finally block is optional. And, for each try block, there can be only one finally block.

```
try {  
    //code  
}  
catch (ExceptionType1 e1) {  
    // catch block  
}  
finally {  
    //  
    finally block always executes  
}
```

If an exception occurs, the finally block is executed after the try...catch block. Otherwise, it is executed after the try block. For each try block, there can be only one finally block.

Example: Java Exception Handling using finally block

```
class Main {  
    public static void main(String[] args) {  
  
        try { // code that generates exception  
            int divideByZero = 5 / 0;  
        }  
        catch (ArithmeticException e) {  
            System.out.println("ArithmeticException => " + e.getMessage());  
        }  
        finally {  
            System.out.println("This is the finally block");  
        }  
    }  
}
```

Output

```
ArithmeticException => / by zero This is the finally block
```

In the above example, we are dividing a number by **0** inside the try block. Here, this code generates an `ArithmeticException`. The exception is caught by the catch block. And, then the finally block is executed.

Note: It is a good practice to use the finally block. It is because it can include important cleanup code that might be accidentally skipped by `return`, [continue](#) or [break](#)

- closing a file or connection

Output

```
ArithmeticException => / by zero This is the finally block
```

In the above example, we are dividing a number by **0** inside the try block. Here, this code generates an `ArithmeticException`. The exception is caught by the catch block. And, then the finally block is executed.

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

3. Java throw and throws keyword

The Java throw keyword is used to explicitly throw a single exception. When we throw an exception, the flow of the program moves from the try block to the catch block.

Example: Exception handling using Java throw

```
class Main {  
    public static void divideByZero() {  
  
        // throw an exception  
        throw new ArithmeticException("Trying to divide by 0");  
    }  
  
    public static void main(String[] args) {  
  
        divideByZero();  
  
    }  
}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0  
at Main.divideByZero(Main.java:5)  
at Main.main(Main.java:9)
```

In the above example, we are explicitly throwing the `ArithmeticException` using the `throw` keyword.

Similarly, the `throws` keyword is used to declare the type of exceptions that might occur within the method. It is used in the method declaration.

Example: Java throws keyword

```
import java.io.*;
class Main {
    // declaring the type of exception
    public static void findFile() throws IOException {

        // code that may generate IOException
        File newFile = new File("test.txt");
        FileInputStream stream = new FileInputStream(newFile);
    } public static void main(String[] args) {
        try {
            findFile();
        }
        catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

Output

```
java.io.FileNotFoundException: test.txt (The system  
cannot find the file specified)
```

When we run this program, if the file **test.txt** does not exist, `FileInputStream` throws a `FileNotFoundException` which extends the `IOException` class.

The `findFile()` method specifies that an `IOException` can be thrown.

The `main()` method calls this method and handles the exception if it is thrown.

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the `throws` clause.

Output

When we run this program, if the file **test.txt** does not exist, FileInputStream throws a FileNotFoundException which extends the IOException class.

The findFile() method specifies that an IOException can be thrown.

The main() method calls this method and handles the exception if it is thrown.

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the throws clause.

Best Practices in Exception Handling

Informational Error Messages

Providing clear and informative error messages helps users and developers understand the cause of exceptions, aiding in efficient debugging and issue resolution.

Custom Exception Classes

Developing custom exception classes enables the representation of specific error scenarios, facilitating targeted exception handling and management.

Logging and Tracing

Implementing logging and tracing mechanisms assists in capturing and reporting exceptions, supporting post-incident analysis and system enhancement.