# **Multi-threading, synchronization in programming using Java: Part 1**

This series is to discuss about multi-threading, and synchronization of programing using Java language. In this part 1 we will explore about thread and multi-threads. But before we delve detail into the matter we need to have a little concept or restore our concepts a little bit on processes and threads.

Both process and thread are quite similar and associated, yet they are not same. An execution of program is known as process where as a thread is driven in the environment of the process, a lightweight process is what it is called sometimes. Both processes and threads are similar that they are independent set of executions.

**Process in software:**

* An instance of a program is called a process. (e.g. MS Word and MS Paint are two different processes)
* A process is stored in the RAM
* Processes are isolated and does not share memory among themselves.
* A process can contain multiple threads
* Multiple processes can be executed parallel in a true multiple processor system, whereas in uni-processor system a process scheduling algorithm is used to achieved to maintain different processes
* One Java application runs in one process

**Thread in software:**

* A thread is a lightweight subset of process. (Editing and auto-saving of data in MS Word(single process) are done by two separate threads.)
* Threads are part of processes and remains in them.
* Multiple threads can share memory among themselves.
* Every process contains minimum one thread.
* In multiple processor system true multi-threading can be achieved. In case of uni-processor thread scheduler maintains the execution of different threads.
* One Java application can have multiple of threads.

**Thread/multi-thread and its requirements:**

If an application (process) contains single thread, every application/steps of application occurs sequentially one after another. But in many cases this might leads to some problems. Like one may want to see the current share prices of a particular company while downloading the historical data of it. In a single thread while the wait for the time to download historical data, might take some time, the current prices may vary a lot. Similarly while monitoring the current prices the data cannot be downloaded. Thus we need concurrency of jobs and that’s where multi-threading comes to help and we use it.

Few important things about .equals method:

* It only compares what is written to compare
* If a class does not override the .equals() method of it, then by default it goes to the equals(Object o)method of the closest parent class that has overridden this method.
* If neither of any parent classes have provided an overridden method of .equals(), then by default it points to the method from the ultimate parent class, Object.

As per the Object API this works same as ==; that is, it returns true *if and only if* both variables refer to the same object, if their references are one and the same. Thus it will test for **object equality** and not **functional equality**.

* If a class overrides the .equals method, the hashcode() method is necessary to be overridden too.

**Basics of thread:**

Thread can be defined in Java in two ways.

1. An instance of the class java.lang.Thread or a class implementing Runnable interface.
2. A thread of execution

An instance of thread is basically a simple java object like any other objects having the inherited properties from the Thread class. It has variables, methods and stays in the heap.

Whereas a thread of execution is a lightweight process having its own call stack. Each thread have their own personal call stack or individual call stack per thread.

One interesting fact is whether or not a developer creates thread, at least single thread runs in the background always when a Java program runs. The main() method is Java which initiates the program creates a call stack by itself and of course the main() method is the first method in the stack.

Whenever a new thread is generated a new ‘call stack’ gets created. All the methods of the new thread stays in the new call stack. Every call stacks created runs parallel making the program to perform multiple tasks concurrently. JVM is responsible for scheduling the running of multiple threads. JVM differs from one to another. Thread scheduling mechanism varies from one JVM to another. Like Oracle vended JVM and open source JVM may vary in implementation. In some cases the JVM hands over the scheduling to the underlying native –OS. *So when it is regarding thread, very few things can be guaranteed. Same program can gives different results when runs on different machines*. So it is recommended that when a program is written with thread handling it should not be coded dependent on one particular JVM. Different JVMs can run threads in profoundly different ways resulting in different output. Some may offer each thread equal chances, some may wait till one of them ends then only any one of the other thread will get chances to run. Developers must be careful to write a good, safe multi thread application. Also *‘daemon thread’* and ‘*user thread’* are not the same.

**Creation of Thread:**

To create a thread first and foremost an instance of thread object is required. The Thread class is responsible for managing threads which includes creation, start, pause, re-start and termination. The Thread class has various methods, the most important of them being the run() method – where all the actions of separate thread, starts and happens. The new call stack and the concurrency happens from the run() method.

So, to start a new user thread run() method needs to be implemented and it can be done in 2 ways.

1. By extending the class java.lang.Thread
2. Implementing the interface Runnable (Basically java.lang.Thread internally implements Runnable)

Let’s have a look at the below code for the two different types of implementation.

*The first one will be like this:*

class MyThread extends Thread{

public void run(){

System.out.println(“Inside new thread”);

}

*second approach will look like this:*

Class MyThread implements Runnable*{*

public void run(){

System.out.println(“Inside new thread”);

}

When both the run method will be invoked properly they will give the same output in running in a separate thread. But by following the OOPS principles it is recommended to use the interface Runnable rather than the inheritance for the following reasons mainly:

* Inheritance or sub classing should be reserved for the specialized version of more generalized super class
* If a class does not extensively use specialized specific version of Thread implementing Runnable interface allows the class to extend or implement other classes.
* Extending Thread class is recommended when a user Thread class is needed for some specific particular behavior.

run() method is free to overload, but that will act as a normal java method, without initializing a new thread when invoked.

**Instantiating a Thread:**

In simple words, to execute a thread an instance of Thread class is required. Irrespective of whether the user Thread class is created by extending java.lang.Thread or by implementing Runnable interface a Thread object(worker) is required to invoke the run() method (the job to be done).

*When the user Thread class is created by extending a Thread class it is relatively simple to get the Thread object. For example*,

MyThread t = new MyThread();

With this reference variable t we can now invoke the run() method, which will create a new thread and its own call stack.

*When the user Thread class is created by implementing Runnable interface the instantiation is slightly different.*

To run a separate Thread an instance of Thread (*worker*) is still required. But in this case the run() method (the job to be done) is not the Thread class but the class implementing the Runnable interface. Here we still need to create a Thread instance but by passing the Runnable interface object to the constructor, so that the Thread we createdknows which run() method to call. (*The worker must know the job to be done*). The Runnable passed to the Thread constructor is known as the ‘*target*’ or ‘*target*’ Runnable. The code to execute this will be as below.

MyRunnable r = new MyRunnable();

Thread t = new Thread(r)

Now the Thread instance t knows that it has to execute the run() method implemented in the Runnable interface. If no target is passed, the Thread will execute it’s own run() method.

We can also pass same or single target Runnable objects to multiple Threads so that all the thread instances perform the same work.

Besides the Thread constructors with no input variable and constructor with input as target Runnable there are also other constructors of thread class which takes different values. Below are some of them, which are commonly used.

* Thread()
* Thread(Runnable target)
* Thread(Runnable target, String name)
* Thread(String name)

As we understood, the last two constructors takes String objects which gives user defined names to the Threads created. The main thread by default has its name as ‘*main’.* If not a name is given Java automatically gives names to threads as Thread-serial-*integer-of-thread-creation*

*e.g.”* *Thread-1”;*

**Starting a Thread:**

So far things are pretty like other Java stuffs. But to create a thread we need a new call stack and as discussed earlier invoking the run() method does that. But suppose, we have Thread instance t which has its target r in it, i.e. the run() method is available.

We can simply invoke the run method by calling t.run() right, just like any other Java code. But it won’t do anything about creating and running a new thread. *It will surely execute the run command, but in the same thread, not in a new thread.* To make a new thread with its own call stack we need to execute the following command. We need to invoke start() method to the Thread object.

My Runnable r = new MyRunnable();

MyThread t = new MyThread(r);

t.start(); // 🡪 **Here happens the magic of new thread creation**

t.run(); 🡪 **Legal but, doesn’t create a new thread**

When we call start() on a Thread object, the Thread object creates a new thread and its own call stack by invoking the run() method. Prior to calling the start() method on the Thread object, it is just another simple Java object and not a new thread of execution. When the start() method is invoked on the thread object following stuffs happens.

* A new thread of execution started i.e. new call stack gets created.
* The thread moves from NEW to RUNNABLE state. (states of thread is discussed later)
* When the thread gets the chance to execute the target run() method it will run.

Below is a pictorial description of how thread and call stack works.

Figure 1: When the Java program initiates.



Figure 2: When start() method is called on a thread object



**Starting and running multiple Threads:**

*Before we delve into this topic we must remember, in Thread not everything is guaranteed.*

Whenever we start a Java program the main thread gets created. So far whenever we created a new thread two threads are already handled. But there can be multiple or any number of threads. Few things to point in in case of multiple threads.

* Behavior of multiple threads are not guaranteed if they have same priorities.
* Thread doesn’t follow the orders in which they are coded.
* There is no guarantee that after invoking start() method it will continue till its job is done.
* A single multi thread program can give different outputs when run multiple time in single machine or in different machines. However codes can be written with proper care to overcome this situation.
* Once the run() method is completed a thread is considered in DEAD state its call stack is dissolved.
* The Thread object might still remain when in DEAD state but the thread of execution doesn’t.
* Once a Thread has terminated start() cannot be called over it again. It till throw exception.
* Multiple start() call on a Thread object throws exception. A Thread object can only call start() once.
* Only a thread in NEW state can invoke the start() method.
* A thread which is already in RUNNABLE state or DEAD state cannot be re-started.
* Each thread is guaranteed to execute till it is completed.
* Within a single thread actions are guaranteed to perform as per the code. But in between different threads they can mix in unpredictable ways.

The uncontrollable behavior of threads are handled by Thread Scheduler and developers have no control on the scheduler. The Thread Scheduler itself gives no guarantee on the order of execution of thread nor does it give any promises to the duration a thread will run.

However one thing can be made *guaranteed.* We can invoke start() to a Thread object and tell it not to run until some other thread has finished i.e. the run() method has completed its job, using the join() method.

Let’s check the below code and see how a same code ran on same machine for multiple time gave different outputs.

Multithread code:

package com.multithreadbasic.main;  
  
public class MultiThread {  
  
 public static void main(String[] args) {  
 MultiThreadRunnable r = new MultiThreadRunnable();  
  
 Thread t1 = new Thread(r);  
 Thread t2 = new Thread(r);  
 Thread t3 = new Thread(r);  
  
 t1.setName("First Name!");  
 t2.setName("Second Name!");  
 t3.setName("Third Name!");  
  
 t1.start();  
 t2.start();  
 t3.start();  
 }

}  
package com.multithreadbasic.main;  
  
public class MultiThreadRunnable implements Runnable {  
 @Override  
 public void run() {  
 for(int i=0; i < 3; i++){  
 System.*out*.println("Run by thread: " + Thread.*currentThread*().getName()  
 + " value of i: " + i);  
 }  
 }

Outputs after running the same program in same machine for several times.

Ddddd

Ddd

D

**Thread states and transition:**

*Output 1:*

Run by thread: First Name! value of i: 0

Run by thread: Third Name! value of i: 0

Run by thread: Third Name! value of i: 1

Run by thread: Third Name! value of i: 2

Run by thread: Second Name! value of i: 0

Run by thread: First Name! value of i: 1

Run by thread: Second Name! value of i: 1

Run by thread: First Name! value of i: 2

Run by thread: Second Name! value of i: 2

*Output 2:*

Run by thread: First Name! value of i: 0

Run by thread: First Name! value of i: 1

Run by thread: First Name! value of i: 2

Run by thread: Second Name! value of i: 0

Run by thread: Second Name! value of i: 1

Run by thread: Second Name! value of i: 2

Run by thread: Third Name! value of i: 0

Run by thread: Third Name! value of i: 1

Run by thread: Third Name! value of i: 2

*Output 3:*

Run by thread: Second Name! value of i: 0

Run by thread: Second Name! value of i: 1

Run by thread: Second Name! value of i: 2

Run by thread: First Name! value of i: 0

Run by thread: First Name! value of i: 1

Run by thread: First Name! value of i: 2

Run by thread: Third Name! value of i: 0

Run by thread: Third Name! value of i: 2

A Thread object can have *one of the* 5 different states. They are as follows:

1. NEW
2. RUNNABLE
3. RUNNING
4. WAITING/BLOCKING/SLEEPING
5. DEAD/TERMINATED



States of thread

State NEW: When a new thread object is created it is said to be in NEW state, before it invokes the start() method. At this point the instance is live but the thread of execution is not considered to be *alive*. A thread once in the NEW state if gets changed to another state cannot return back to this state again.

State RUNNABLE: When the start() method is invoked on a Thread object the object moves to the RUNNABLE state. At this state the Thread object is eligible to run but yet not in the running state or the scheduler has not selected the thread to run. At this state he thread is considered to be *alive*.

A thread can come back to this state from the NEW state or from the other states like RUNNING or WAITING/BLOCKING/SLEEPING, but not once it has got terminated and moved to DEAD state it is not possible to return to this state.

The scheduler selects the thread to run from the thread pool if the thread is in this state.

State RUNNING: This is the state when the thread is actually performing its actions. This happens when the scheduler selects a thread from the thread pool and let it execute. A Thread at this stage can go back to either RUNNABLE state or WAITING/BLOCKING/SLEEPING state or move to DEAD state, if it has completed its task. But to come back to this state the Thread has to have to be in the RUNNABLE state. The scheduler decides to move a Thread to this state. Of course the state thread is *alive* at this stage.

State WAITING/BLOCKING/SLEEPING: At this stage the thread is neither eligible to run nor it is in RUNNABLE state. But it can return back to RUNNABLE state again. In this particular state we can see 3 options are clubbed together. But one thing is still common, the thread is still *alive.*

* WAITING: A code (*not another thread, which is not possible*) may tell the thread to wait. In this case another thread can sends a notification to this thread not to wait anymore and the current thread goes back to RUNNABLE state.

In Javadoc there is another stage in this state namely [TIMED\_WAITING](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#TIMED_WAITING) where a thread that is waiting for another thread to perform an action awaits for a specified waiting time in this state.

* BLOCKING: A thread may be fetching some resource and so it is blocked. For example it is waiting for some I/O operation to complete or writing some data to file or database. Once the resource is available it goes back to RUNNABLE state.
* SLEEPING: If a code tells the thread to sleep for some time it stays in this state. It is used in code to slow down a thread or force to give other threads some chances, since reasonable turn-taking is not guaranteed and handled by scheduler. Once the time period is over it goes back to RUNNABLE state.

State DEAD/TERMINATED: A thread is considered dead when its run() method gets over. The Thread object may stay in the heap, but the thread of execution is over. This Thread cannot be run again calling the start() function, as it will throw exception. At this stage the thread is of course not considered to be *alive.*

**Thread priority:**

Every thread has a priority and gets its turn based on it (mostly). The priority id defined by integer values generally 1 to 10 but in some cases it might be 1 to 5. In most of the cases the scheduler of the JVM uses time-slicing methods where each thread is given a fair amount of time. But that is not guaranteed for all JVMs. Some JVM may let the scheduler to let one thread to complete before executing another thread.

However, thread priorities are used in one important way in most of JVMs. If a thread has higher priority and is in thread pool i.e. in RUNNABLE state than any other thread or the currently running thread, the thread running is pushed back to RUNNABLE and the thread with highest priority is chosen to run by the scheduler. Usually the currently running thread has highest priority or at least equal than any of the priorities of the threads in the thread pool. But as JVM may vary this can only give some support but never guarantees it. Thread scheduling priority behavior is not guaranteed. It can increase the efficiency but do not promises.

If all the threads in the thread pool has equal priorities, the scheduler is free to choose any of them. In a time slicing JVM a thread once kicked back to RUNNABLE from RUNNING can again come back to execute as the priorities are same for all. It can also pick a thread and wait till it gets completed.

**Setting up thread priorities:**

A thread if not explicitly mentioned, gets the default priority of the JVM. JVM never changes thread priorities. But the range of priorities from one JVM may vary from other. Let’s have a look at the following codes.

public class MyThread{

public static void main(String[] args){

MyThread t = new MyThread();

}

}

public

**dddacdcadc dddddddd**

Thread t will now have priority 8

public class MyThread{

public static void main(String[] args){

MyThread t = new MyThread();

t.setPriority(8)

}

}

dd

Thread with same value as main thread

**Important methods/steps for prevention of thread execution:**

* sleep(): Guaranteed to prevent the thread from execution for the specified amount of time.
* join(): Guaranteed to stop the current thread from execution until the thread it joins with exits or completes its execution
* yield (): Not guaranteed to make an useful effect. But causes the current running thread to go back to RUNNABLE state from execution, so that other threads in the pool may have a chance to run.
* A call to wait() on an object. (Will be explained in Part 2.)
* A thread scheduler can decided to stop a currently running thread from execution back to the pool. It depends on the scheduler itself and its choice.
* When a thread completes its run () method.