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, its 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

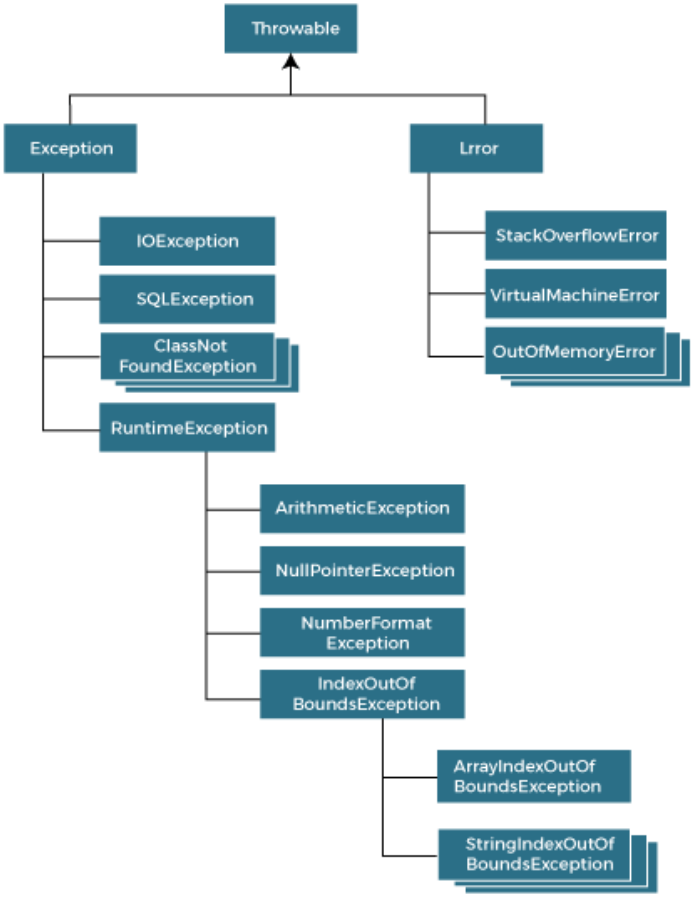
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. 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 in Java.

Hierarchy of Java Exception classes

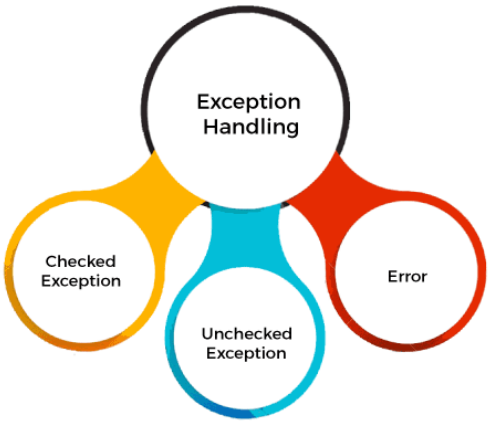
The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



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



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:

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.

JavaExceptionExample.java

```
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.   }
10. }
```

Output:

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

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions:

1) A scenario where **ArithmeticException** occurs

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

```
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**.

```
String s=null;
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**.

```
String s="abc";
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.

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

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.

Note: 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** block

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

Syntax of Java **try-finally** block

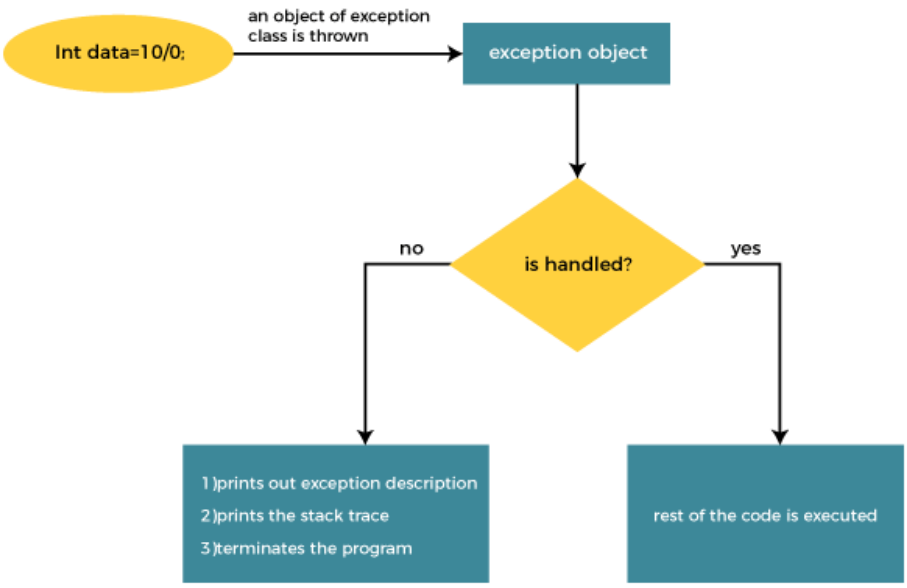
```
1. try{
2. //code that may throw an exception
3. }finally{}
```

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.

Internal 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

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

Example 1

TryCatchExample1.java

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

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

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

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

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

Output:

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

Note: 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

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

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

Output:

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

Example 6

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

TryCatchExample6.java

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

Output:

```
25
```

Example 7

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

TryCatchExample7.java

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

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

Output:

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

Example 9

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

TryCatchExample9.java

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

Output:

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

Example 10

Let's see an example to handle **checked** exception.

TryCatchExample10.java

```
1.  import java.io.FileNotFoundException;
2.  import java.io.PrintWriter;
3.
4.  public class TryCatchExample10 {
5.
6.      public static void main(String[] args) {
7.
8.          PrintWriter pw;
9.          try {
10.             pw = new PrintWriter("jtp.txt"); //may throw exception
11.             pw.println("saved");
12.         }
13. // providing the checked exception handler
14.  catch (FileNotFoundException e) {
15.
16.             System.out.println(e);
17.         }
18.     System.out.println("File saved successfully");
19.     }
20. }
```

Output:

```
File saved successfully
```

Java Catch Multiple Exceptions:

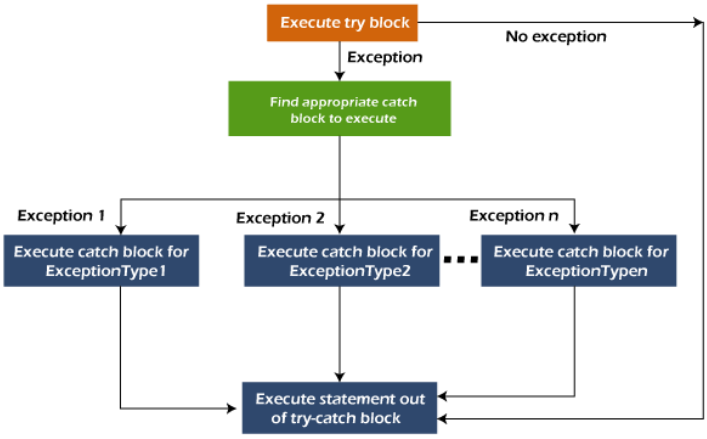
Java Multi-catch block

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



Example 1

Let's see a simple example of java multi-catch block.

MultipleCatchBlock1.java

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

Output:

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

Example 2

MultipleCatchBlock2.java

```
1.  public class MultipleCatchBlock2 {
2.     public static void main(String[] args) {
3.
4.          try{
5.              int a[]=new int[5];
6.              System.out.println(a[10]);
7.             }
8.             catch(ArithmeticException e)
9.               {
10.               System.out.println("Arithmetic Exception occurs");
11.              }
12.            catch(ArrayIndexOutOfBoundsException e)
13.              {
14.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
15.              }
16.            catch(Exception e)
17.              {
18.               System.out.println("Parent Exception occurs");
19.              }
20.            System.out.println("rest of the code");
21.    }
22. }
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

Example 3

Note: 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("ArrayIndexOutOfBounds Exception 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("ArrayIndexOutOfBounds 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
```

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).

MultipleCatchBlock5.java

```
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){
8.            System.out.println("common task completed");
9.    }
10.   catch(ArithmeticException e){
11.           System.out.println("task1 is completed");
12.    }
13.   catch(ArrayIndexOutOfBoundsException e){
14.           System.out.println("task 2 completed");
15.    }
16.   System.out.println("rest of the code...");
17.  }
18. }
```

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 **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

Syntax:

```
....
//main try block
try {
   statement 1;
   statement 2;
//try catch block within another try block
   Try {
      statement 3;
      statement 4;
//try catch block within nested try block
      try {
         statement 5;
         statement 6;
    }
      catch(Exception e2)
      {
//exception message
      }
   }
   catch(Exception e1)
   {
//exception message
   }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....
```

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.                  System.out.println(e);
13.             }
14.
15.         //inner try block 2
16.         try{
17.              int a[]=new int[5];
18.              //assigning the value out of array bounds
19.             a[5]=4;
20.              }
21.         //catch block of inner try block 2
22.         catch(ArrayIndexOutOfBoundsException e){
23.                 System.out.println(e);
24.          }
25.      System.out.println("other statement");
26.       }
27.  //catch block of outer try block
28.  catch(Exception e){
29.         System.out.println("handled the exception (outer catch)");
30.       }
31.  System.out.println("normal flow..");
32.  }
33. }
```

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.

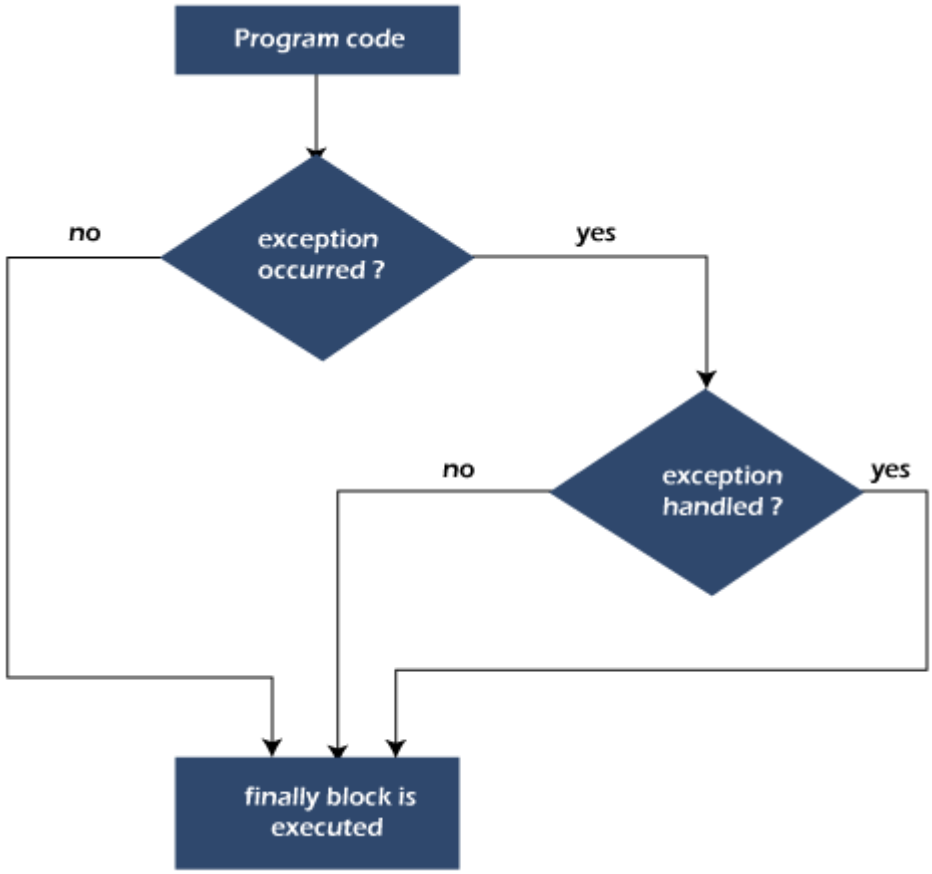
Java finally block

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.

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 **occur** 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 occured 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.        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 occured 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>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 will discuss custom exceptions later in this section.

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.,

```
throw new exception_class("error message");
```

Let's see the example of throw IOException.

```
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.

TestThrow1.java

```
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 checked 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.  public class TestThrow2 {
3.  //function to check if person is eligible to vote or not
4.  public static void method() throws FileNotFoundException {
5.  FileReader file = new FileReader("C:\\Users\\mk896\\Desktop\\abc.txt");
6.  BufferedReader fileInput = new BufferedReader(file);
7.  throw new FileNotFoundException();
8.  }
9.  //main method
10. public static void main(String args[]){
11. try
12. {
13. method();
14. }
15. catch (FileNotFoundException e)
16. {
17. e.printStackTrace();
18. }
19. System.out.println("rest of the code...");
20. }
21. }
```

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...
```

<u>Example 3:</u> Throwing User-defined Exception
Exception is everything else under the **Throwable** class.

TestThrow3.java

```
// class represents user-defined exception
class UserDefinedException extends Exception
{
   public UserDefinedException(String str)
   {
      // Calling constructor of parent Exception
      super(str);
   }
}
// Class that uses above MyException
public class TestThrow3
{
   public static void main(String args[])
   {
      try
      {
         // throw an object of user defined exception
         throw new UserDefinedException("This is user-defined exception");
      }
      catch (UserDefinedException ude)
      {
         System.out.println("Caught the exception");
         // Print the message from MyException object
         System.out.println(ude.getMessage());
      }
   }
}
```

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

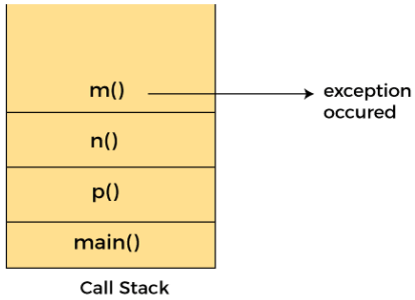
TestExceptionPropagation1.java

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

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).

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).

It provides information to the caller of the method about the exception.

Java **throws** Example

Let's see the example of Java throws clause which describes that checked exceptions can be propagated by throws keyword.

Testthrows1.java

```
1. import java.io.IOException;
2. class Testthrows1{
3.  void m()throws IOException{
4.   throw new IOException("device error");//checked exception
5.  }
6.  void n()throws IOException{
7.   m();
8.  }
9.  void p(){
10.  try{
11.   n();
12.  }catch(Exception e){System.out.println("exception handled");}
13. }
14. public static void main(String args[]){
```

```
15.   Testthrows1 obj=new Testthrows1();
16.   obj.p();
17.   System.out.println("normal flow...");
18.  }
19. }
```

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

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

Output:

```
device operation performed
     normal flow...
```

B) If exception **occurs**

Testthrows4.java

```
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:

Basis of Differences	throw	throws
Definition	Java throw keyword is used to 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.
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.
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.
Declaration	**throw** is used within the method.	**throws** is used with the method signature.
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

TestThrow.java

```
public class TestThrow {
  //defining a method
  public static void checkNum(int num) {
    if (num < 1) {
      throw new ArithmeticException("\nNumber is negative, cannot calculate square");
    }
    else {
      System.out.println("Square of " + num + " is " + (num*num));
    } }
  //main method
  public static void main(String[] args) {
      TestThrow obj = new TestThrow();
      obj.checkNum(-3);
      System.out.println("Rest of the code..");
  } }
```

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

TestThrows.java

```
1.  public class TestThrows {
2.     //defining a method
3.     public static int divideNum(int m, int n) throws ArithmeticException {
4.        int div = m / n;
5.        return div;
6.     }
7.     //main method
8.     public static void main(String[] args) {
9.        TestThrows obj = new TestThrows();
10.       try {
11.          System.out.println(obj.divideNum(45, 0));
12.       }
13.       catch (ArithmeticException e){
14.          System.out.println("\nNumber cannot be divided by 0");
15.       }
16.
17.       System.out.println("Rest of the code..");
18.    }
19. }
```

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

TestThrowAndThrows.java

```
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.    } }
```

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:

Key	final	finally	finalize
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.
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**.
Functionality	(1) 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.
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

FinalExampleTest.java

```
1.  public class FinalExampleTest {
2.     //declaring final variable
3.     final int age = 18;
4.     void display() {
5.     // reassigning value to age variable  // gives compile time error
6.     age = 55;     }
7.     public static void main(String[] args) {
8.         FinalExampleTest obj = new FinalExampleTest();
9.     // gives compile time error
10.    obj.display();
11.    } }
```

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
```

References:

Java how to program tenth edition: paul deitel, Harvey deitel
javaTpoint
tutorialsPoint