

Exception handling fundamentals

When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.

- Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within **try** block, it is thrown.
- Your code will **catch** this exception and handle in some rational manner.
- System generated exception automatically thrown to Java runtime system.
- To manually throw exception, use keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Code that absolutely must be executed before a method returns is put in **finally** block.

```
try {  
  
    // block of code to monitor for error  
  
}  
  
catch (ExceptionType1 exObj) {  
  
    //exception handle for ExceptionType1  
  
}  
  
catch (ExceptionType2 exObj) {  
  
    //exception handle for ExceptionType2  
  
}  
  
finally {  
  
    // block of code to be executed before try block ends  
  
}
```

Multiple catch Clauses

- We can specify two or more catch clauses
- When an exception is thrown, each catch statement is inspected in order and the first one whose type matches that of exception will be executed.

Example

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 Exception 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

```

Example 2

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 Exception 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
```

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

Example 1

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.  
16.  
17.            //inner try block 2  
18.            try{  
19.                int a[]=new int[5];  
20.  
21.                //assigning the value out of array bounds  
22.                a[8]=4;  
23.            }  
24.  
25.            //catch block of inner try block 2  
26.            catch(ArrayIndexOutOfBoundsException e)  
27.            {  
28.                System.out.println(e);  
29.            }  
30.  
31.  
32.            System.out.println("other statement");  
33.        }  
}
```

```

34. //catch block of outer try block
35. catch(Exception e)
36. {
37.     System.out.println("handled the exception (outer catch)");
38. }
39.
40. System.out.println("normal flow..");
41. }
42. }

```

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..

```

Note: 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.

Example 2

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.

NestedTryBlock.java

```

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:

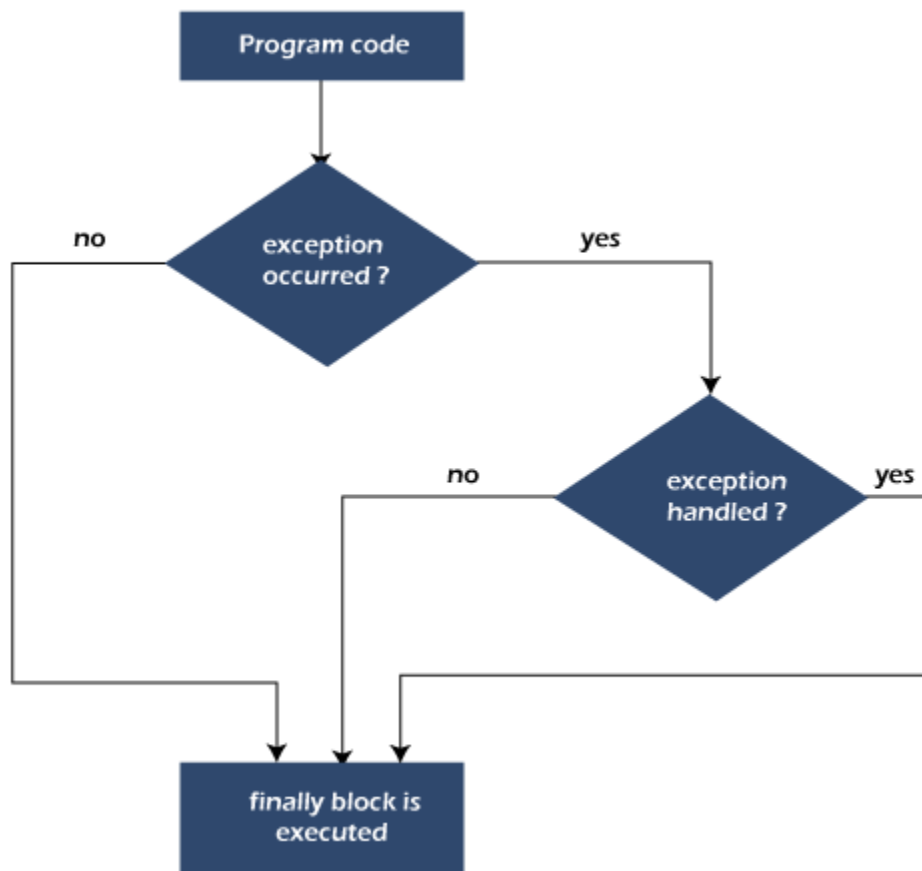
```

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.



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.

TestFinallyBlock.java

```
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...
```

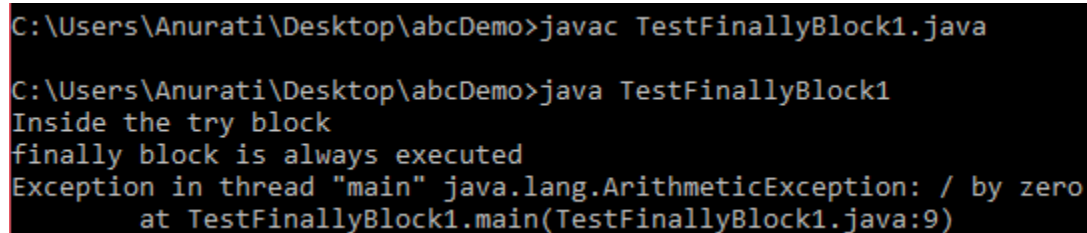
Case 2: When an exception occurs but not handled by the catch block

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.**

TestFinallyBlock1.java

```
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

Example:

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.**

TestFinallyBlock2.java

```
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

- You can only be catching exceptions that are thrown by the java runtime system
- It is also possible for your program to throw an exception explicitly, using the **throw** statement

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.

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.

TestThrow1.java

In this example, we have created the `validate` method that takes integer value 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.

TestThrow2.java

```

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. {
4.     public UserDefinedException(String str)
5.     {
6.         // Calling constructor of parent Exception
7.         super(str);
8.     }
9. }

```

```
10. // Class that uses above MyException
11. public class TestThrow3
12. {
13.     public static void main(String args[])
14.     {
15.         try
16.         {
17.             // throw an object of user defined exception
18.             throw new UserDefinedException("This is user-defined exception");
19.         }
20.         catch (UserDefinedException ude)
21.         {
22.             System.out.println("Caught the exception");
23.             // Print the message from MyException object
24.             System.out.println(ude.getMessage()); //value that is set by calling constructor
                //can get through getMessage()
25.         }
26.     }
27. }
```

Output:

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