Unit-8-OOP NOTES

Exception Handling in Java

Exception is an abnormal condition that arises in a piece of code.

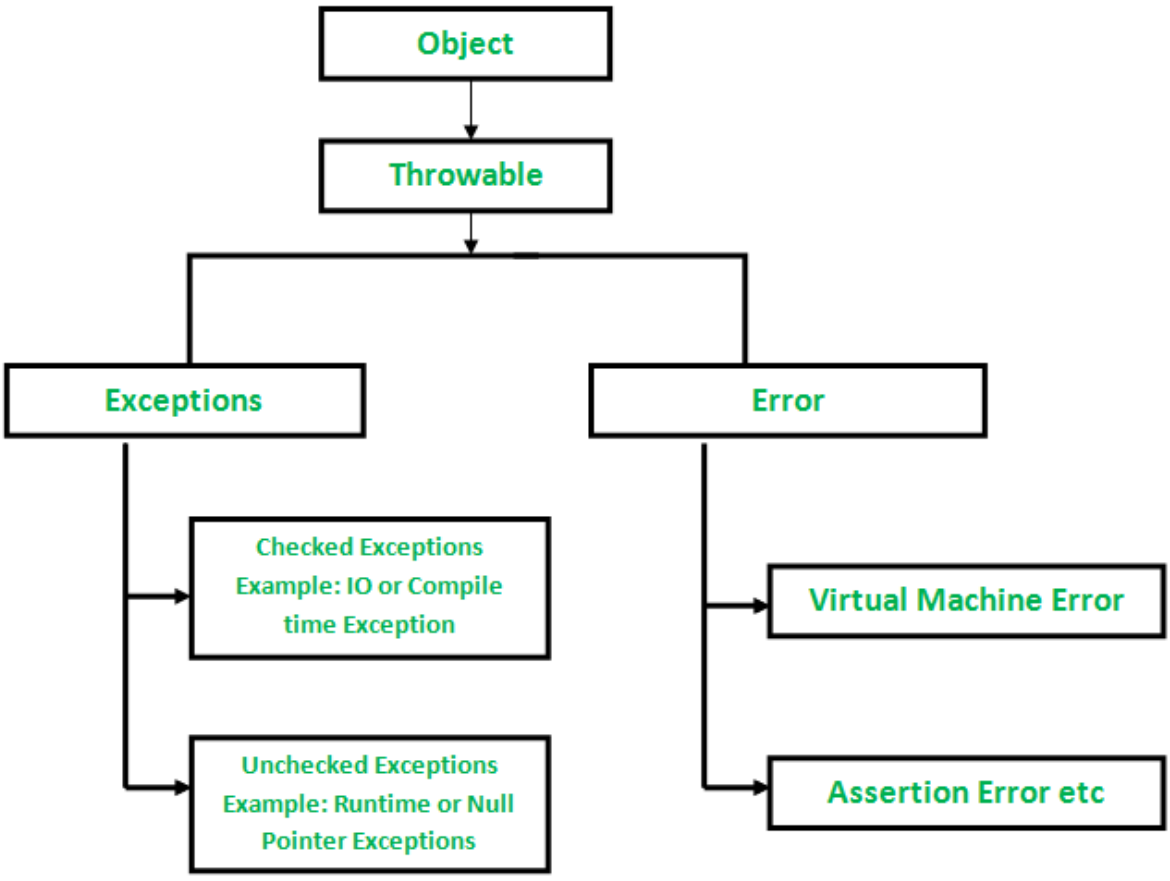
Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program.

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.

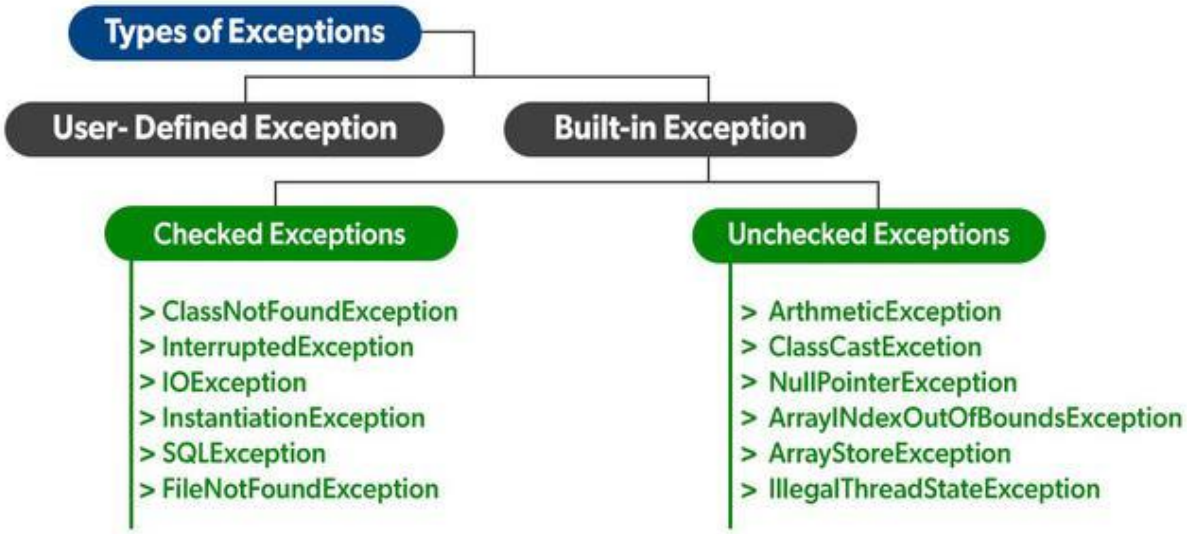
Major reasons why an exception Occurs

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

Exception Hierarchy



Types of Exceptions



Built-in Exception

1. Checked Exceptions: Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

2. 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.

```
import java.io.*;
class A {
   public static void main (String[] args) {
     int a=5;
     int b=0;
      try{
        System.out.println(a/b);
      }
     catch(ArithmeticException e){
      e.printStackTrace();
     }
```

```
        }
    }
```

Output:

java.lang.ArithmeticException: / by zero

at GFG.main(File.java:10)

Java try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

Syntax

```
try {
  // Block of code to try
}
catch(Exception e) {
  // Block of code to handle errors
}
```

This will generate an error, because **myNumbers[10]** does not exist.

```
public class Main {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
```

```
    System.out.println(myNumbers[10]); // error!
  }
}
```

The output will be something like this:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at Main.main(Main.java:4)
```

```
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

The output will be:

```
Something went wrong.
```

Finally

The finally statement lets you execute code, after try...catch, regardless of the result:

```
public class Main {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
}
```

The throw keyword

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception type**. There are many exception types available in
Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc:

Example

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
```

```
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    }
    else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)
  }
}
```

Java throws keyword

We use the throws keyword in the method declaration to declare the type of exceptions that might occur within it.

Its syntax is:

```
accessModifier returnType methodName() throws ExceptionType1, ExceptionType2 … {
  // code
}
```

Difference between throw and throws

Throw	throws
Used to throw an exception for a method	Used to indicate what exception type may be thrown by a method
Cannot throw multiple exceptions	Can declare multiple exceptions
Syntax: • throw is followed by an object (new *type*) • used inside the method	**Syntax:** • throws is followed by a class • and used with the method signature

Nested try block

using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

```
class NestedTry {

    // main method
    public static void main(String args[])
    {
        // Main try block
        try {

            // initializing array
            int a[] = { 1, 2, 3, 4, 5 };

            // trying to print element at index 5
            System.out.println(a[5]);

            // try-block2 inside another try block
            try {

                // performing division by zero
                int x = a[2] / 0;
            }
            catch (ArithmeticException e2) {
                System.out.println("division by zero is not possible");
            }
        }
        catch (ArrayIndexOutOfBoundsException e1) {
            System.out.println("ArrayIndexOutOfBoundsException");
            System.out.println("Element at such index does not exists");
        }
    }
    // end of main method
}
```