Topics to cover:

1. Exception handling
2. Types of exception handling

1. Exception handling

What is Exception 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.

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

Main reason why an exception occurs:

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

Difference between Error and Exception

- **Error**: An Error indicates a serious problem that a reasonable application should not try to catch.
- **Exception**: Exception indicates conditions that a reasonable application might try to catch.

Advantage of Exception Handling

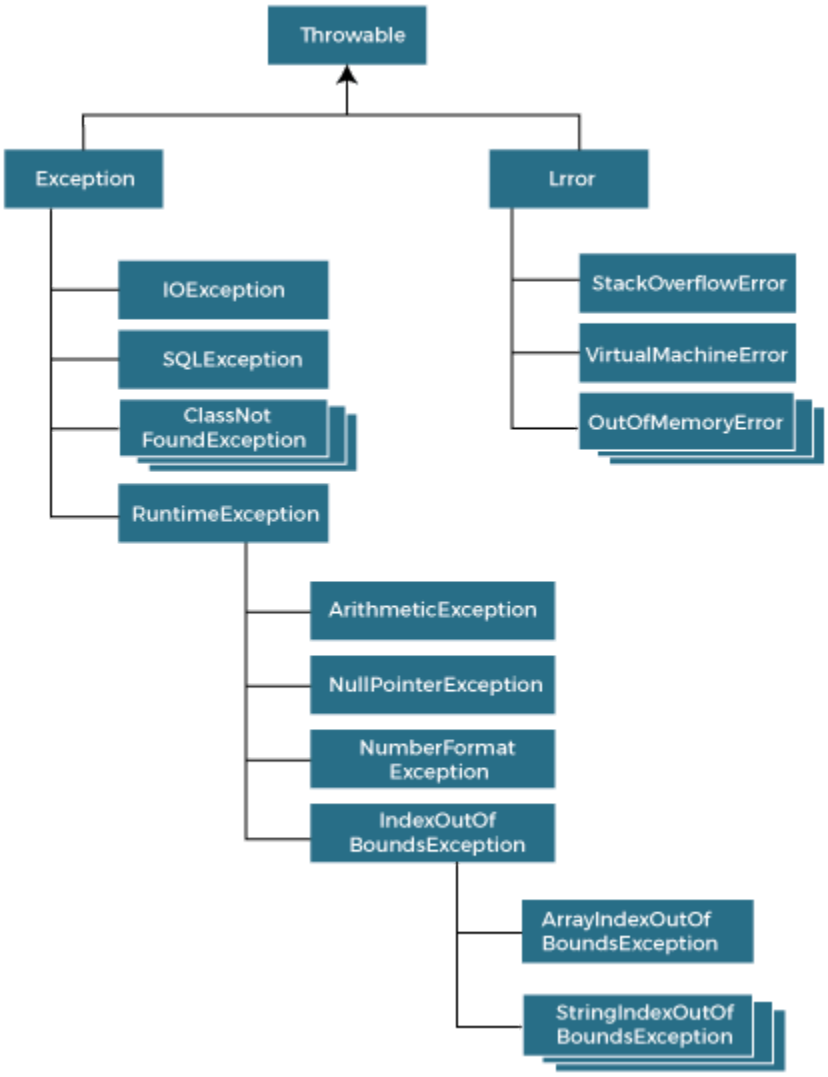
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

```
statement 1;

statement 2;

statement 3;

statement 4;

statement 5;  //exception occurs

statement 6;

statement 7;

statement 8;

statement 9;

statement 10;
```

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

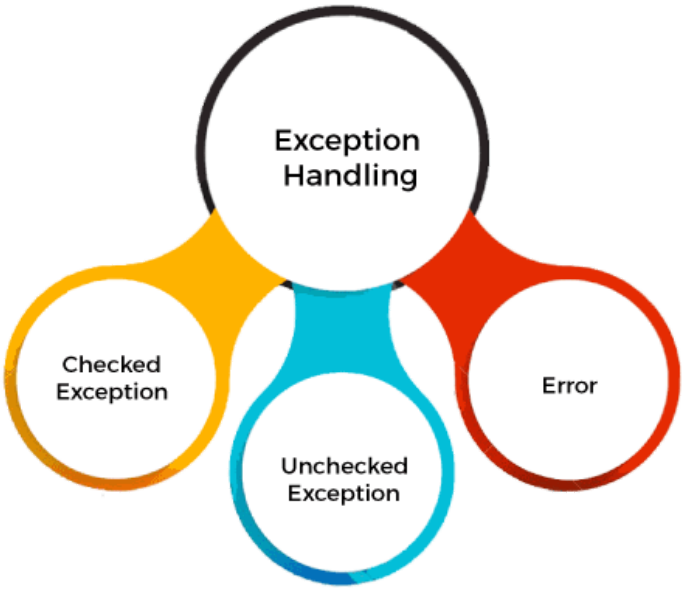


2. Types of exceptions

Types of Java Exceptions

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

1. Checked Exception
2. Unchecked Exception
3. Error



Difference between Checked and Unchecked Exceptions

1) Checked Exception

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

2) Unchecked Exception

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

3) Error

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

Java Exception Keywords

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

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by a 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 signatures.

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.

Program 1:

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

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

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.

```
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 it's 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
```

Exercise: Write a program to handle ArrayIndexOutOfBoundsException in any sorting algorithm.

Java try-catch block

Java try block

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

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

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

Syntax of Java try-catch

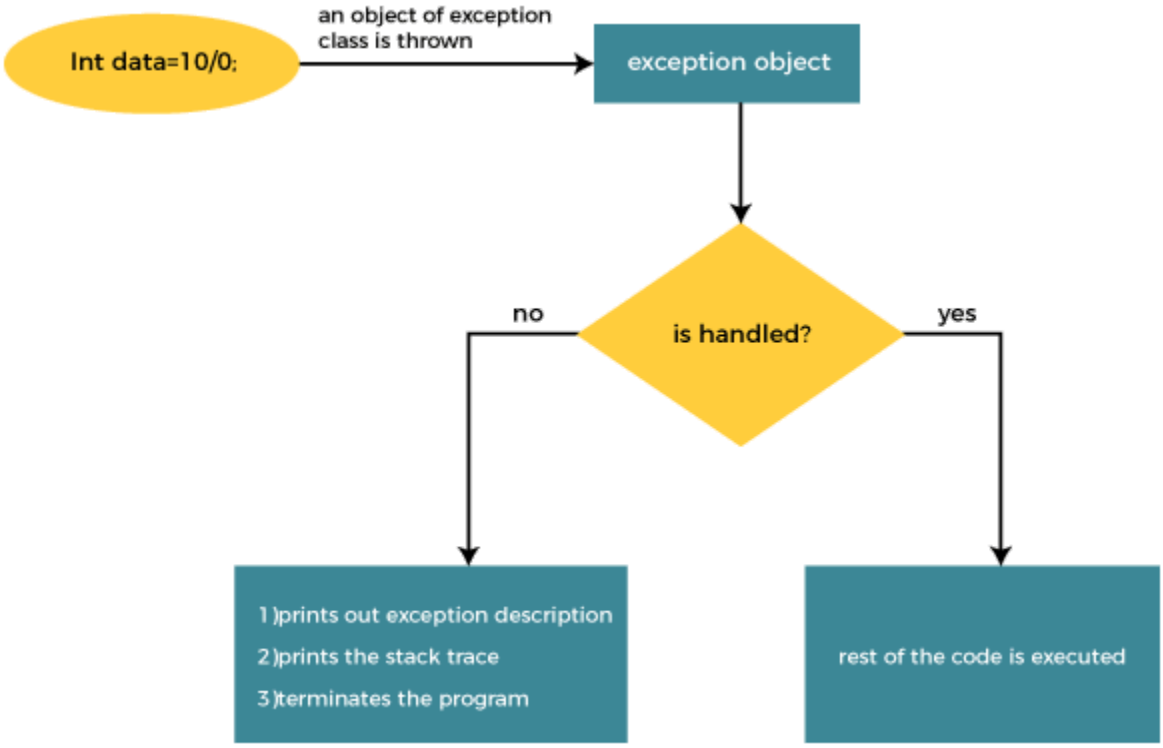
```
try{

//code that may throw an exception

}catch(Exception_class_Name ref){}
```

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 blocks 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

```
public class TryCatchExample1 {

  public static void main(String[] args) {

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

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

  }

}
```

Output:

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

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

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

Solution by exception handling

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

Example 2

```
public class TryCatchExample2 {
```

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

}
```

Output:

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

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

Example 3

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

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

Output:

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

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

Example 4

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

```
public class TryCatchExample4 {


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

Output:

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

Example 5

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

```
public class TryCatchExample5 {
   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
      // handling the exception
      catch(Exception e)
      {
            // displaying the custom message
         System.out.println("Can't divided by zero");
      }
   }

}
```

Output:

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

Example 6

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

```
public class TryCatchExample6 {
    public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try
        {
        data=i/j; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
             // resolving the exception in catch block
            System.out.println(i/(j+2));
        }
    }
}
```

Output:

25

Example 7

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

```
public class TryCatchExample7 {
   public static void main(String[] args) {
      try
      {
      int data1=50/0; //may throw exception
      }
          // handling the exception
      catch(Exception e)
      {
         // generating the exception in catch block
      int data2=50/0; //may throw exception
      }
   System.out.println("rest of the code");
   }
}
```

Output:

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

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

Example 8

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

```
public class TryCatchExample8 {

   public static void main(String[] args) {

      try

      {

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

      }

         // try to handle the ArithmeticException using
ArrayIndexOutOfBoundsException

      catch(ArrayIndexOutOfBoundsException e)

      {

         System.out.println(e);

      }

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

   }

}
```

Output:

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

Example 9

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

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

Output:

java.lang.ArrayIndexOutOfBoundsException: 10

rest of the code

Write a program to implement FileNotFoundException in two different ways.

Example 10

Let's see an example to handle checked exception.

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

public class TryCatchExample10 {
   public static void main(String[] args) {
      PrintWriter pw;
      try {
         pw = new PrintWriter("jtp.txt"); //may throw exception
         pw.println("saved");
      }
// providing the checked exception handler
 catch (FileNotFoundException e) {
         System.out.println(e);
      }
   System.out.println("File saved successfully");
   }
}
```

Output:

File saved successfully

throw and throws in Java

Java throw:

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exceptions. The throw keyword is mainly used to throw custom exceptions.

Syntax in Java throw

Example:

```
throw new ArithmeticException("/ by zero");
```

But this exception i.e., Instance must be of type Throwable or a subclass of Throwable.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception. If it finds a match, control is transferred to that statement, otherwise the next enclosing try block is checked, and so on. If no matching catch is found then the default exception handler will halt the program.

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:

throws is a keyword in Java that is used in the signature of a method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Syntax in Java throws

type method_name(parameters) throws exception_list

exception_list is a comma separated list of all the

exceptions which a method might throw.

In a program, if there is a chance of raising an exception then the compiler always warns us about it and compulsorily we should handle that checked exception, Otherwise, we will get a compile time error saying unreported exception XXX must be caught or declared to be thrown. To prevent this compile time error we can handle the exception in two ways:

- By using try catch
- By using the throws keyword

We can use the throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then the caller method is responsible to handle that exception.

Java throws Example

TestThrows.java

```
public class TestThrows {

  //defining a method

  public static int divideNum(int m, int n) throws ArithmeticException {

    int div = m / n;

    return div;

  }

  //main method

  public static void main(String[] args) {

    TestThrows obj = new TestThrows();

    try {

      System.out.println(obj.divideNum(45, 0));

    }

    catch (ArithmeticException e){

      System.out.println("\nNumber cannot be divided by 0");

    }



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

```
    }
}
```

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..
```

Difference between throw and throws in Java

The throw and throws are the concept of exception handling where the throw keyword throws 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 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.
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.

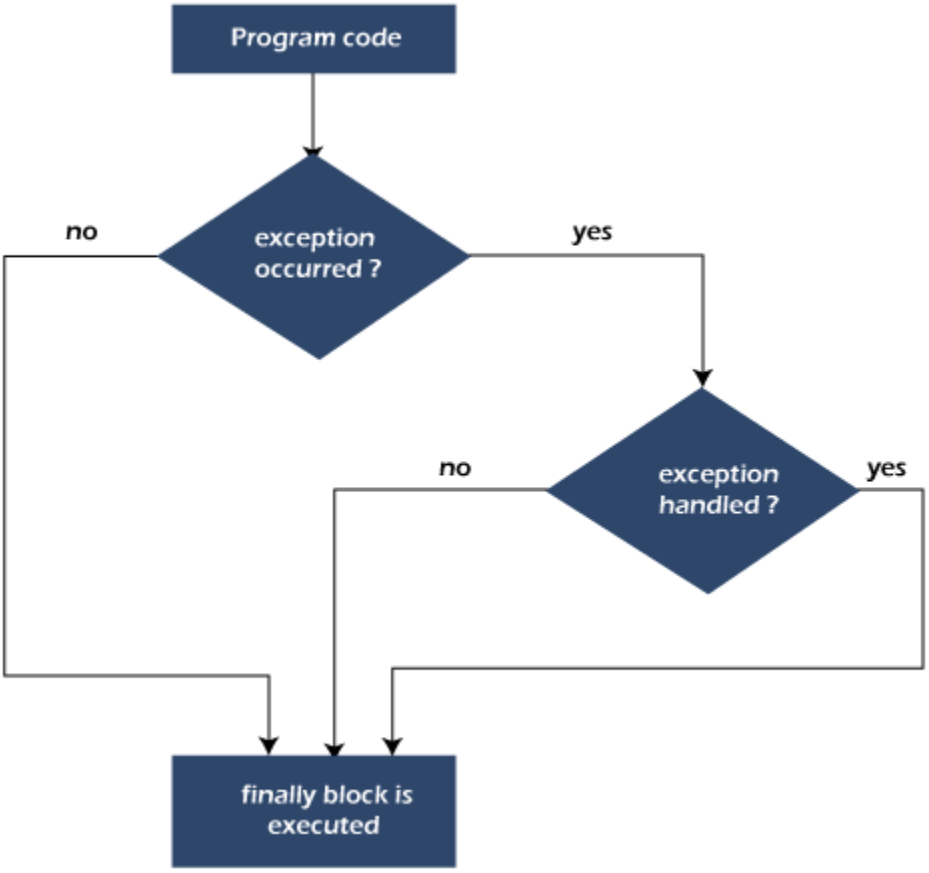
Java finally block

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 whether the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block



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

```
class TestFinallyBlock {

 public static void main(String args[]){

 try{

//below code do not throw any exception

  int data=25/5;

  System.out.println(data);

 }

 catch(ArithmeticException e){

System.out.println(e);

}

//executed regardless of exception occurred or not

 finally {

System.out.println("finally block is always executed");

}
```

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

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 the fillowing 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

```
        public class TestFinallyBlock1{
            public static void main(String args[]){


            try {


              System.out.println("Inside the try block");
```

```
  //below code throws divide by zero exception
 int data=25/0;
 System.out.println(data);
}
//cannot handle Arithmetic type exception
//can only accept Null Pointer type exception
catch(NullPointerException e){
  System.out.println(e);
}

//executes regardless of exception occurred or not
finally {
  System.out.println("finally block is always executed");
}
System.out.println("rest of the code...");
}
}
```

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

```
public class TestFinallyBlock2{

    public static void main(String args[]){

    try {

      System.out.println("Inside try block");

      //below code throws divide by zero exception

     int data=25/0;

     System.out.println(data);

    }
```

```
        //handles the Arithmetic Exception / Divide by zero exception
        catch(ArithmeticException e){
          System.out.println("Exception handled");
          System.out.println(e);
        }
        //executes regardless of exception occured or not
        finally {
          System.out.println("finally block is always executed");
        }
        System.out.println("rest of the code...");
        }
      }
```

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).

Confusion about finally keyword

In some examples, the usefulness of the finally keyword is not obvious. This results in a doubt whether finally is really useful or not.

Let us consider the following examples.

Example 1:

```
public class FinallyKeyword {
      public static void main(String[] args) {
            try {
                  int a=5;
                  int b=0;
                  int c=a/b;
            }
            catch(Exception e) {
                  System.out.println(e);
            }
            finally {
                  System.out.println("Rest of the code");
            }
      }
}
```

Output:

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

Example 2:

```
public class FinallyKeyword {
	public static void main(String[] args) {
		try {
			int a=5;
			int b=0;
			int c=a/b;
		}
		catch(Exception e) {
			System.out.println(e);
		}

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

	}
}
```

Output:

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

We notice that example1 and example2 both have the same output even though they differ in the usage of the finally block. The former uses finally and the latter does not. One might ask that if the statement "`Rest of the code`" is executed with or without using the finally keyword then why is it used? The answer is, there are some situations where the finally block is really useful. So we have better examples as follows.

Example 3:

```
public class FinallyKeyword2 {

	public static int division() {
		try {
			int a=5;
			int b=0;
			int c=a/b;
			return c;
		}
		catch(Exception e) {
```

```
                System.out.println(e);
            }
            System.out.println("Rest of the code");
            return 0;
        }
        public static void main(String[] args) {
            division();
        }
}
```

Output:

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

In the above program, an exception occurs at "`int c=a/b;`". Therefore, it stopped there and we could not reach the "`return c`" statement. It directly goes to the catch statement and then the rest part for that prints "`Rest of the code`". Note that we did not print anything in the main method. We just called the division() method which can return a value but it won't be printed. You can print if you want.

Now we change the program to the following one:

Example 4:

```
public class FinallyKeyword3 {

    public static int division() {
        try {
            int a=5;
            int b=1;
            int c=a/b;
            return c;
        }
        catch(Exception e) {
            System.out.println(e);
        }
        System.out.println("Rest of the code");
        return 0;
```

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

Output:

.....

There is no output in the above program. The reason is, b is changed to 1 which results in a non-zero value after division. After successful division, we reach the return statement and the value is returned to the main method. This is why the remaining part of the division() is not executed (i.e, "Rest of the code" is not printed).

Now we further change the program to the following:

Example 5:

```
public class FinallyKeyword2 {

	public static int division() {
		try {
			int a=5;
			int b=1;
			int c=a/b;
			return c;
		}
		catch(Exception e) {
			System.out.println(e);
		}
		finally {
			System.out.println("Rest of the code");
		}
		return 0;
	}
	public static void main(String[] args) {
		division();
	}
```

Output:

Rest of the code

In the above program we put the print statement in the division() method inside the finally block. You can notice that the output has changed to “Rest of the code” from nothing. Here the finally block plays an important role. It stops the return statement in the try block from directly moving to the main method. Instead, the finally block forces the statement “Rest of the code” even after the return statement.