Exceptions in Java

Difficulty Level : Easy Last Updated : 10 Feb, 2022

Exception Handling in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

What is an Exception?
An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception such as the name and description of the exception and the state of the program when the exception occurred.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

What is an Error?
Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

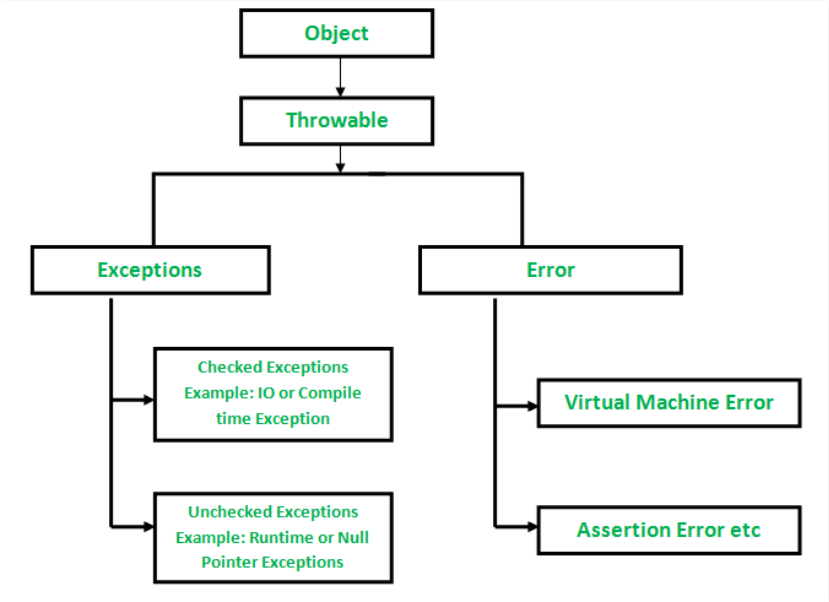
Errors are usually beyond the control of the programmer and we should not try to handle errors.

Error vs Exception

- **Error:** An Error indicates a serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

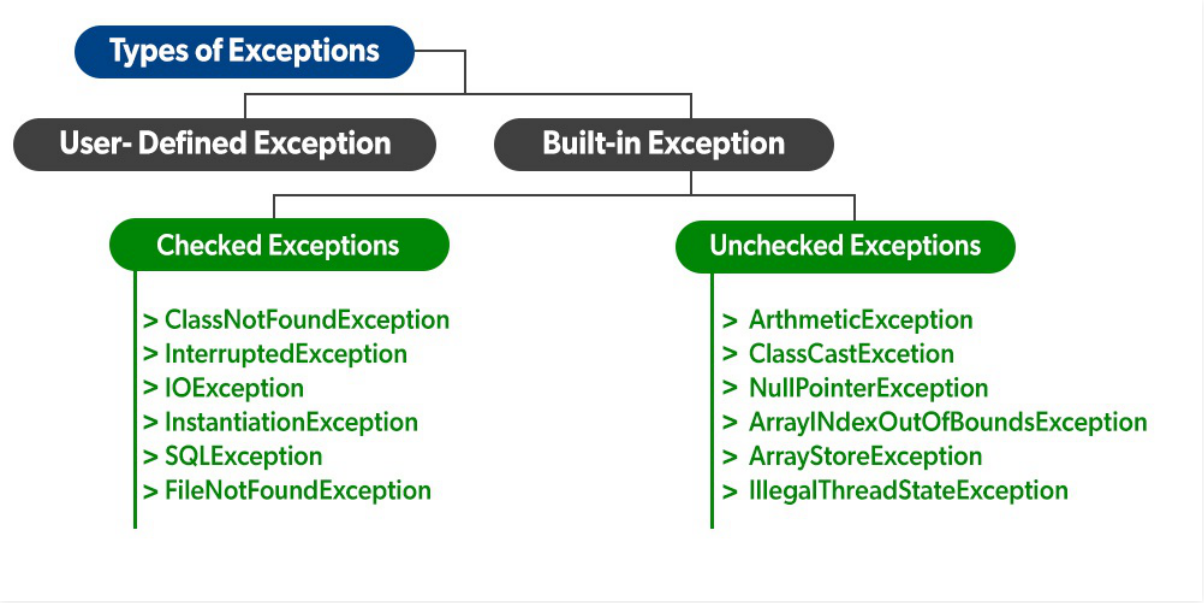
Exception Hierarchy

All exception and errors types are subclasses of class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be Categorized into 2 Ways:

1. Built-in Exceptions
   - Checked Exception
   - Unchecked Exception

2. User-Defined Exceptions

1. Built-in Exceptions: Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Note: *For checked vs unchecked exception, see Checked vs Unchecked Exceptions*

2. User-Defined Exceptions: Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions which are called 'user-defined Exceptions'.

The advantages of Exception Handling in Java are as follows:

- Provision to Complete Program Execution
- Easy Identification of Program Code and Error-Handling Code
- Propagation of Errors
- Meaningful Error Reporting
- Identifying Error Types

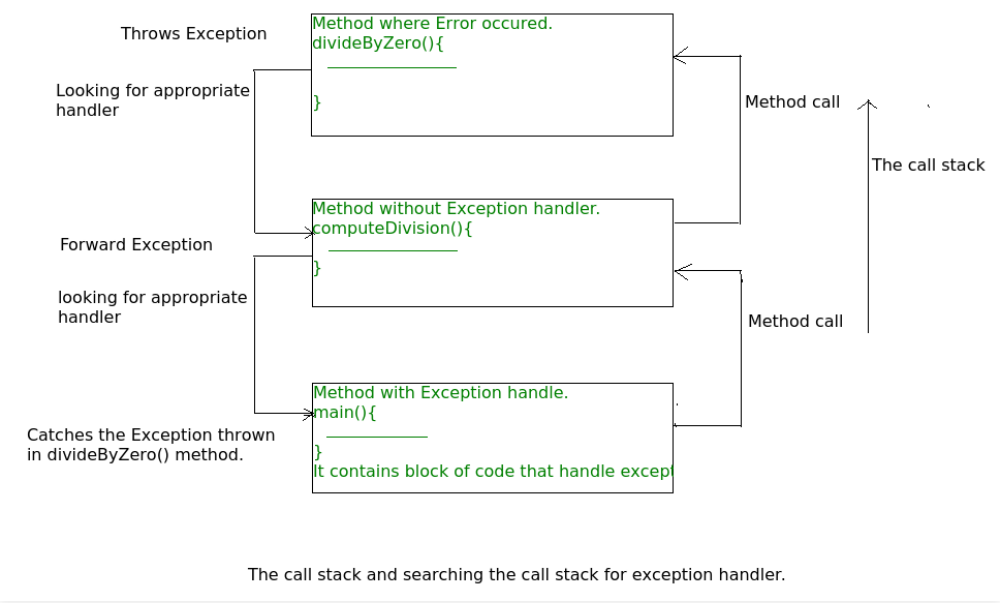
How does JVM handle an Exception?

Default Exception Handling: Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of the methods is called **Call Stack**. Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an **Exception handler**.
- The run-time system starts searching from the method in which the exception occurred, proceeds through the call stack in the reverse order in which methods were called.
- If it finds an appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program **abnormally**.

```
Exception in thread "xxx" Name of Exception : Description
... ..... ..  // Call Stack
```

See the below diagram to understand the flow of the call stack.



The call stack and searching the call stack for exception handler.

Example:

```
// Java program to demonstrate how exception is thrown.

class ThrowsExecp{

    public static void main(String args[]){

        String str = null;
        System.out.println(str.length());

    }
}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException
    at ThrowsExecp.main(File.java:8)
```

Let us see an example that illustrates how a run-time system searches appropriate exception handling code on the call stack :

```
// Java program to demonstrate exception is thrown
// how the runTime system searches th call stack
// to find appropriate exception handler.

class ExceptionThrown
{
    // It throws the Exception(ArithmeticException).
    // Appropriate Exception handler is not found within this method.
    static int divideByZero(int a, int b){

        // this statement will cause ArithmeticException(/ by zero)
        int i = a/b;

        return i;
    }

    // The runTime System searches the appropriate Exception handler
    // in this method also but couldn't have found. So looking forward
    // on the call stack.
    static int computeDivision(int a, int b) {

        int res =0;

        try
        {
        res = divideByZero(a,b);
        }
        // doesn't matches with ArithmeticException
        catch(NumberFormatException ex)
        {
        System.out.println("NumberFormatException is occured");
        }
        return res;
    }

    // In this method found appropriate Exception handler.
    // i.e. matching catch block.
    public static void main(String args[]){

        int a = 1;
        int b = 0;

        try
        {
            int i = computeDivision(a,b);

        }

        // matching ArithmeticException
        catch(ArithmeticException ex)
        {
            // getMessage will print description of exception(here / by zero)
            System.out.println(ex.getMessage());
        }
    }
```

```
}
```

Output

```
/ by zero
```

How Programmer handles an Exception?

Customized Exception Handling: Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

Detailed Article: Control flow in try catch finally block

Need of try-catch clause(Customized Exception Handling)

Consider the following Java program.

```
// java program to demonstrate
// need of try-catch clause

class GFG {
    public static void main(String[] args)
    {
        // array of size 4.
        int[] arr = new int[4];

        // this statement causes an exception
        int i = arr[4];
```

```
        // the following statement will never execute
        System.out.println("Hi, I want to execute");
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at GFG.main(GFG.java:9)
```

Explanation: In the above example an array is defined with size i.e. you can access elements only from index 0 to 3. But you trying to access the elements at index 4(by mistake) that's why it is throwing an exception. In this case, JVM terminates the program **abnormally**. The statement *System.out.println("Hi, I want to execute");* will never execute. To execute it, we must handle the exception using try-catch. Hence to continue the normal flow of the program, we need a try-catch clause.

How to use try-catch clause?

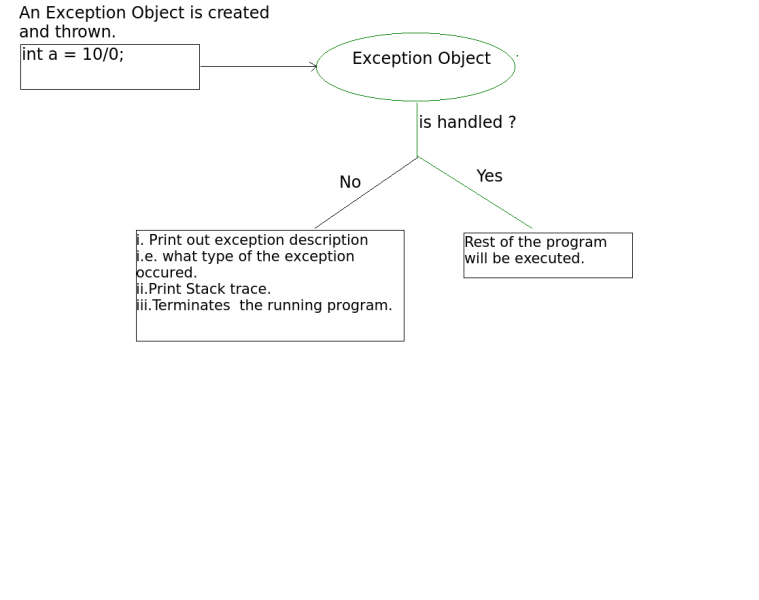
```
try {
    // block of code to monitor for errors
    // the code you think can raise an exception
}
catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// optional
finally {
    // block of code to be executed after try block ends
}
```

Points to remember:

- In a method, there can be more than one statement that might throw an exception, So put all these statements within their own **try** block and provide a separate exception handler within their own **catch** block for each of them.
- If an exception occurs within the try block, that exception is handled by the exception

- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put a **catch** block after it. There can be more than one exception handlers. Each **catch** block is a exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of exception that it can handle and must be the name of the class that inherits from the **Throwable** class.
- For each try block there can be zero or more catch blocks, but **only one** final block.
- The finally block is optional.It always gets executed whether an exception occurred in try block or not. If an exception occurs, then it will be executed after **try and catch blocks.** And if an exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

Summary



Related Articles:

- Types of Exceptions in Java
- Checked vs Unchecked Exceptions
- Throw-Throws in Java

This article is contributed by **Nitsdheerendra** and **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.