Exception Handling

By
Anand Kulkarni
anand.pune38@gmail.com

Contents

What is an exception?

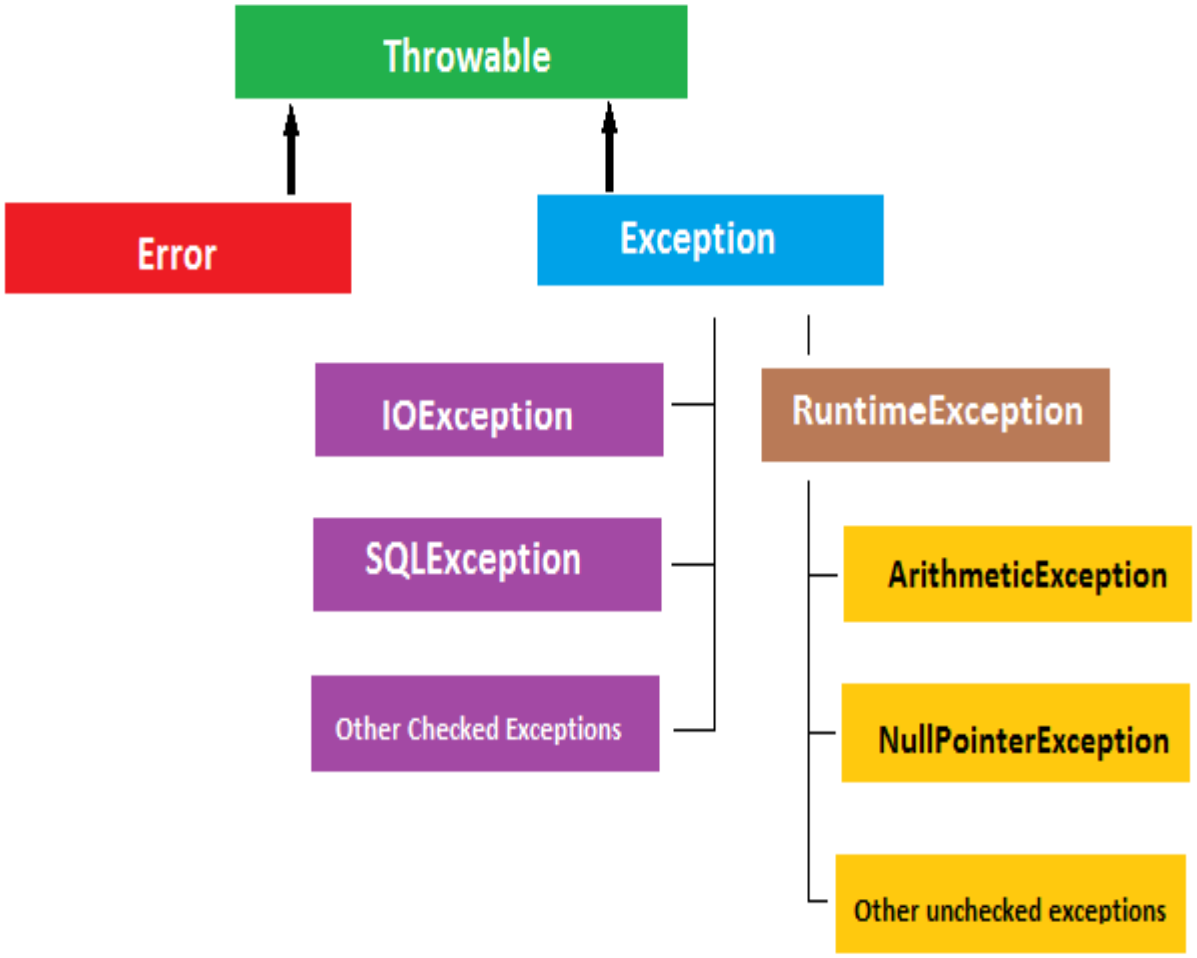
- Exception is an abnormal condition occurred in your program.
- Exception Handling is a mechanism to handle runtime errors.
- Java handles exception in object oriented fashion. i.e. if an exception occurred during program execution then JVM throws corresponding exception class object on the user.

Exception handling code

```
try {
        System.out.println("Getting division");
        int division = x / y;
        System.out.println("Division = " + division);
}
catch(ArithmeticException e) {
        e.printStackTrace();
}
```

- Any java code that might throw exception, should be kept inside 'try' block.
- If exception is raised then JVM generates corresponding exception object & thrown on user. If you have suitable catch block then it will run the catch block & continue with remaining code. However, if suitable catch is not found then JVM will terminate the thread.

Exception Hierarchy



Exception vs Error

Exception	Error
Exception is an unwanted condition an application can handle	Error is serious problem which cannot be handled in an application.
Exception can be handled using try/catch block	Error cannot be handled using try/catch block.
You can recover an exception using try/catch.	Error cannot be recovered. Even though you use try/catch block, it will terminate the application.
Exception can be checked or unchecked.	Error can be only unchecked.

Checked vs unchecked exception

Checked Exception	Unchecked Exception
Checked exceptions are verified at compilation time & compiler gives error if checked exception is not handled in a program.	Unchecked exceptions are not verified by compiler & hence you may or may not handle unchecked exception in a program.
Checked exception is not sub class of RuntimeException	All unchecked exceptions extend RuntimeException class.
The ‘throws’ keyword is applicable for checked exception.	The ‘throws’ keyword is not used for unchecked exception.

Different forms of try/catch block

There are mainly three forms of try/catch block:

- Try with single catch
- Try with multiple catch
- Nested try/catch blocks

Try with single catch

```
try {
        System.out.println(“Getting division”);
        int division = x / y;
        System.out.println(“Division = ” + division);
}
catch(ArithmeticException e) {
        e.printStackTrace();
}
```

Try with multiple catch

```
try {
        System.out.println("Getting division");
        int division = x / y;
        System.out.println("Division = " + division);
}
catch(ArithmeticException e) {
        e.printStackTrace();
}
catch(Exception e) {
        e.printStackTrace();
}
```

In try with multiple catch, specific exception catch should be declared before generic exception catch.

Nested try/catch blocks

```
try {
        int division = x / y;
        try {
                int ary[] = new int[5];
                ary[7] = 34;
        }
        catch(ArrayIndexOutOfBoundsException e) {
                e.printStackTrace();
        }
}
catch(ArithmeticException e) {
        e.printStackTrace();
}
```

Nested try/catch blocks continue…

- In nested try/catch block, you will find try inside try block.
- If outer try block throws exception then control goes to outer catch block.
- If inner try block throws exception then JVM first tries to find corresponding inner catch. If no suitable inner catch is found then JVM checks for suitable outer catch block.

‘throws’ keyword

```
class FileOperations {
        public String readFile(String fileName) throws FileNotFoundException
}
```

- In the above program, readFile() function write a logic to read contents of a file. However, if somebody calling readFile() function, he might pass incorrect file name. In such cases, readFile() function should not handle FileNotFoundException because function caller may want to handle it in different way. Hence, readFile() declares that it might throw FileNotFoundException if file name is invalid. Such declaration can me made using ‘throws’ clause.

‘throws’ keyword

- The ‘throws’ keyword should be applied only on checked exception. If you try to apply throws for unchecked exception then compiler will simply ignore it.
- Thus, if you are calling any method that throws an exception then function caller must handle it using try/catch block.

Exception handling in method overriding

```
class Super {
        public void m1() throws FileNotFoundException { }
        public void m2() throws IOException { }
}
class Sub extends Super {
        public void m1() throws IOException { } //compilation error
        public void m2() throws FileNotFoundException { } //correct
}
```

During method override, overridden method can keep exceptions same as exceptions mentioned in throws clause of super class method or throw specific exception.

‘finally’ keyword

- ✓ Sometimes you wish to execute a code irrespective whether or not try block throws exception. In such case, you can use ‘finally’ block.
- ✓ The ‘finally’ blocks can have two forms:
    1. Try, catch, finally
    2. Try with only finally

```
try {
        int z = 5/0;
}
catch(ArithmeticException e) {
        e.printStackTrace();
}
finally {
        println("Done");
}
```

```
try {
        int z = 5/0;
}
finally {
        println("Done");
}
```

‘finally’ keyword continue…

- **Try, catch, finally**

  In this form, if try throws an exception then it is caught by catch & at the end JVM runs finally block.

- **Try with only finally**

  In this form, if try throws an exception then control directly goes to finally block & after that it terminates your application without handling the exception.

Try with resources

- There are several try block definitions where we create external resource object like file, database connection etc., call their business logic methods & at the end invoke close() function in finally block.
- The close() operation can be automatically handled by java if we use ‘try with resources’ feature.
- It is mandatory that your resource must implement AutoCloseable interface.

```
try(FileInputStream input = new FileInputStream("file.txt")) {

    int data = input.read();
    while(data != -1){
       System.out.print((char) data);
       data = input.read();
    }
}
```

Try with multiple resources

```
try(FileInputStream input = new FileInputStream("file.txt"); BufferedInputStream
bufferedInput = new BufferedInputStream(input)) {
        int data = input.read();
        while(data != -1){
            System.out.print((char) data);
            data = input.read();
        }
}
```

AutoCloseable interface

The try-with-resources construct does not just work with Java's built-in classes. You can also implement the java.lang.AutoCloseable interface in your own classes, and use them with the try-with-resources construct.

```
public interface AutoClosable {
    public void close() throws Exception;
}
public class MyAutoClosable implements AutoCloseable {
    @Override
    public void close() throws Exception {
        System.out.println("MyAutoClosable closed!");
    }
}
```

Custom exceptions

- There are several places where our application error cannot be handled by built-in exception class & hence we write our own exception class. It is called as custom exception.
- The custom exception class can extend either Exception or RuntimeException i.e. custom exception can be either checked exception or unchecked exception.

Custom exception class

```
public class DayException extends Exception { //checked exception
        private String message;
        public DayException() { this.message = “”; }
        public DayException(String message) { this.message = message; }
        public String toString() {
                return “Day should be in between 1 to 31. “ + this.message;
        }
}
```

Using custom exception

```
class Date {
        public void setDay(int day) throws DayException {
                if(day > 31 || day < 1)
                        throw new DayException("Invalid day: " + day);
                this.day = day;
        }
}
class Main {
        public static void main(String args[]) {
                Date date = new Date(10, 3, 2010);
                try {
                        date.setDay(35);
                }
                catch(DayException e) { e.printStackTrace(); }
        }
}
```