# L50

Friday, February 4, 2022 10:39 PM

# Multi-Threading, Multitasking & Multiprocessing, Understanding Threads & its States

# Multi-Threading, Multitasking & Multiprocessing

Q1. The word Multitasking is applied to the operating system performing multiple tasks simultaneously. These tasks are called processes.

A process is a running instance of a program code, which uses resources like Memory, CPU cycles etc.

While you are reading this text, you already have many tasks (processes) being simultaneously run by your OS (operating system) like the browser instance, clock.. etc.

When we execute a Java class, a JVM (Java Virtual Machine) instance is created and launched as a separate program, which in turn executes the code in our Java class.

Java language has support for multithreading. Meaning we can write code which will create multiple threads. A thread is an independent execution flow that can run simultaneously along with its parent which launched it.

The way processes are scheduled and managed by the operating system, threads that are created and run in a JVM have a one-to-one mapping to a corresponding OS threads (called native threads). For more details click here and read the **Threading Model** section..

In Java, threads have a priority which can be set. When a new thread is created, by default it receives the priority of its parent thread. This priority is used in thread scheduling.

Individual processes in an operating system have their own memory space. While the threads launched in a JVM, share the same context and the memory space used by the JVM. For this reason threads are also considered light-weight when compared to processes. though such a statement is debatable.

Until Java version 1.3, Java used to have **Green Threads** which were light-weight and were scheduled by the JVM. Green threads **were not taking advantage** of multiple processors available in a multi-processor machine. From **Java version 1.3** and later versions we only have native threads, which are scheduled by the native OS and provide performance benefits by taking advantage of the multiple processors available on a multi-processor machine.

See and retype the below example which creates two threads in Java. We will learn more about the details of this code in the later sections.

Note Thread.sleep(1000); statement causes the execution to pause for 1000 milliseconds (a second).

```java
package q11340;
public class SimpleThreadDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter c1 = new Counter("Ganga");
        Counter c2 = new Counter("Yamuna");
        Thread t1 = new Thread(c1);
        Thread t2 = new Thread(c2);
        t1.start();
        t2.start();
        System.out.println("started t1 and t2 threads");
        t1.join();
        System.out.println("t1 has completed. t1.isAlive() = " + t1.isAlive());
        t2.join();
        System.out.println("t2 has completed. t2.isAlive() = " + t2.isAlive());
    }
}
class Counter implements Runnable {
    private String name;
    public Counter(String name) {
        this.name = name;
    }
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(name + " : " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

# Understanding Threads & its States

Q1. In Java a thread can be created and executed in two ways. One is by writing a class which extends the Thread class and calling the start() method.

**Approach #1**

```
public class MyClass extends Thread {
        public void run() {
  ...
        }
}
```

execute the thread by calling:

**new MyThread().start();**

The second approach is by writing a class which implements the Runnable interface. Then we create a Thread object by passing an instance of our class into the Thread's constructor and call the start() method.

**Approach #2**

```
public class MyClass implements Runnable {
        public void run() {
  ...
```

```
        }
}
```
execute the thread by calling:
MyClass mc = new MyClass();
**new Thread(mc).start();**
Among the two, the **second approach** which implements a Runnable interface is recommended.

See and retype the below code.

A call to Thread.sleep(long milliseconds) method causes the current execution to pause for the given duration in milliseconds. The **sleep** method throws an InterruptedException if it is interrupted by any other thread. Since InterruptedException is a checked exception, we have to handle the exception using a try-catch block.

```java
package q11341;
public class SleepDemo {
    public static void main(String[] args) {
        try {
            System.out.println("About to take a short nap. Start counting from 1 to 2.");
            Thread.sleep(1000);
            System.out.println("Fresh after 2 seconds of good sleep!");
            Thread.sleep(1000);
            System.out.println("Very fresh after sleeping for 2 more seconds!!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

## Q2. A thread can be in one of the below mentioned states:

1. NEW - when a thread is just created and is not started yet (meaning the start() method is not yet called on it).
2. RUNNABLE - when the start method is called and the thread is executing the code in run() method.
3. BLOCKED - when a thread is unable to proceed with execution because it is waiting for a monitor lock (we will learn more about locks la ter.)
4. WAITING - when a thread is waiting indefinitely for another thread to perform a particular action.
5. TIMED_WAITING - when a thread that is waiting for another thread to perform an action for a specified waiting time, after which it will resum e.
6. TERMINATED - when a thread finishes its execution.

Note that when a new Thread is created it does not start automatically. At that moment it is in the **NEW** state. And after a thread's state changes to **TERMINATED**, it cannot be started again.

**Note**: We should never use the stop() method provided in the **Thread** class as it is deprecated and can lead to unexpected results. Instead we should manually write the code to stop a running thread.

See and retype the below code.

We should always remember that the main method is also executed in a thread by the JVM.

The t1.join() statement causes the thread which is executing the main method to wait till the thread t1 terminates, meaning completes its execution.

```java
package q11342;
public class SimpleThreadDemo2 {
    public static void main(String[] args) throws InterruptedException {
        Counter c1 = new Counter("Ganga");
        Thread t1 = new Thread(c1);
        System.out.println("Before start() method call t1.getState() = " + t1.getState());
        System.out.println("Before start() method call t1.isAlive() = " + t1.isAlive());
        t1.start();
        System.out.println("After start() method call t1.getState() = " + t1.getState());
        System.out.println("After start() method call t1.isAlive() = " + t1.isAlive());
        t1.join();
        System.out.println("After t1 has terminated t1.getState() = " + t1.getState());
        System.out.println("After t1 has terminated t1.isAlive() = " + t1.isAlive());
    }
}
class Counter implements Runnable {
    private String name;
    public Counter(String name) {
        this.name = name;
    }
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(name + " : " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```