# Unit-1 Object Oriented Programming with Java

Outcome of this unit-: Mastering this unit enables you to handle runtime errors effectively, manage file input/output operations, and implement multithreading for concurrent execution. You'll write robust, efficient, and error-resilient Java applications, ensuring proper resource management, synchronization, and communication between threads.

## 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.

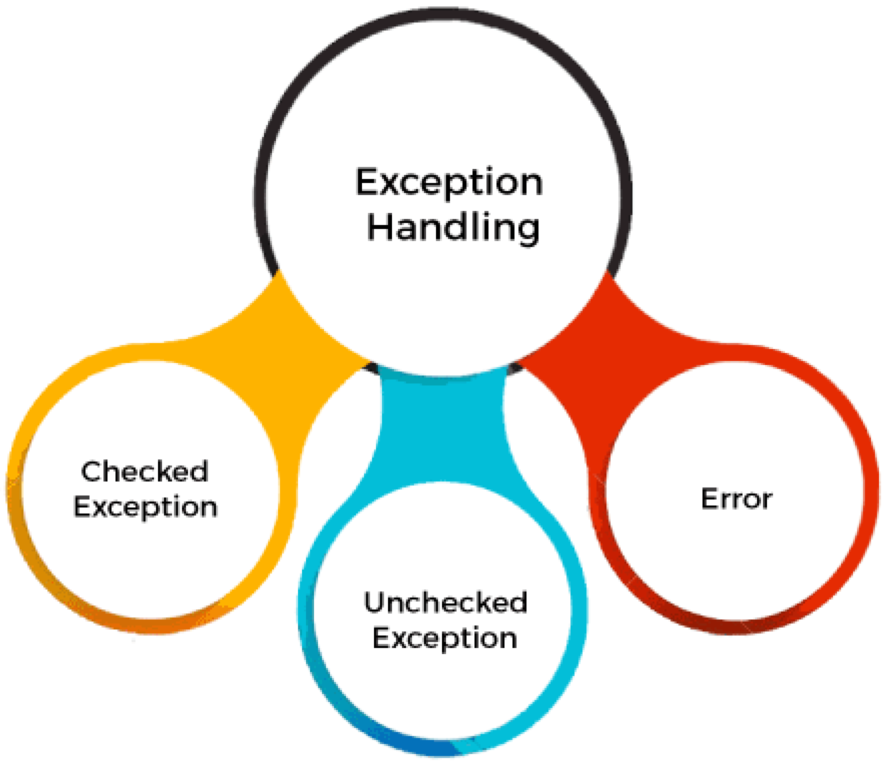
## Types of Java Exceptions

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

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

**Unit-1 Object Oriented Programming with Java**



## 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.

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

## Java Exception Keywords

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

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

## Java try block

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

## Syntax of Java try-catch

```
try{
```

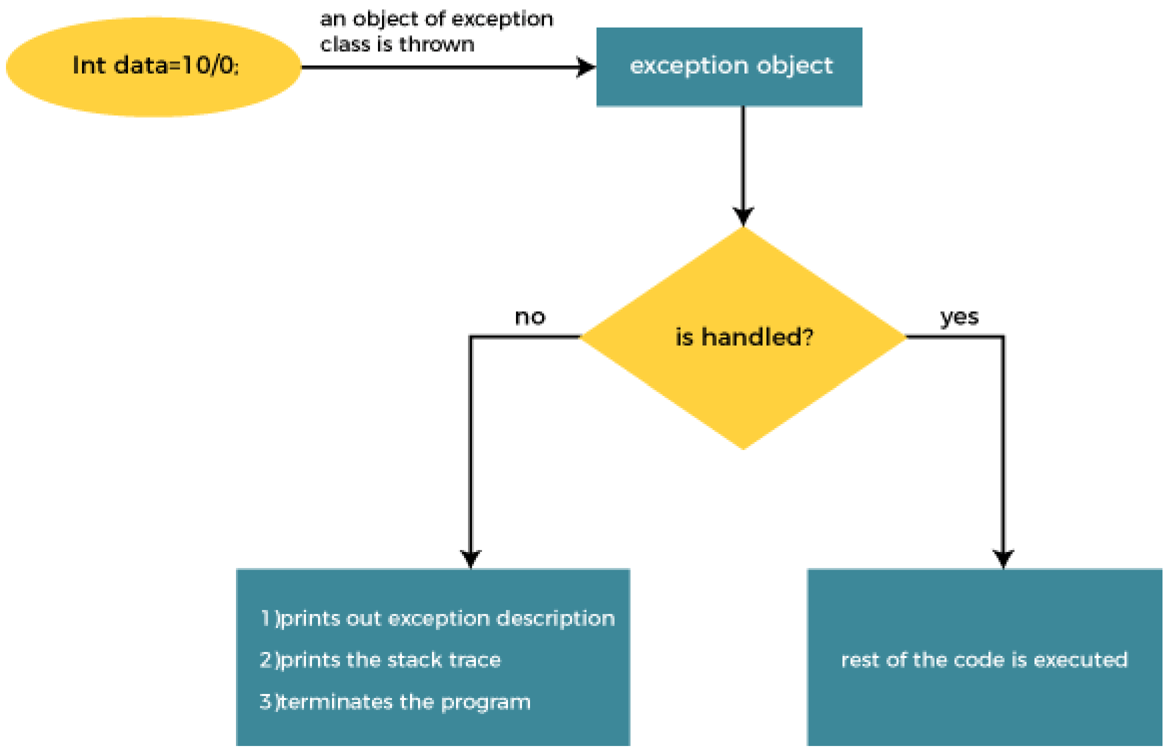
```
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

## 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 block with a single try block.

## Internal Working of Java try-catch block


Int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

**Unit-1 Object Oriented Programming with Java**

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

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

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

### Problem without exception handling

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

### Example 1

**TryCatchExample1.java**

```
public class TryCatchExample1 {

    public static void main(String[] args) {

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

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

    }
```

```
        }
```

Test it Now

**Output:**

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

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

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

## Solution by exception handling

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

### Example 2

**TryCatchExample2.java**

```
  public class TryCatchExample2 {


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

   }
```

**Test it Now**

**Output:**

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

## Java finally block

**Java finally block** is a block used to execute important code such as closing the connection, etc.

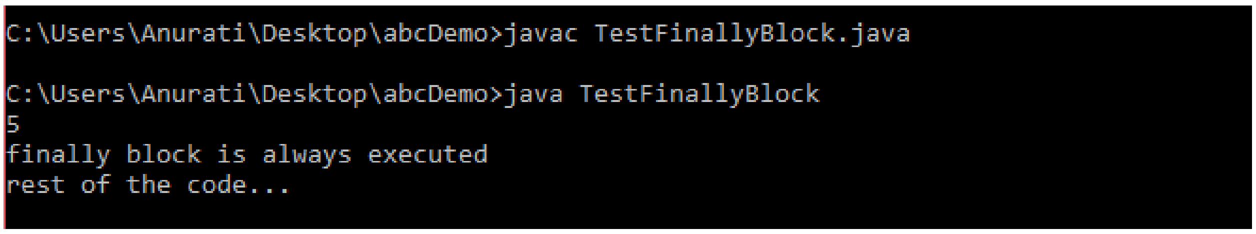
Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

**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 won't be executed

   catch(NullPointerException 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...
```

## 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.

```
class TestException

{


        public int exception( int x,int y)

        {

                int r;

                try

                {


                r=x/y;

                return r;


                }

                catch(ArithmeticException ae)

                {
                    throw ae;
                }
          }
public static void main(String [] args)
{
int x,y,r;
TestException ob=new TestException();
try
{
x=Integer.parseInt(args[0]);
y=Integer.parseInt(args[1]);
r=ob.exception(x,y);

System.out.println("add of two no "+r);
}
catch(NumberFormatException ae)
{
      System.out.println("both should be Number");
}
catch(ArithmeticException ae)
{
        System.out.println("second Nubmer must no be zero");
}
catch(Exception ae)
{
        System.out.println("Both the input should be not zero no");
}
}
}
```

## 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

```
return_type method_name() throws exception_class_name{
//method code
}
```

## 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.

## Java throws Example

**Testthrows1.java**

```
import java.io.IOException;
class Testthrows1
{
  void m()throws IOException
{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException
{
          m();
        }
        void p()
      {
         Try
      {
          n();
    }
catch(Exception e)
{
System.out.println("exception handled");
}
  }
  public static void main(String args[])
{
    Testthrows1 obj=new Testthrows1();
    obj.p();
    System.out.println("normal flow...");
  }
}
```

Test it Now

**Output:**

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

## Java I/O Tutorial

**Java I/O** (Input and Output) is used *to process the input* and *produce the output*.

### Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream

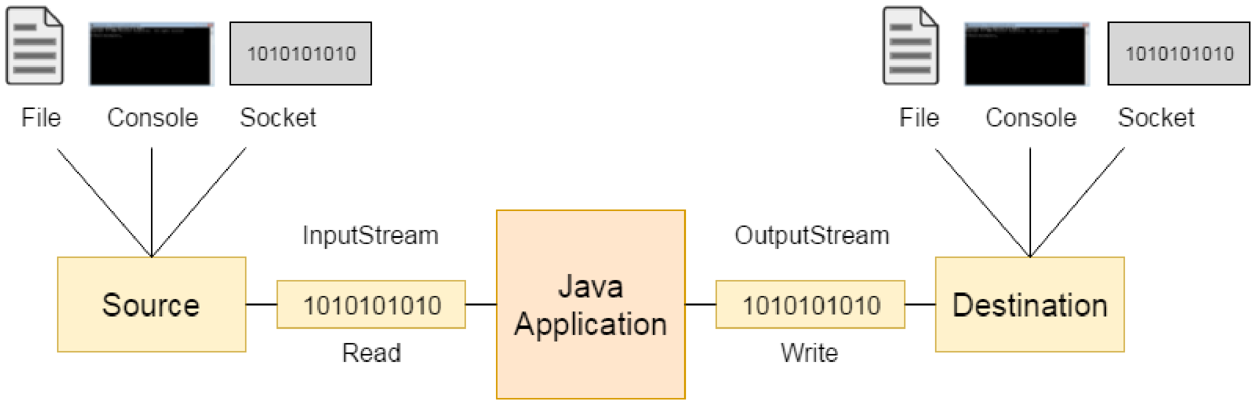
### OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

### InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.



### OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

### ByteStream Classes in Java

ByteStream classes are used to read bytes from the input stream and write bytes to the output stream. In other words, we can say that ByteStream classes read/write the data of 8-bits. We can store video, audio, characters, etc., by using ByteStream classes. These classes are part of the java.io package.

The ByteStream classes are divided into two types of classes, i.e., InputStream and OutputStream. These classes are abstract and the super classes of all the Input/Output stream classes.

### InputStream Class

The InputStream class provides methods to read bytes from a file, console or memory. It is an abstract class and can't be instantiated; however, various classes inherit the InputStream class and override its methods. The subclasses of InputStream class are given in the following table.

### OutputStream Class

The OutputStream is an abstract class that is used to write 8-bit bytes to the stream. It is the superclass of all the output stream classes. This class can't be instantiated; however, it is inherited by various subclasses that are given in the following table.

## CharacterStream Classes in Java

The java.io package provides CharacterStream classes to overcome the limitations of ByteStream classes, which can only handle the 8-bit bytes and is not compatible to work directly with the Unicode characters. CharacterStream classes are used to work with 16-bit Unicode characters. They can perform operations on characters, char arrays and Strings.

However, the CharacterStream classes are mainly used to read characters from the source and write them to the destination. For this purpose, the CharacterStream classes are divided into two types of classes, I.e., Reader class and Writer class.

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

## Reader Class

Reader class is used to read the 16-bit characters from the input stream. However, it is an abstract class and can't be instantiated, but there are various subclasses that inherit the Reader class and override the methods of the Reader class. All methods of the Reader class throw an IOException. The subclasses of the Reader class are given in the following table.

## File Operations

We can perform the following operation on a file:

- Create a File
- Write to a File
- Read from a File

## Create a File

**Create a File** operation is performed to create a new file. We use the **createNewFile()** method of file. The **createNewFile()** method returns true when it successfully creates a new file and returns false when the file already exists.

Example :-

```
//Importing File class

import java.io.File;

// Importing the IOException class for handling errors
```

```
import java.io.IOException;


class CreateFile

{

    public static void main(String[] args)

    {

        try

        {

            // Creating an object of a file

            File f0 = new File("E:FileOperationExample15.txt");

            if (f0.createNewFile())

            {

                System.out.println("File " + f0.getName() + " is created successfully.");

            }

            else

            {

                System.out.println("File already exists in the directory.");

            }

        }

        catch (IOException exception)

        {
            System.out.println("An unexpected error occurred.");
            exception.printStackTrace();
        }
    }
}
```

## Write to a File

The next operation which we can perform on a file is **"writing into a file"**. In order to write data into a file, we will use the **FileWriter** class and its **write()** method together. We need to close the stream using the **close()** method to retrieve the allocated resources.

Let's take an example to understand how we can write data into a file.

**WriteToFile.java**

```
package com.javatpoint;
import java.io.FileWriter;
public class FileWriterExample {
    public static void main(String args[]){
         try{
           FileWriter fw=new FileWriter("E:fileFileOperationExample15.txt");
```

```
          fw.write("Welcome to javaTpoint.");

          fw.close();

         }catch(Exception e){System.out.println(e);}

         System.out.println("Success...");

    }

}
```

## Read from a File

The next operation which we can perform on a file is **"read from a file"**. In order to write data into a file, we will use the **Scanner** class. Here, we need to close the stream using the **close()** method. We will create an instance of the Scanner class and use the **hasNextLine()** method **nextLine()** method to get data from the file.

Example:-

```
// Importing the File class

import java.io.File;

// Importing FileNotFoundException class for handling errors

import java.io.FileNotFoundException;

// Importing the Scanner class for reading text files

import java.util.Scanner;


class ReadFromFile {

   public static void main(String[] args) {
        try {
            // Create f1 object of the file to read data
            File f1 = new File("E:fileFileOperationExample15.txt");
            Scanner dataReader = new Scanner(f1);
            while (dataReader.hasNextLine()) {
                String fileData = dataReader.nextLine();
                System.out.println(fileData);
            }
            dataReader.close();
        } catch (FileNotFoundException exception) {
            System.out.println("Unexcpected error occurred!");
            exception.printStackTrace();
        }
    }
}
```

## Delete a File

The next operation which we can perform on a file is **"deleting a file"**. In order to delete a file, we will use the **delete()** method of the file. We don't need to close the stream using the **close()** method because for deleting a file, we neither use the FileWriter class nor the Scanner class.

Let's take an example to understand how we can write data into a file.

**DeleteFile.java**

```
// Importing the File class

import java.io.File;

// Importing FileNotFoundException class for handling errors

import java.io.FileNotFoundException;

// Importing the Scanner class for reading text files

import java.util.Scanner;


class ReadFromFile {

    public static void main(String[] args) {

        try {

            // Create f1 object of the file to read data

            File f1 = new File("E:fileFileOperationExample15.txt");

            Scanner dataReader = new Scanner(f1);

            while (dataReader.hasNextLine()) {

                String fileData = dataReader.nextLine();

                System.out.println(fileData);

            }

            dataReader.close();

        } catch (FileNotFoundException exception) {

            System.out.println("Unexcpected error occurred!");
            exception.printStackTrace();
        }
    }
}
```

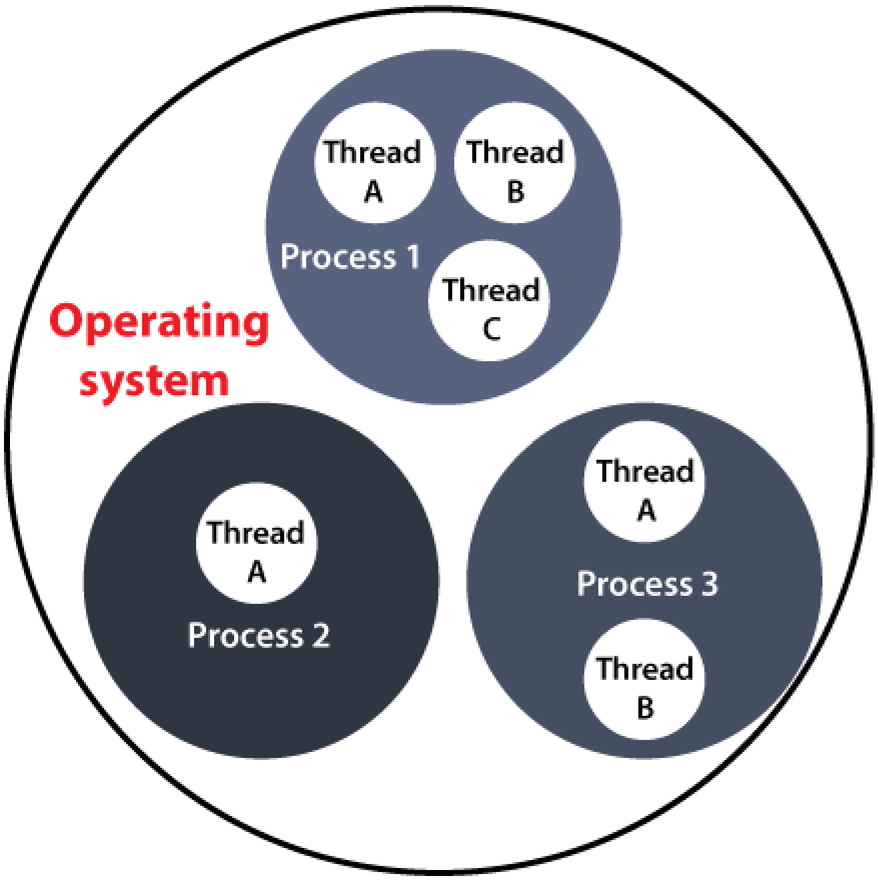
**Explanation:**

In the above code, we import the **File** class and create a class **DeleteFile**. In the main() method of the class, we create **f0** object of the file which we want to delete. In the **if** statement, we call the **delete()** method of the file using the f0 object. If the delete() method returns true, we print the success custom message. Otherwise, it jumps to the else section where we print the unsuccessful custom message.

All the above-mentioned operations are used to read, write, delete, and create file programmatically.

## Thread Concept in Java

A **Thread** is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.



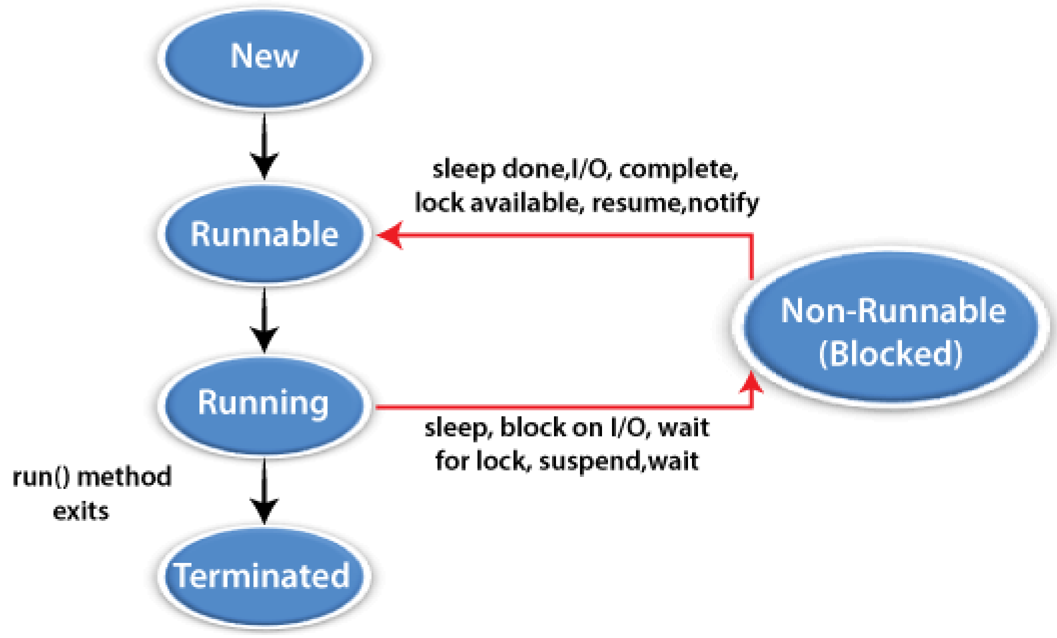
In a simple way, a Thread is a:

- Feature through which we can perform multiple activities within a single process.
- Lightweight process.
- Series of executed statements.
- Nested sequence of method calls.

## Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated


New
Runnable
Running
Terminated
Non-Runnable (Blocked)
sleep done,I/O, complete, lock available, resume,notify
sleep, block on I/O, wait for lock, suspend,wait
run() method exits

**New:** Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

**Active:** When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the

thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.

- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

**Blocked or Waiting:** Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

## Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

### Types of Synchronization

There are two types of synchronization

1. Process Synchronization

2. Thread Synchronization

Here, we will discuss only thread synchronization.

### Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

### Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

### Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.