Day 26
Exception Handling in Java

The Exception Handling in Java is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

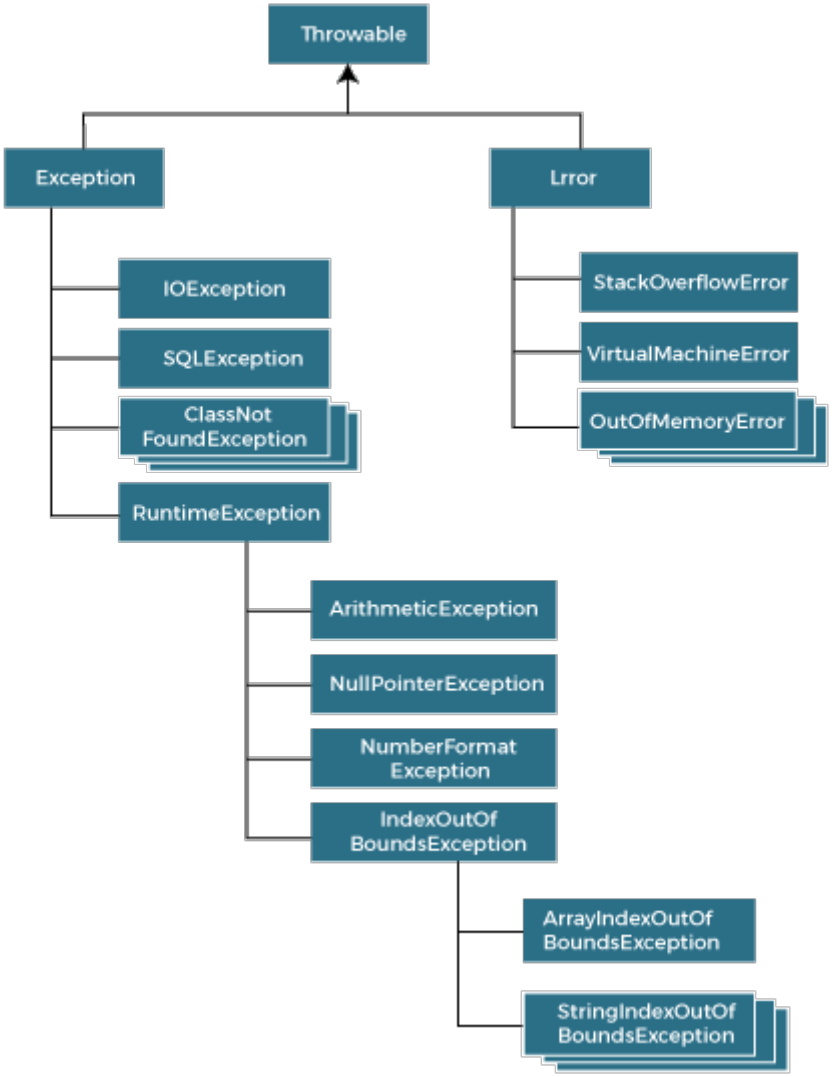
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling

Hierarchy of Java Exception classes

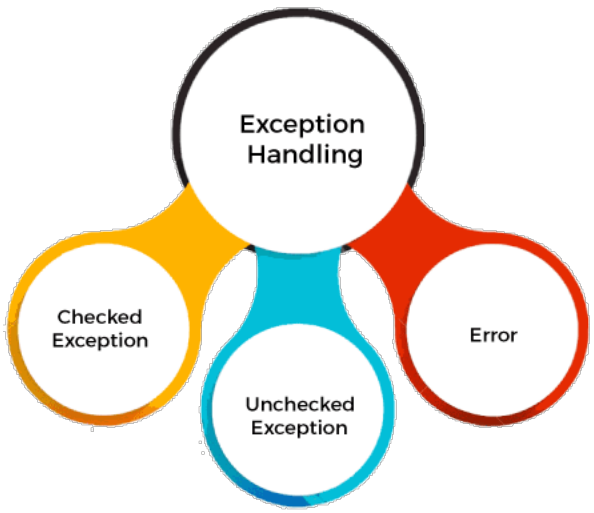
The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error



Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Java try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
   try{
   //code that may throw an exception
   }catch(Exception_class_Name ref){}

Syntax of try-finally block

   try{
   //code that may throw an exception
   }finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block

Java try-catch block

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.
-

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.
Example 1

TryCatchExample1.java

```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

    }

}
```

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

As displayed in the above example, the rest of the code is not executed (in such case, the rest of the code statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.
Example 2

TryCatchExample2.java

```
public class TryCatchExample2 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            //handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

java.lang.ArithmeticException: / by zero
rest of the code

As displayed in the above example, the rest of the code is executed, i.e., the rest of the code statement is printed.

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

TryCatchExample3.java

```
public class TryCatchExample3 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
                    // if exception occurs, the remaining statement will not exceute
        System.out.println("rest of the code");
        }
            // handling the exception
        catch(ArithmeticException e)
        {
```

```
            System.out.println(e);
        }

    }

}
```

Output:

java.lang.ArithmeticException: / by zero

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Example 4

Here, we handle the exception using the parent class exception.

TryCatchExample4.java

```
public class TryCatchExample4 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            // handling the exception by using Exception class
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

java.lang.ArithmeticException: / by zero
rest of the code

Example 5

Let's see an example to print a custom message on exception.

TryCatchExample5.java

```
public class TryCatchExample5 {

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

```
        try
        {
        int data=50/0; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
                // displaying the custom message
           System.out.println("Can't divided by zero");
        }
    }

}
```

Output:

Can't divided by zero

Example 6

Let's see an example to resolve the exception in a catch block.

TryCatchExample6.java

```
public class TryCatchExample6 {

    public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try
        {
        data=i/j; //may throw exception
        }
           // handling the exception
        catch(Exception e)
        {
            // resolving the exception in catch block
           System.out.println(i/(j+2));
        }
    }
}
```

Output:

25

Example 7

In this example, along with try block, we also enclose exception code in a catch block.

TryCatchExample7.java

```
public class TryCatchExample7 {

    public static void main(String[] args) {

        try
        {
        int data1=50/0; //may throw exception

        }
             // handling the exception
        catch(Exception e)
        {
            // generating the exception in catch block
        int data2=50/0; //may throw exception

        }
    System.out.println("rest of the code");
    }
}
```

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

Example 8

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

TryCatchExample8.java

```
public class TryCatchExample8 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception

        }
            // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
```

```
}
```

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

Example 9

Let's see an example to handle another unchecked exception.

TryCatchExample9.java

```
public class TryCatchExample9 {

    public static void main(String[] args) {
        try
        {
        int arr[]= {1,3,5,7};
        System.out.println(arr[10]); //may throw exception
        }
            // handling the array exception
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code

Example 10

Let's see an example to handle checked exception.

TryCatchExample10.java

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class TryCatchExample10 {

    public static void main(String[] args) {


        PrintWriter pw;
```

```
        try {
            pw = new PrintWriter("jtp.txt"); //may throw exception
            pw.println("saved");
        }
   // providing the checked exception handler
    catch (FileNotFoundException e) {

            System.out.println(e);
        }
      System.out.println("File saved successfully");
      }
   }
```

Codes

Q. Write a Java program to use the try and catch and finally block.

Answer:

In this example, we are implementing try and catch block to handle the exception. The error code written is in try block and catch block handles the raised exception. The finally block will be executed on every condition.

```
class ExceptionTest
{
	public static void main(String[] args)
	{
		int a = 40, b = 4, c = 4;
		int result;
		try
		{
			result = a / (b-c);
		}
		catch (ArithmeticException ae)
		{
			System.out.println("Cannot divided by zero."+ae);
		}
		finally
		{
			System.out.println("finally block");
		}
		result = a / (b+c);
		System.out.println("Result: "+result);
	}
}
```