

Exception Handling in Java

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.

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.

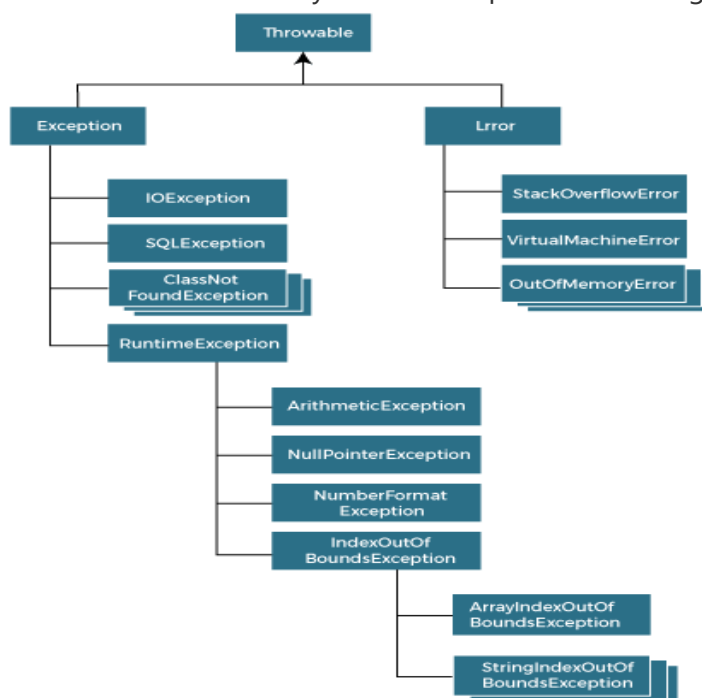
Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5; *//exception occurs*
6. statement 6;
7. statement 7;

Suppose there are 7 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 and 7 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 in [Java](#).

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.

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

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

9. `}}`

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where `ArithmeticException` occurs

If we divide any number by zero, there occurs an `ArithmeticException`.

1. `int a=50/0;//ArithmeticException`

2) A scenario where `NullPointerException` occurs

If we have a null value in any variable, performing any operation on the variable throws a `NullPointerException`.

1. `String s=null;`
2. `System.out.println(s.length());//NullPointerException`

3) A scenario where `NumberFormatException` occurs

If the formatting of any variable or number is mismatched, it may result into `NumberFormatException`. Suppose we have a string variable that has characters; converting this variable into digit will cause `NumberFormatException`.

1. `String s="abc";`
2. `int i=Integer.parseInt(s);//NumberFormatException`

4) A scenario where `ArrayIndexOutOfBoundsException` occurs

When an array exceeds to its size, the `ArrayIndexOutOfBoundsException` occurs. There may be other reasons to occur `ArrayIndexOutOfBoundsException`. Consider the following statements.

1. `int a[]=new int[5];`
2. `a[10]=50; //ArrayIndexOutOfBoundsException`

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

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

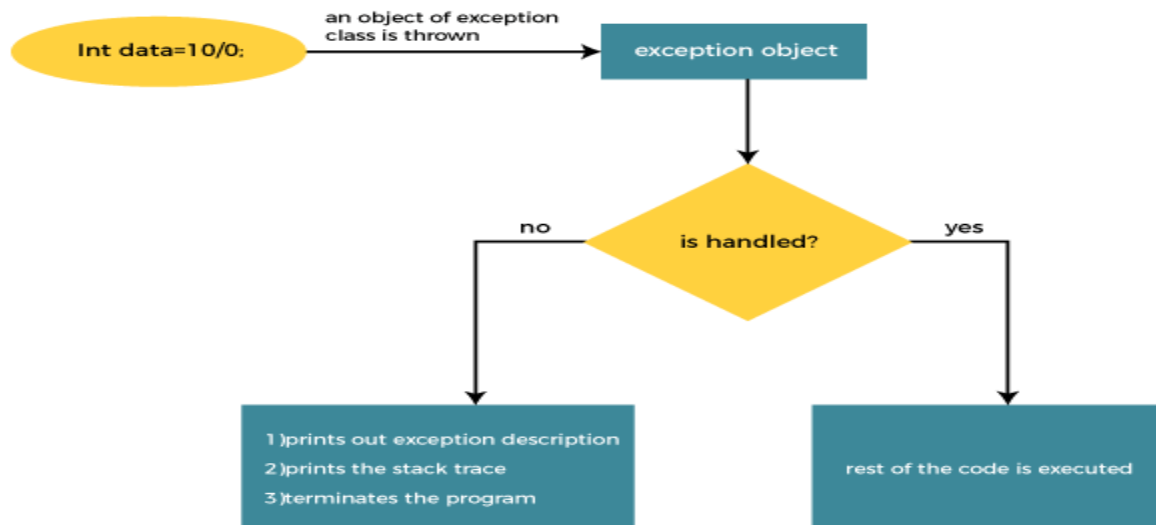
Syntax of try-finally block

1. `try{`
2. `//code that may throw an exception`
3. `}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.

Working of 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

```
1. public class TryCatchExample1 {
2.
3.     public static void main(String[] args) {
4.
5.         int data=50/0; //may throw exception
6.
7.         System.out.println("rest of the code");
8.
9.     }
10. }
```

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

```
1. public class TryCatchExample2 {
2.
3.     public static void main(String[] args) {
4.         try
5.         {
6.             int data=50/0; //may throw exception
7.         }
8.         //handling the exception
9.         catch(ArithmeticException e)
10.        {
11.            System.out.println(e);
12.        }
13.        System.out.println("rest of the code");
14.    }
15.
16. }
```

Output:

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

Example 3

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

```
1. public class TryCatchExample3 {
2.     public static void main(String[] args) {
3.         try
4.         {
5.             int data=50/0; //may throw exception
6.             // if exception occurs, the remaining statement will not execute
7.             System.out.println("rest of the code");
8.         }
9.         // handling the exception
10.        catch(ArithmeticException e)
11.        {
12.            System.out.println(e);
13.        }
14.
15.    }
16.
17. }
```

Output:

```
java.lang.ArithmeticException: / by zero
```

Example 4

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

```
1. public class TryCatchExample4 {
2.
3.     public static void main(String[] args) {
4.         try
5.         {
6.             int data=50/0; //may throw exception
7.         }
8.         // handling the exception by using Exception class
9.         catch(Exception e)
10.        {
11.            System.out.println(e);
12.        }
13.        System.out.println("rest of the code");
14.    }
15. }
```

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

```
1. public class TryCatchExample5 {
2.
3.     public static void main(String[] args) {
4.         try
5.         {
6.             int data=50/0; //may throw exception
7.         }
8.         // handling the exception
9.         catch(Exception e)
10.        {
11.            // displaying the custom message
12.            System.out.println("Can't divided by zero");
13.        }
14.    }
15.
16. }
```

Output:

```
Can't divided by zero
```

Example 6

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

```
1. public class TryCatchExample6 {
2.
3.     public static void main(String[] args) {
4.         int i=50;
5.         int j=0;
6.         int data;
7.         try
8.         {
9.             data=i/j; //may throw exception
10.        }
11.        // handling the exception
12.        catch(Exception e)
13.        {
14.            // resolving the exception in catch block
15.            System.out.println(i/(j+2));
16.        }
17.    }
18. }
```

Output:

25

Example 7

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

```
1. public class TryCatchExample7 {
2.
3.     public static void main(String[] args) {
4.
5.         try
6.         {
7.             int data1=50/0; //may throw exception
8.
9.         }
10.        // handling the exception
11.        catch(Exception e)
12.        {
13.            // generating the exception in catch block
14.            int data2=50/0; //may throw exception
15.
16.        }
17.        System.out.println("rest of the code");
18.    }
19. }
```

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

```
1. public class TryCatchExample8 {
2.
3.     public static void main(String[] args) {
4.         try
5.         {
6.             int data=50/0; //may throw exception
7.
8.         }
9.         // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
10.        catch(ArrayIndexOutOfBoundsException e)
11.        {
12.            System.out.println(e);
13.        }
14.        System.out.println("rest of the code");
15.    }
16.
17. }
```

Output:

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

Example 9

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

```
1. public class TryCatchExample9 {
2.
3.     public static void main(String[] args) {
4.         try
5.         {
6.             int arr[]= {1,3,5,7};
7.             System.out.println(arr[10]); //may throw exception
8.         }
9.         // handling the array exception
10.        catch(ArrayIndexOutOfBoundsException e)
11.        {
12.            System.out.println(e);
13.        }
14.        System.out.println("rest of the code");
15.    }
16.
17. }
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```


Example 10

Let's see an example to handle checked exception.

```
1. import java.io.FileNotFoundException;
2. import java.io.PrintWriter;
3. public class TryCatchExample10 {
4.
5.     public static void main(String[] args) {
6.         PrintWriter pw;
7.         try {
8.             pw = new PrintWriter("jtp.txt"); //may throw exception
9.             pw.println("saved");
10.        }
11.    // providing the checked exception handler
12.    catch (FileNotFoundException e) {
13.
14.        System.out.println(e);
15.    }
16.    System.out.println("File saved successfully");
17. }
18. }
```

Output:

```
File saved successfully
```

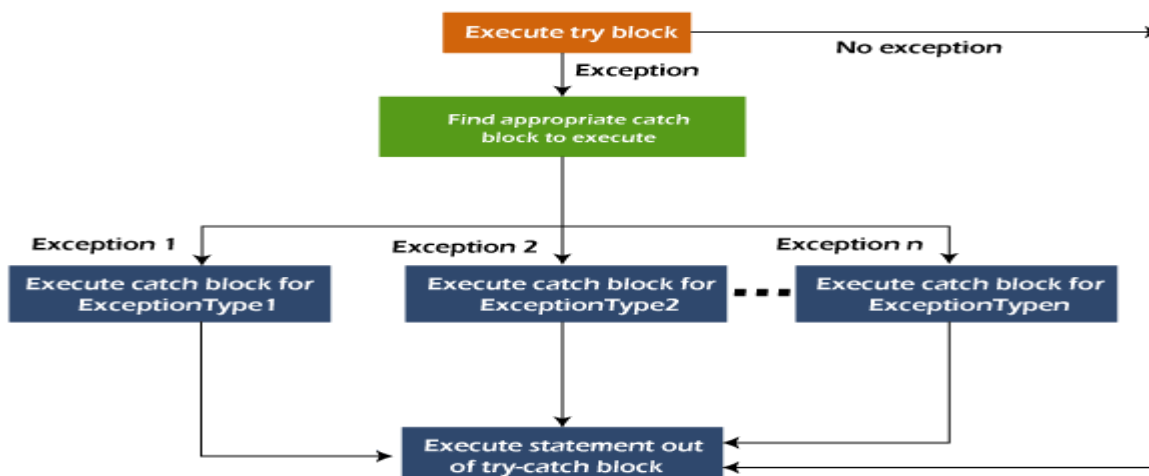
Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmeticException` must come before catch for `Exception`.

Flowchart of Multi-catch Block



MultipleCatchBlock1.java

```
1. public class MultipleCatchBlock1 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
7.             a[5]=30/0;
8.         }
9.         catch(ArithmeticException e)
10.            {
11.                System.out.println("Arithmetic Exception occurs");
12.            }
13.        catch(ArrayIndexOutOfBoundsException e)
14.            {
15.                System.out.println("ArrayIndexOutOfBoundsException occurs");
16.            }
17.        catch(Exception e)
18.            {
19.                System.out.println("Parent Exception occurs");
20.            }
21.        System.out.println("rest of the code");
22.    }
23. }
```

Output:

```
Arithmetic Exception occurs
rest of the code
Arithmetic Exception occurs
rest of the code
```

MultipleCatchBlock2.java

```
1. public class MultipleCatchBlock2 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
7.
8.             System.out.println(a[10]);
9.         }
10.        catch(ArithmeticException e)
11.            {
12.                System.out.println("Arithmetic Exception occurs");
13.            }
14.        catch(ArrayIndexOutOfBoundsException e)
15.            {
16.                System.out.println("ArrayIndexOutOfBoundsException occurs");
17.            }
18.        catch(Exception e)
```

```

19.     {
20.         System.out.println("Parent Exception occurs");
21.     }
22.     System.out.println("rest of the code");
23. }
24. }

```

Output:

```

ArrayIndexOutOfBoundsException occurs
rest of the code

```

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

MultipleCatchBlock3.java

```

1. public class MultipleCatchBlock3 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
7.             a[5]=30/0;
8.             System.out.println(a[10]);
9.         }
10.        catch(ArithmeticException e)
11.        {
12.            System.out.println("Arithmetic Exception occurs");
13.        }
14.        catch(ArrayIndexOutOfBoundsException e)
15.        {
16.            System.out.println("ArrayIndexOutOfBoundsException occurs");
17.        }
18.        catch(Exception e)
19.        {
20.            System.out.println("Parent Exception occurs");
21.        }
22.        System.out.println("rest of the code");
23.    }
24. }

```

Output:

```

Arithmetic Exception occurs
rest of the code

```

Example 4

In this example, we generate `NullPointerException`, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked.

MultipleCatchBlock4.java

```
1. public class MultipleCatchBlock4 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             String s=null;
7.             System.out.println(s.length());
8.         }
9.         catch(ArithmeticException e)
10.            {
11.                System.out.println("Arithmetic Exception occurs");
12.            }
13.        catch(ArrayIndexOutOfBoundsException e)
14.            {
15.                System.out.println("ArrayIndexOutOfBoundsException occurs");
16.            }
17.        catch(Exception e)
18.            {
19.                System.out.println("Parent Exception occurs");
20.            }
21.        System.out.println("rest of the code");
22.    }
23. }
```

Output:

```
Parent Exception occurs
rest of the code
```

Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
1. class MultipleCatchBlock5{
2.     public static void main(String args[]){
3.         try{
4.             int a[]=new int[5];
5.             a[5]=30/0;
6.         }
7.         catch(Exception e){System.out.println("common task completed");}
8.         catch(ArithmeticException e){System.out.println("task1 is completed");}
9.         catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
10.        System.out.println("rest of the code...");
11.    }
12. }
```

Output:

```
Compile-time error
```

Java Nested try block

In Java, 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 **ArithmeticException** (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.

Syntax:

```
1. ....
2. //main try block
3. try
4. {
5.     statement 1;
6.     statement 2;
7. //try catch block within another try block
8.     try
9.     {
10.         statement 3;
11.         statement 4;
12. //try catch block within nested try block
13.         try
14.         {
15.             statement 5;
16.             statement 6;
17.         }
18.         catch(Exception e2)
19.         {
20. //exception message
21.         }
22.
23.     }
24.     catch(Exception e1)
25.     {
26. //exception message
27.     }
28. }
29. //catch block of parent (outer) try block
30. catch(Exception e3)
31. {
32. //exception message
33. }
34. ....
```

Java Nested try Example

Let's see an example where we place a try block within another try block for two different exceptions.

NestedTryBlock.java

```
1. public class NestedTryBlock{
2.     public static void main(String args[]){
3.         //outer try block
4.         try{
5.             //inner try block 1
6.             try{
7.                 System.out.println("going to divide by 0");
8.                 int b = 39/0;
9.             }
10.            //catch block of inner try block 1
11.            catch(ArithmeticException e)
12.            {
13.                System.out.println(e);
14.            }
15.            //inner try block 2
16.            try{
17.                int a[] = new int[5];
18.
19.                //assigning the value out of array bounds
20.                a[5] = 4;
21.            }
22.            //catch block of inner try block 2
23.            catch(ArrayIndexOutOfBoundsException e)
24.            {
25.                System.out.println(e);
26.            }
27.            System.out.println("other statement");
28.        }
29.        //catch block of outer try block
30.        catch(Exception e)
31.        {
32.            System.out.println("handled the exception (outer catch)");
33.        }
34.
35.        System.out.println("normal flow..");
36.    }
37.}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock.java
C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..
```

When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

```
1. public class NestedTryBlock2 {
2.
3.     public static void main(String args[])
4.     {
5.         // outer (main) try block
6.         try {
7.
8.             //inner try block 1
9.             try {
10.
11.                 // inner try block 2
12.                 try {
13.                     int arr[] = { 1, 2, 3, 4 };
14.
15.                     //printing the array element out of its bounds
16.                     System.out.println(arr[10]);
17.                 }
18.
19.                 // to handles ArithmeticException
20.                 catch (ArithmeticException e) {
21.                     System.out.println("Arithmetic exception");
22.                     System.out.println(" inner try block 2");
23.                 }
24.             }
25.
26.             // to handle ArithmeticException
27.             catch (ArithmeticException e) {
28.                 System.out.println("Arithmetic exception");
29.                 System.out.println("inner try block 1");
30.             }
31.         }
32.
33.         // to handle ArrayIndexOutOfBoundsException
34.         catch (ArrayIndexOutOfBoundsException e4) {
35.             System.out.print(e4);
36.             System.out.println(" outer (main) try block");
37.         }
38.         catch (Exception e5) {
```

```

39.     System.out.print("Exception");
40.     System.out.println(" handled in main try-block");
41. }
42. }
43. }

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock2.java

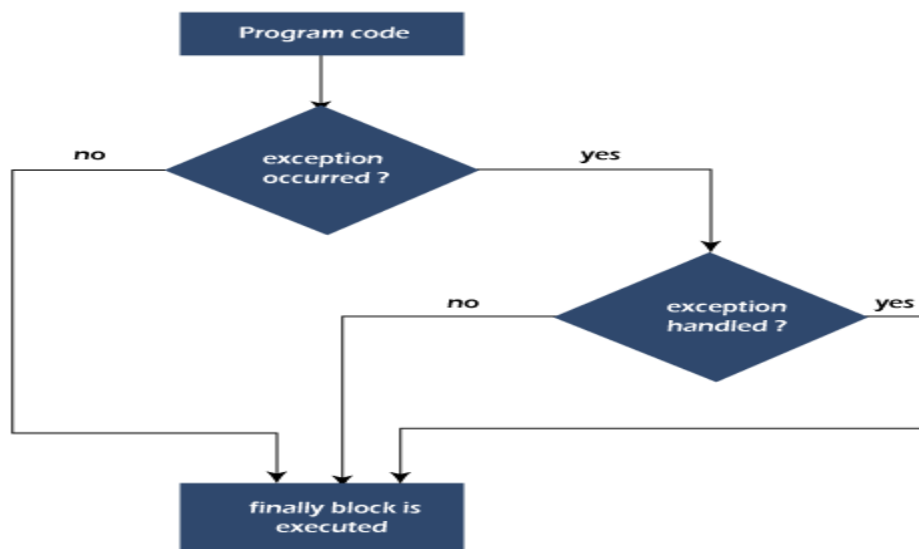
C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock2
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 4 outer
(main) try block

```

Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc. Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not. The finally block follows the try-catch block.

Flowchart of finally block



Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Usage of Java finally

Let's see the different cases where Java finally block can be used.

Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

```
1. class TestFinallyBlock {
2.     public static void main(String args[]){
3.         try{
4.             //below code do not throw any exception
5.             int data=25/5;
6.             System.out.println(data);
7.         }
8.         //catch won't be executed
9.         catch(NullPointerException e){
10.            System.out.println(e);
11.        }
12.        //executed regardless of exception occurred or not
13.        finally {
14.            System.out.println("finally block is always executed");
15.        }
16.
17.        System.out.println("rest of the code...");
18.    }
19. }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

Case 2: When an exception occurs but not handled by the catch block

Let's see the following example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

```
1. public class TestFinallyBlock1{
2.     public static void main(String args[]){
3.
4.         try {
5.
6.             System.out.println("Inside the try block");
7.
8.             //below code throws divide by zero exception
9.             int data=25/0;
10.            System.out.println(data);
11.        }
12.        //cannot handle Arithmetic type exception
```

```

13. //can only accept Null Pointer type exception
14. catch(NullPointerException e){
15.     System.out.println(e);
16. }
17.
18. //executes regardless of exception occurred or not
19. finally {
20.     System.out.println("finally block is always executed");
21. }
22.
23. System.out.println("rest of the code...");
24. }
25. }

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestFinallyBlock1.main(TestFinallyBlock1.java:9)

```

Case 3: When an exception occurs and is handled by the catch block

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

```

1. public class TestFinallyBlock2{
2.     public static void main(String args[]){
3.
4.     try {
5.
6.         System.out.println("Inside try block");
7.
8.         //below code throws divide by zero exception
9.         int data=25/0;
10.        System.out.println(data);
11.    }
12.
13.    //handles the Arithmetic Exception / Divide by zero exception
14.    catch(ArithmeticException e){
15.        System.out.println("Exception handled");
16.        System.out.println(e);
17.    }
18.
19.    //executes regardless of exception occurred or not
20.    finally {
21.        System.out.println("finally block is always executed");
22.    }
23.
24.    System.out.println("rest of the code...");
25. }

```

26. }

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock2
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if the program exits (either by calling `System.exit()` or by causing a fatal error that causes the process to abort).

Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

The Java throw keyword is used to throw an exception explicitly. We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw `ArithmeticException` if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

1. **throw new** exception_class("error message");

Let's see the example of throw `IOException`.

1. **throw new** `IOException("sorry device error");`

Where the Instance must be of type `Throwable` or subclass of `Throwable`. For example, `Exception` is the sub class of `Throwable` and the user-defined exceptions usually extend the `Exception` class.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

In this example, we have created a method named `validate()` that accepts an integer as a parameter. If the age is less than 18, we are throwing the `ArithmeticException` otherwise print a message welcome to vote.

1. **public class** TestThrow1 {
2. //function to check if person is eligible to vote or not
3. **public static void** validate(**int** age) {

```

4.     if(age<18) {
5.         //throw Arithmetic exception if not eligible to vote
6.         throw new ArithmeticException("Person is not eligible to vote");
7.     }
8.     else {
9.         System.out.println("Person is eligible to vote!!");
10.    }
11. }
12. //main method
13. public static void main(String args[]){
14.     //calling the function
15.     validate(13);
16.     System.out.println("rest of the code...");
17. }
18. }

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
vote
    at TestThrow1.validate(TestThrow1.java:8)
    at TestThrow1.main(TestThrow1.java:18)

```

The above code throw an unchecked exception.

Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

Example 2: Throwing Checked Exception

Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

```

1. import java.io.*;
2.
3. public class TestThrow2 {
4.
5.     //function to check if person is eligible to vote or not
6.     public static void method() throws FileNotFoundException {
7.
8.         FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
9.         BufferedReader fileInput = new BufferedReader(file);
10.
11.
12.         throw new FileNotFoundException();
13.
14.     }

```

```

15. //main method
16. public static void main(String args[]){
17.     try
18.     {
19.         method();
20.     }
21.     catch (FileNotFoundException e)
22.     {
23.         e.printStackTrace();
24.     }
25.     System.out.println("rest of the code...");
26. }
27. }

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestThrow2.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow2
java.io.FileNotFoundException
    at TestThrow2.method(TestThrow2.java:12)
    at TestThrow2.main(TestThrow2.java:22)
rest of the code...

```

Example 3: Throwing User-defined Exception

exception is everything else under the Throwable class.

TestThrow3.java

```

1. // class represents user-defined exception
2. class UserDefinedException extends Exception {
3.     public UserDefinedException(String str)
4.     {
5.         // Calling constructor of parent Exception
6.         super(str);
7.     }
8. }
9. // Class that uses above MyException
10. public class TestThrow3
11. {
12.     public static void main(String args[])
13.     {
14.         try
15.         {
16.             // throw an object of user defined exception
17.             throw new UserDefinedException("This is user-defined exception");
18.         }
19.         catch (UserDefinedException ude)
20.         {
21.             System.out.println("Caught the exception");
22.             // Print the message from MyException object
23.             System.out.println(ude.getMessage());
24.         }
25.     }
26. }

```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow3.java  
C:\Users\Anurati\Desktop\abcDemo>java TestThrow3  
Caught the exception  
This is user-defined exception
```

Java Exception Propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Note: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Exception Propagation Example

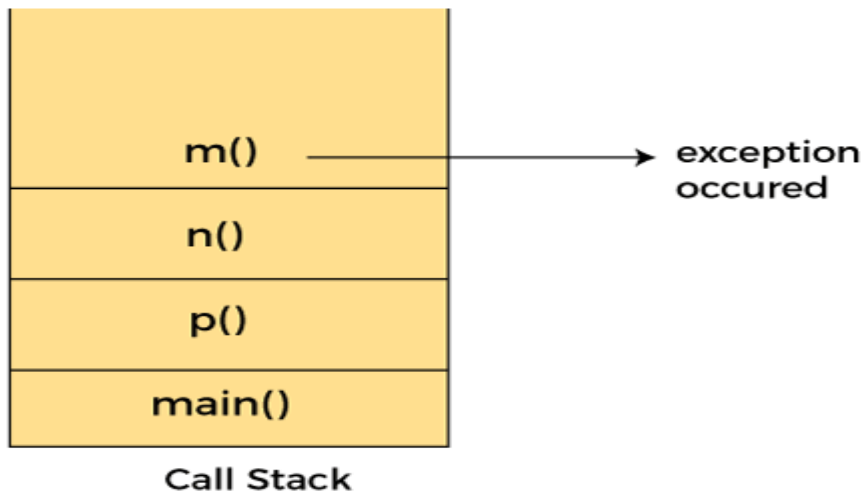
```
1. class TestExceptionPropagation1{  
2.     void m(){  
3.         int data=50/0;  
4.     }  
5.     void n(){  
6.         m();  
7.     }  
8.     void p(){  
9.         try{  
10.            n();  
11.        }catch(Exception e){System.out.println("exception handled");}  
12.    }  
13.    public static void main(String args[]){  
14.        TestExceptionPropagation1 obj=new TestExceptionPropagation1();  
15.        obj.p();  
16.        System.out.println("normal flow...");  
17.    }  
18. }
```

Output:

```
exception handled  
normal flow...
```

In the above example exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.

Exception can be handled in any method in call stack either in the main() method, p() method, n() method or m() method.



Note: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Exception Propagation Example

TestExceptionPropagation1.java

```
1. class TestExceptionPropagation2{
2.     void m(){
3.         throw new java.io.IOException("device error");//checked exception
4.     }
5.     void n(){
6.         m();
7.     }
8.     void p(){
9.         try{
10.            n();
11.        }catch(Exception e){System.out.println("exception handled");}
12.    }
13.    public static void main(String args[]){
14.        TestExceptionPropagation2 obj=new TestExceptionPropagation2();
15.        obj.p();
16.        System.out.println("normal flow");
17.    }
18. }
```

Output:

Compile Time Error

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as `NullPointerException`, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

1. return_type method_name() **throws** exception_class_name{
2. //method code
3. }

Which exception should be declared?

Ans: Checked exception only, because:

- **unchecked exception:** under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

1. return_type method_name() **throws** exception_class_name{
2. //method code
3. }

Which exception should be declared?

Ans: Checked exception only, because:

- **unchecked exception:** under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

Output:

```
exception handled
normal flow...
```

Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.

There are two cases:

1. **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
2. **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

Case 1: Handle Exception Using try-catch block

In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

Testthrows2.java

1. **import** java.io.*;
2. **class** M{


```

3. void method()throws IOException{
4.     throw new IOException("device error");
5. }
6. }
7. public class Testthrows2{
8.     public static void main(String args[]){
9.         try{
10.            M m=new M();
11.            m.method();
12.        }catch(Exception e){System.out.println("exception handled");}
13.
14.        System.out.println("normal flow...");
15.    }
16. }

```

Output:

```

exception handled
normal flow...

```

Case 2: Declare Exception

- In case we declare the exception, if exception does not occur, the code will be executed fine.
- In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.

Let's see examples for both the scenario.

A) If exception does not occur

Testthrows3.java

```

1. import java.io.*;
2. class M{
3.     void method()throws IOException{
4.         System.out.println("device operation performed");
5.     }
6. }
7. class Testthrows3{
8.     public static void main(String args[])throws IOException{//declare exception
9.         M m=new M();
10.        m.method();
11.
12.        System.out.println("normal flow...");
13.    }
14. }

```

Output:

```

device operation performed
normal flow...

```

B) If exception occurs

```

1. import java.io.*;
2. class M{
3.     void method()throws IOException{
4.         throw new IOException("device error");

```

```

5.  }
6.  }
7.  class Testthrows4{
8.      public static void main(String args[])throws IOException{//declare exception
9.          M m=new M();
10.         m.method();
11.
12.         System.out.println("normal flow...");
13.     }
14. }

```

Output:

```

Exception in thread "main" java.io.IOException: device error
at M.method(Testthrows4.java:4)
at Testthrows4.main(Testthrows4.java:10)

```

Difference between throw and throws in Java

The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method.

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example,

			main() throws IOException, SQLException.
--	--	--	--

Java throw Example

```
1. public class TestThrow {
2.     //defining a method
3.     public static void checkNum(int num) {
4.         if (num < 1) {
5.             throw new ArithmeticException("\nNumber is negative, cannot calculate square");
6.         }
7.         else {
8.             System.out.println("Square of " + num + " is " + (num*num));
9.         }
10.    }
11.    //main method
12.    public static void main(String[] args) {
13.        TestThrow obj = new TestThrow();
14.        obj.checkNum(-3);
15.        System.out.println("Rest of the code..");
16.    }
17. }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
    at TestThrow.checkNum(TestThrow.java:6)
    at TestThrow.main(TestThrow.java:16)
```

Java throws Example

```
1. public class TestThrows {
2.     public static int divideNum(int m, int n) throws ArithmeticException {
3.         int div = m / n;
4.         return div;    }
5.     public static void main(String[] args) {
6.         TestThrows obj = new TestThrows();
7.         try {
8.             System.out.println(obj.divideNum(45, 0));    }
9.         catch (ArithmeticException e){
10.             System.out.println("\nNumber cannot be divided by 0");    }
11.         System.out.println("Rest of the code..");
12.     } }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrows

Number cannot be divided by 0
Rest of the code..
```

Java throw and throws Example

```
1. public class TestThrowAndThrows
2. {
3.     // defining a user-defined method
4.     // which throws ArithmeticException
5.     static void method() throws ArithmeticException
6.     {
7.         System.out.println("Inside the method()");
8.         throw new ArithmeticException("throwing ArithmeticException");
9.     }
10.    //main method
11.    public static void main(String args[])
12.    {
13.        try
14.        {
15.            method();
16.        }
17.        catch(ArithmeticException e)
18.        {
19.            System.out.println("caught in main() method");
20.        }
21.    }
22. }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrowAndThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrowAndThrows
Inside the method()
caught in main() method
```

Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **final** is an access modifier, **finally** is the block in Exception Handling and **finalize** is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

Sr. no.	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.
3.	Functionality	(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.
4.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.	finalize method is executed just before the object is destroyed.

Java final Example

Let's consider the following example where we declare final variable age. Once declared it cannot be modified.

FinalExampleTest.java

1. public class FinalExampleTest {
2. //declaring final variable

```

3.    final int age = 18;
4.    void display() {
5.
6.        // reassigning value to age variable
7.        // gives compile time error
8.        age = 55;
9.    }
10.
11.    public static void main(String[] args) {
12.
13.        FinalExampleTest obj = new FinalExampleTest();
14.        // gives compile time error
15.        obj.display();
16.    }
17. }

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac FinalExampleTest.java
FinalExampleTest.java:10: error: cannot assign a value to final variable age
        age = 55;
        ^
1 error

```

In the above example, we have declared a variable final. Similarly, we can declare the methods and classes final using the final keyword.

Java finally Example

Let's see the below example where the Java code throws an exception and the catch block handles that exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

FinallyExample.java

```

1. public class FinallyExample {
2.     public static void main(String args[]){
3.         try {
4.             System.out.println("Inside try block");
5.             // below code throws divide by zero exception
6.             int data=25/0;
7.             System.out.println(data);
8.         }
9.         // handles the Arithmetic Exception / Divide by zero exception

```

```

10. catch (ArithmeticException e){
11.     System.out.println("Exception handled");
12.     System.out.println(e);
13. }
14. // executes regardless of exception occurred or not
15. finally {
16.     System.out.println("finally block is always executed");
17. }
18. System.out.println("rest of the code...");
19. }
20. }

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>java FinallyExample.java
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...

```

Java finalize Example

FinalizeExample.java

```

1. public class FinalizeExample {
2.     public static void main(String[] args)
3.     {
4.         FinalizeExample obj = new FinalizeExample();
5.         // printing the hashCode
6.         System.out.println("HashCode is: " + obj.hashCode());
7.         obj = null;
8.         // calling the garbage collector using gc()
9.         System.gc();
10.        System.out.println("End of the garbage collection");
11.    }
12.    // defining the finalize method
13.    protected void finalize()
14.    {
15.        System.out.println("Called the finalize() method");
16.    }
17. }

```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac FinalizeExample.java
Note: FinalizeExample.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

```
C:\Users\Anurati\Desktop\abcDemo>java FinalizeExample
HashCode is: 746292446
End of the garbage collection
Called the finalize() method
```