# Java Concurrency / Multithreading

### What do mean by Multithreading?

Multithreading basically means a lightweight process. Threads are lightweight processes.

Threads are separate paths of execution which are functionally independent on each other.  
To make use the CPU time in better way.

### Multiprocessing and Multitasking

**Multiprocessing** means many user processes which are being processed by a single processor. For example- On UNIX System 10 user’s processes can be processed at a time by Unix Server.

**Multitasking** means single user many processes which are being processed by a single processor. For example – On UNIX System, a user can issue a single foreground command and many background commands to execute on Unix Server.

**Multithreading** means within single processes there are separate paths of execution. There is not more than one process here. For example- In word document while tying auto save, spell check are getting executed parallel.

Therefore, in case of **multitasking and multiprocessing each process has a process id**; it has its own memory space to work with. In case of multithreaded application, all threads share the same memory space.

### What is a Thread?

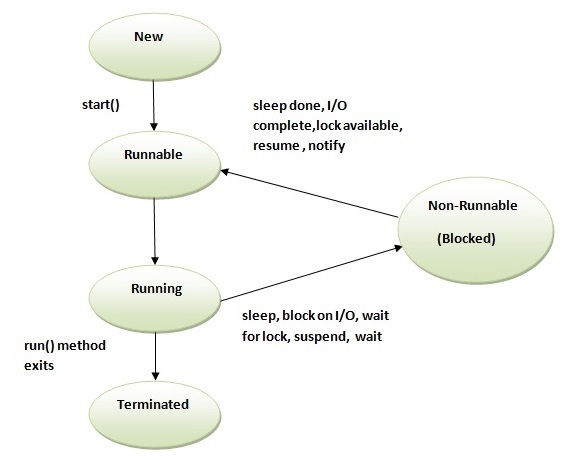
Thread is a sequence of code executed independently with other threads of control with   in a single executed program.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon.

### Life cycle of Thread

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



* **New** - The thread is in new state if you create an instance of Thread class but before the invocation of start() method
* **Runnable** - The thread is in runnable state after invocation of start () method, but the thread scheduler has not selected it to be the running thread.
* **Running** - The thread is in running state if the thread scheduler has selected it.
* **Non-Runnable (Blocked)** - This is the state when the thread is still alive, but is currently not eligible to run.
* **Terminated -** A thread is in terminated or dead state when its run () method exits.
* From *Runnable* to *Running*, we invoke a start() method which intern invoke run() {} method
* From *Running* to *Non-Runnable* state, there are 3 ways:  
  Using sleep(ms) method: After specified period thread automatically comes back to running state  
  Using suspend() method: If we call resume(), then only suspended thread comes back to running state  
  Using wait() method : If we call notify() / notifyAll(), then only suspended thread comes back to running state
* From *Running* state to *Terminated* state: Control come out of the run() method
* It is the run () method where entire execution of a thread is defined.
* Since JDK1.2 *stop* (), *suspend* () and *resume* () have been deprecated.

**Question-** Why is **Thread.stop() has been deprecated?**

This method is deprecated for the simple reason that it's unsafe and may lead the program to some unexpected circumstances. Thread.stop() when called, it causes the thread to release all the acquired monitors and throw a ThreadDeath error, which ultimately causes the thread to die. Since, the thread releases all the acquired monitors immediately, so it may leave few objects (whose monitors were acquired by the thread) in an inconsistent state. Such objects are called damaged objects and obviously they may result into some arbitrary behavior.

**Question-** Why are Thread.suspend and Thread.resume deprecated?

*Thread.suspend* is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling *resu me*, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes.

### Different ways to create a Thread

* Two ways for creating a thread  
  1) Extending from Thread class  
  2) Implementing Runnable Interface0
* Steps to create a Thread **by extending Thread clas**s  
  1) Extend Thread class0 into your class  
  2) Override run() method // Optional  
  3) Create an instance of your class (Will give you thread instance)  
  4) Start the thread
* Steps to create a Thread **by implementing Runnable Interface**  
  1) Implement Runnable interface into your class  
  2) Override run() method  
  3) Create an instance of your class (Will give you runnable instance)  
  4) Create a Thread object by passing Runnable object as parameter  
  5) Start the thread

**Question-** Which one the above way is preferred?

**Question-** Can I call run () method directly?

**Example of Runnable-** com.src.java.multithreading – DisplayBanner.java, TestBanner.java  
**Example of Thread Class-** com.src.java.multithreading – DisplayThread.java, TestThread.java

### Working with a Thread

* Following classes and interface: *Thread; ThreadGroup; Runnable*; All these resides in the lang package.
* Every thread have their own properties-  
  *Every thread has a name-* getName () / setName ()  
  Every thread has a priority- getPriority () / setPriority(): 1-10  
  *Every thread belongs to a Group-* getThreadGroup() : default is main  
  For example - main thread: main / 5/ main
* By default thread name is assigned by thread API: Thread0, Thread1, and Thread2…  
  By default thread group name always based on the name of the parent group.  
  Normal priority of any thread is 5. 1 is minimum and 10 is maximum. If we provide priority out this range, then will get an **illegalArgumentException**.
* Thread.Sleep () Method-  
  Sleep method in Thread tells the currently executing thread to sleep for specified amount of time.  
  Sleep method accepts the time in MilliSeconds. However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS.  
  It can throw interruptedException. So it should be embedded in the try catch block.   
  Once the thread completes or out of its sleep state, it can move to Running or Runnable state.  
  **Example -** com.src.java.multithreading – SleepMessages.java
* threadObject.Join () throws **InterruptedException**: **It is used to impose order on execution of multiple Threads**. The join () method waits for a thread to die.   
  In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.

**Question-** You have three threads T1, T2 and T3, How do you ensure that they finish in order T1, T2, T3 ?

**Example -** com.src.java.multithreading – ThreadJoinExample.java, MyRunnable.java, JoinExample.java

* Threads work in 2 ways-  
  1) *pre-emptive multithreading where all threads get chance to execute*   
  2) *co-operative multithreading which is also called as selfish thread where highest get chance to execute first. If we want to make it unselfish, we can use* ***yield ()*** *method provided that all threads having same priority. Yield will gives up execution of current thread and will go in running state. So that next thread comes and running*.
* isAlive()
* ThreadGroup.setMaxPriority()  
  ThreadGroup tg1 = new ThreadGroup();  
  t1, t2, and t3 belongs to tg1  
  t1.setPriority(9);  
  tg1.setMaxPriority(8);  
  t2.setPriority(9);  
  t3.setPriority(6);  
  S.O.P(t1.getPriority()); // 9  
  S.O.P(t2.getPriority()); // 8  
  S.O.P(t3.getPriority()); // 6  
  **t1.setPriority(10);**  
  **S.O.P(t1.getPriority()); // 8**
* There are 2 types of Thread-  
  User thread and Daemon Thread (System thread running in background like Garbage collector which is having low priority).  
  User Thread- When we create a thread object and call start () method then it is a User Thread  
  Daemon Thread- When we create a thread object and before calling start () method if we call setDaemon(true) and then call start() method then it is a Daemon thread.  
  *Once we start a thread we can’t change the type of a thread.*  
  Any thread which is a Daemon thread, within which we are creating a child thread, then children will also be a Daemon thread.  
  IsDaemon () method return the type of thread.  
  **JVM run till all the user threads are alive. Once all users thread ends then JVM has responsibility to kill all Daemon thread and get out of the execution from your program**. That’s why we can’t define same task inside a user thread instead of a Daemon thread.  
  **User can’t kill a Daemon thread otherwise it will give an exception.**
* What's the difference between the methods sleep () and wait ()?

Sleep ():

It is a static method on Thread class. It makes the current thread into the "Not Runnable" state for specified  
 amount of time. During this time, the thread keeps the lock (monitors) it has acquired.

Wait ():

It is a method on Object class. It makes the current thread into the "Not Runnable" state. Wait is called on an   
 object, not a thread. Before calling wait () method, the object should be synchronized, means the object should  
 be inside synchronized block.

The call to wait () releases the acquired lock.

**Question-** I have method run () in a Class SimpleSchedular. I want to create a schedular so those run () get called in every 5mins interval.

**com.src.java.multithreading –** SimpleSchedular.java

### Locks / Thread synchronization

* Synchronization is built around an internal entity known as the **intrinsic lock or monitor lock**. Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them. A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.
* When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.
* You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the **intrinsic lock for the Class object associated with the class**. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.
* The *synchronized* keyword in Java ensures that only a single thread can execute a block of code at the same time.
* Either synchronized a method OR synchronize a block using Synchronized modifier.  
  Synchronized public void m() {  
  }  
   OR  
  public void m() {  
   Synchronized(<object>) {  
   }  
  }

**Example -** com.src.java.multithreading.lock – PrintDemo.java, ThreadDemo.java, SyncrinizedThreadExample.java, SynchronizedThreadDemo.java

* **Scenario -** Let’s suppose we have an online banking system, where people can log in and access their account information. Whenever someone logs in to their account online, they receive a separate and unique thread so that different bank account holders can access the central system simultaneously.

Let’s say that there’s a husband and wife **- Jack and Jill** - who share a joint account. They currently have 3,000 in their account. They both log in to their online bank account at the same time, but from different locations.

They both decide to deposit 1000 each into their account through a wire transfer from other bank accounts that they have at the same time. So, the t otal account balance after these 2 deposits should be 3,000 + (1000 \* 2), which equals 5,000.

Now, the processor is running the thread for Jack, who is also depositing 1000 into their account. When Jack’s thread deposits 1000, the account balance is still only $1,000, because the variable **accountBalance** has not yet been updated in Jill’s thread. Remember that Jill’s thread stopped execution right before the accountBalance variable was updated.

So, Jack’s thread runs until it completes the deposit function, and then updates the value of the accountBalance variable to 4000. After this, control returns to Jill’s thread, where newAccountBalance has the value of 4000. Then, it just assigns this value of 4000 to accountBalance and returns. And that is the end of execution.

What is the result of these 2 deposits of 1000? Well, the accountBalance variable ends up being set to only 4000, when it should have been 5000. This means Jack and Jill lost 1000. This is good for the bank, but a huge problem for Jack and Jill, and any other of the bank's customers.

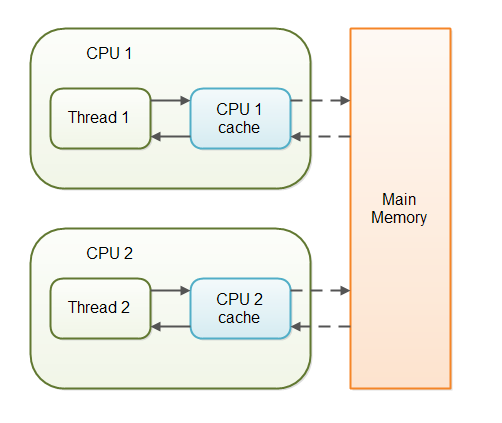
**Example -** com.src.java.multithreading.lock – BankAccount.java, DepositMoney.java, WithdrawMoney.java, AccountTesting.java

* Few important points-
  1. Synchronized keyword can be used only with methods and code blocks. These methods or blocks can be static or non-static both.
  2. When ever a thread enters into java synchronized method or block it acquires a lock and whenever it leaves java synchronized method or block it releases the lock. Lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.
  3. Java synchronized keyword is re-entrant in nature it means if a java synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.
  4. Java Thread acquires an object level lock when it enters into an instance synchronized java method and acquires a class level lock when it enters into static synchronized java method.
  5. Java Synchronization will throw NullPointerException if object used in java synchronized block is null.
  6. One Major disadvantage of Java synchronized keyword is that it doesn't allow concurrent read, which can potentially limit scalability. By using concept of lock stripping and using different locks for reading and writing, you can overcome this limitation of synchronized in Java. You will be glad to know that java.util.concurrent.locks.ReentrantReadWriteLock provides ready made implementation of ReadWriteLock in Java.
  7. One more limitation of java synchronized keyword is that it can only be used to control access of shared object within the same JVM. If you have more than one JVM and need to synchronized access to a shared file system or database, the Java synchronized keyword is not at all sufficient. You need to implement a kind of global lock for that.
  8. Java synchronized block is better than java synchronized method in Java because by using synchronized block you can only lock critical section of code and avoid locking whole method which can possibly degrade performance.

### Volatile variables

The Java **volatile** keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means that each thread may copy the variables into the CPU cache of different CPUs.



With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory.

**Scenario-**

public class SharedObject { public int counter = 0;}

Thread 1 could read a shared counter variable with the value 0 into its CPU cache, increment it to 1 and not write the changed value back into main memory. Thread 2 could then read the same counter variable from main memory where the value of the variable is still 0, into its own CPU cache. Thread 2 could then also increment the counter to 1, and also not write it back to main memory. Thread 1 and Thread 2 are now practically out of sync. The real value of the shared counter variable should have been 2, but each of the threads has the value 1 for the variable in their CPU caches, and in main memory the value is still 0. It is a mess! Even if the threads eventually write their value for the shared counter variable back to main memory, the value will be wrong.

By declaring the shared counter *variable* volatile the JVM guarantees that every read of the variable will always be read from main memory, and that all writes to the variable will always be written back to main memory. If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B.

public class SharedObject { public volatile int counter = 0;}

Reading and writing of volatile variables causes the variable to be read or written to main memory. Reading from and writing to main memory is more expensive than accessing the CPU cache.

### Wait – Notify - NotifyAll

* Dead Lock - Dining Philospher problem



/\*\*The dining philosophers problem is summarized as five silent philosophers

\* sitting at a circular table doing one of two things: **eating** or **thinking**.

\* While eating, they are not thinking, and while thinking, they are not eating.

\* A large bowl of Rice is placed in the center, which requires two forks

\* to serve and to eat. A fork is placed in between

\* each pair of adjacent philosophers, and each philosopher may only use the

\* fork to his left and the fork to his right. However, the philosophers do not

\* speak to each other. Deadlock would arise if every philosopher held a left

\* fork and waited perpetually for a right fork (or vice versa). Originally used

\* as a means of illustrating the problem of deadlock, this system reaches

\* deadlock when there is a 'cycle of unwarranted requests'. In this case

\* philosopher P1 waits for the fork grabbed by philosopher P2 who is waiting

\* for the fork of philosopher P3 and so forth, making a circular chain.

\*/  
**Example -** com.src.java.multithreading.lock – Fork.java, Philospher.java, Dining.java

* To resolve Producer-consumer issue we can use Wait / Notify / NotifyAll

**Example -** com.src.java.multithreading.lock – Stock.java, Producer.java, Consumer.java, ProducerConsumerTestClient.java

### Inter – Thread Communication

In multithreading, a thread has a local data which is required to be passed to another thread and based upon data another thread get processed it. This approach is called as Inter-thread communication. An inter-thread communication always achieved via pipes. This is the place where we use PipedInputStream / PipedOutputStream class or PipedReader / PipedWriter class.

**Scenario** - There are two threads: Number Generator thread always generating a random number and passed through pipes which will be used by Average Generator thread to calculate the average.

**Example -** com.src.java.multithreading.lock – PipeTest.java, NumGen.java, RunningAvg.java

### Executors, Schedule Executor & Thread Pool

Thread pools manage a pool of worker threads. The thread pool contains a work queue which holds tasks waiting to get executed.

A thread pool can be described as a collection of Runnable objects (work queue) and a connection of running threads. These threads are constantly running and are checking the work query for new work. If there is new work to be done they execute this Runnable. The Thread class itself provides a method, e.g. execute (Runnable r) to add a new Runnable object to the work queue.

The Executor framework provides example implementation of the **java.util.concurrent.Executor** interface, e.g. Executors.newFixedThreadPool (int n) which will create n worker threads. The ExecutorService adds life cycle methods to the Executor, which allows to shutdown the Executor and to wait for termination.

ScheduledThreadExecutor provides multiple working threads by using Thread pool. This class extends ThreadPoolExecutor class and uses thread pool to have a pool of threads to execute multiple tasks in parallel.

**Example -** com.src.java.multithreading.lock – MyRunnable.java, ExecutorServiceExample.java, ScheduleExecutorServiceExample.java

### Futures and Callables

The executor framework presented above works with Runnables. Runnable do not return result. In case you expect your threads to return a computed result you can use java.util.concurrent.Callable. The Callable object allows returning values after completion.

The **Callable** interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.

The Callable object uses generics to define the type of object which is returned.

If you submit a Callable object to an Executor the framework returns an object of type java.util.concurrent.Future. This Future object can be used to check the status of a Callable and to retrieve the result from the Callable.

On the Executor you can use the method submit to submit a Callable and to get a future. To retrieve the result of the future we can use the get () method.

A **Future** represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed

**Example -** com.src.java.multithreading.lock – MyCallable.java, CallableFutures.java

### Nonblocking algorithms

Java 5.0 provides supports for additional atomic operations. This allows to develop algorithm which are non-blocking algorithm, e.g. which do not require synchronization, but are based on low-level atomic hardware primitives such as compare-and-swap (CAS). A compare-and-swap operation checks if the variable has a certain value and if it has this value it will perform this operation. This operation takes three parameters, the memory address, the expected current value and the new value. It atomically update the value at the given memory address if the current value is the expected.

Non-blocking algorithm are usually much faster then blocking algorithms as the synchronization of threads appears on a much finer level (hardware).

**Example -** com.src.java.multithreading.lock – Counter.java, UniqueCounter.java, AtomicTest.java

### ****ReentrantLock****

This is the most widely used implementation class of Lock interface. This class implements the Lock interface in similar way as synchronized keyword. Apart from Lock interface implementation, ReentrantLock contains some utility methods to get the thread holding the lock, threads waiting to acquire the lock etc.

**Example -** com.src.java.multithreading.lock – Resource.java, ConcurrencyLockExample.java, ConcurrencyLockClient.java

### ****Lock vs synchronized****

1. Lock provides more visibility and options for locking, unlike synchronized where a thread might end up waiting indefinitely for the lock, we can use tryLock () to make sure thread waits for specific time only.
2. Synchronization code is much cleaner and easy to maintain whereas with Lock we are forced to have try-finally block to make sure Lock is released even if some exception is thrown between lock() and unlock() method calls.
3. Synchronization blocks or methods can cover only one method whereas we can acquire the lock in one method and release it in another method with Lock API.