What Is a Java Thread?

A thread is actually a lightweight process. Unlike many other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run **concurrently**. Each part of such a program is called a thread and each thread defines a separate path of the execution. Thus, multithreading is a specialized form of multitasking.

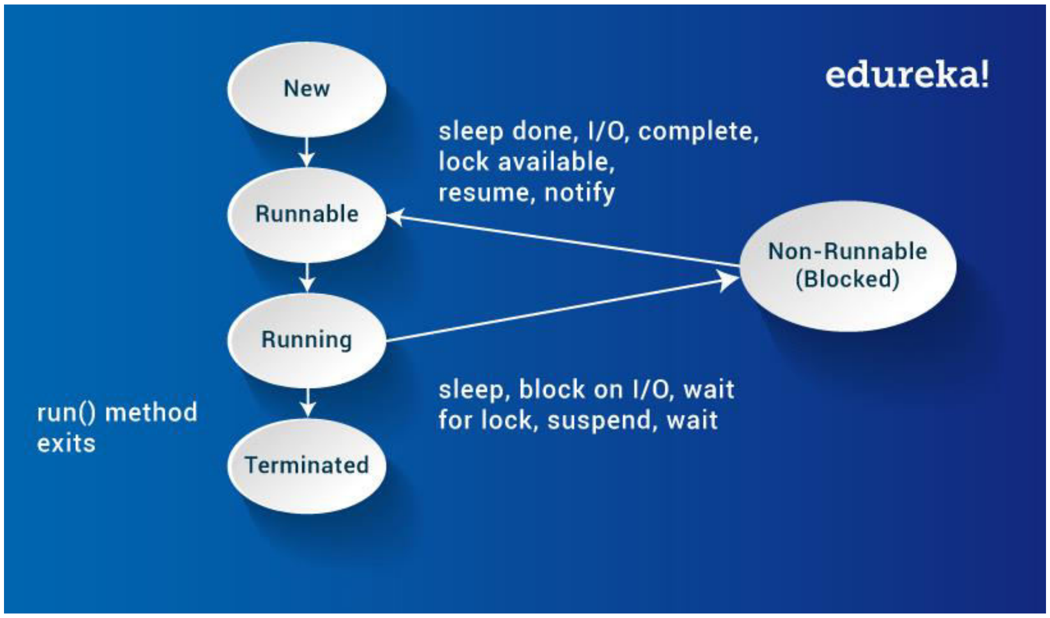
The next concept in this Java Thread blog is integral to the concept of Threads and Multithreading.

The Java Thread Model

The Java run-time system depends on threads for many things. Threads reduce inefficiency by preventing the waste of CPU cycles.

Threads exist in several states. The following are those states:

- **New -** When we create an instance of Thread class, a thread is in a new state.
- **Running -** TheJava thread is in running state.
- **Suspended** - A running thread can be **suspended**, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
- **Blocked** - A java thread can be blocked when waiting for a resource.
- **Terminated** - A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.


edureka!
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

So, this was all about the Java Thread states. Now, let's jump to the most important topic of Java threads i.e. thread class and runnable interface. We will discuss these one by one below.

Multithreading in Java: Thread Class and Runnable Interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, **Runnable**. To create a new thread, your program will either extend **Thread** or **implement** the **Runnableinterface**.

The Thread class defines several methods that help manage threads. The table below displays the same:

Method	Meaning
getName	Obtain thread’s name
getPriority	Obtain thread’s priority
isAlive	Determine if a thread is still running
join	Wait for a thread to terminate
run	Entry point for the thread
sleep	Suspend a thread for a period of time
start	Start a thread by calling its run method

Now let's see how to use a Thread which begins with the **main java thread** that all Java programs have.

Main Java Thread

Here, I'll show you how to use Thread and Runnable interface to create and manage threads, beginning with the **main java thread,** that all Java programs have. So, let's discuss the main thread.

Why Is Main Thread So Important?

- Because this thread affects the other 'child' threads
- Because it performs various shutdown actions
- It is created automatically when your program is started.

So, that's the main thread. Let's see how we can create a java thread?

How to Create a Java Thread?

Java lets you create thread in following two ways:

- By **implementing** the **Runnableinterface**.
- By **extending** the **Thread**

Let's look at how both ways help in implementing the Java thread.

Runnable Interface

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

To implement Runnable interface, a class need only implement a single method called run( ), which is declared like this:

```
public void run( )
```

Inside run( ), we will define the code that constitutes the new thread. Example:

```
public class MyClass implements Runnable {
public void run(){
System.out.println("MyClass running");
   }
}
```

To execute the run() method by a thread, pass an instance of MyClass to a Thread in its constructor (*A* **constructor in Java** *is a block of code similar to a method that's called when an instance of an object is created*). Here is how that is done:

```
Thread t1 = new Thread(new MyClass ());
t1.start();
```

When the thread is started it will call the run() method of the MyClass instance instead of executing its own run() method. The above example would print out the text "**MyClass running** ".

Extending Java Thread

The second way to create a thread is to create a new class that extends Thread, then override the run() method and then to create an instance of that class. The run() method is what is executed by the thread after you call start(). Here is an example of creating a Java Thread subclass:

```
public class MyClass extends Thread {
     public void run(){
     System.out.println("MyClass running");
   }
}
```

To create and start the above thread you can do so like this:

```
MyClass t1 = new MyClass ();
T1.start();
```

When the run() method executes it will print out the text " **MyClass running** ".

So far, we have been using only two threads: the **main** thread and one **child** thread. However, our program can affect as many threads as it needs. Let's see how we can create multiple threads.

Creating Multiple Threads

```
class MyThread implements Runnable {
String name;
Thread t;
    MyThread String thread){
    name = threadname;
    t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start();
}
public void run() {
 try {
     for(int i = 5; i > 0; i--) {
     System.out.println(name + ": " + i);
      Thread.sleep(1000);
}
}catch (InterruptedException e) {
     System.out.println(name + "Interrupted");
}
     System.out.println(name + " exiting.");
}
}
class MultiThread {
public static void main(String args[]) {
     new MyThread("One");
     new MyThread("Two");
     new NewThread("Three");
try {
     Thread.sleep(10000);
} catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
}
      System.out.println("Main thread exiting.");
      }
}
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
```

```
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

This is how multithreading in Java works.