**1. What is MultiTasking, MultiThreading:**

**MultiTasking**: Normally, we humans do several tasks concurrently or simultaneously, like watching a movie and texting to someone , like writing an article and surfing web browser etc etc.

**Multi-tasking is the ability of an operating system to run multiple processes or tasks concurrently.** **In multi-tasking, the operating system divides the CPU time between multiple tasks, allowing them to execute simultaneously**

**MultiThreading:**

Sometimes, we may observe that we are using MS Word and also, we are using MS Paint alongside, so this is not MultiThreading , this is actually Multi-Tasking where multiple apps or multiple processes are running concurrently.

Multi-Threading is where we are using a MS-Word and while writing something in the MS-Word, at the same time, we are getting a type/error which we can correct. So, along with writing, an auto-correct function is also getting executing simultaneously.

**Multi-threading is a technique in which an operating system divides a single process into multiple threads, each of which can be executed concurrently**.

So, in MS-Word within MS Word, we are writing and concurrently, auto-correct function is also getting executed.

Threads share the same memory space and resources of the parent process, allowing them to communicate and synchronize data easily.

**Multi-threading is useful for improving application performance by allowing different parts of the same application to execute simultaneously**.

**2. Process, Thread, Diff between them**

**Process:**

* A process is an instance of a program running on a computer.
* Each process has its own memory space, containing the program code, data, and resources allocated to it.
* Processes are independent of each other and isolated. They cannot directly access the memory or resources of other processes.

.

**Threads:**

* A thread is the smallest unit of execution within a process.
* A thread is a single sequential flow of control within a program
* At runtime, threads in a program share the same memory space and can therefore, share both data and code(i.e. they are lightweight compared to processes)
* Threads within the same process can communicate directly through shared memory, making communication faster

A screen shot of a computer

Description automatically generated

**Main Thread:**

* When a single-threaded application runs, a main thread is automatically created.
* In a standalone application, a user thread is automatically created to execute the main method. This thread is referred to as the Main Thread.
* The main thread is responsible for initializing various components of the Java application, including static variables, class loaders, and other resources.
* If no other user threads are created, the program terminates when the main method finishes its execution.
* However, if there are custom or user-defined threads running after the main method's execution completes, the program will continue running. These additional threads keep the program alive.

**Daemon Threads:-**

Daemon threads in Java are threads that run in the background and provide services to other threads or perform tasks that do not require user interaction

**There is an important difference between user-threads and daemon threads:-**

1. JVM shuts down i.e. the program terminates when execution of user-threads including the main thread gets completed, even if the Daemon Thread is still running in the backgroud.
2. If our main thread or other user-threads goes in dead state, then there is no use of Daemon thread.
3. Creating Daemon Thread using setDaemon(boolean b)

Converts simple thread to Daemon thread if set to true

1. Using isDaemon() method, we can check if a thread

is a daemon thread.

1. Daemon threads are commonly used for tasks such

as automatic resource cleanup,

periodic maintenance, or background monitoring

Can you change a thread's status from daemon to non-daemon or vice versa after it has been started?

The status of a thread as daemon or non-daemon is determined at the time of thread creation and cannot be changed later.

Thread myThread = **new** Thread(() -> {

// Thread logic goes here

});

So, we can set the user thread as Daemon thread before the use of start() method in this way

// Set the thread as daemon

myThread.setDaemon(**true**);

// Start the thread

myThread.start()

Once the thread has been started using the **start()** method, its daemon status is fixed and cannot be altered during its lifetime.

**LifeCycle of a Thread:-** **New -> Runnable -> Running -> [Blocked/Waiting/Timed Waiting] -> Running -> Terminated**

t.interrupt()

New -> Runnable -> Running -> Dead

Create an Object

t.start()

Blocked/Waiting

wait(),sleep(),join()

1. **New**:
   * When a thread is created using the **new** keyword or by instantiating the **Thread** class, it is in the New state.
   * In this state, the thread has been created but has not yet started its execution.
2. **Runnable**:
   * After calling the **start()** method on the thread object, it transitions to the Runnable state.
   * In this state, the thread is ready to run, but the scheduler has not yet selected it to be executed.
3. **Running**:
   * When the scheduler selects the thread for execution, it enters the Running state.
   * In this state, the thread's code is actively being executed by the CPU.
4. **Blocked/Waiting**:
   * A thread can transition to the Blocked or Waiting state for various reasons, such as waiting for I/O operations, synchronization locks, or other threads.
   * In the Blocked state, the thread is temporarily unable to proceed with its execution and must wait for a specific condition to be met.
   * In the Waiting state, the thread is waiting indefinitely until it receives a notification or interrupt to resume its execution.
5. **Timed Waiting**:
   * Similar to the Waiting state, a thread can also enter the Timed Waiting state by invoking methods that specify a timeout period, such as **Thread.sleep()** or **Object.wait(timeout)**.
   * In this state, the thread waits for a certain amount of time before either resuming its execution or being interrupted.
6. **Terminated**:
   * The thread transitions to the Terminated state when its **run()** method completes or when an unhandled exception occurs.
   * In this state, the thread has finished its execution and can no longer be restarted.
   * Once terminated, the thread cannot transition back to any other state.
   * interrupt() method call on a thread also terminates a thread explicitly

**Creating the User-Defined Threads**:-

There are basically 2 ways to create an user-defined thread:-

1. By extending a Thread Class
2. By implementing a Runnable Interface

Thread Creating By Extending a Thread Class:-

**public** **class** Thread1 **extends** Thread{

//Assigning name to my custom thread

**public** Thread1(String threadName) {

**super**(threadName);

}

@Override

**public** **void** run() {

**for**(**int** i=0;i<5;i++) {

//Thread.currentThread returns a reference to the currently executing thread

//including it's name, thread-priority,parent-thread

System.***out***.println("inside thread "+Thread.*currentThread*() + " "+i);

}

}

}

Thread.currentThread().getName() gives only Thread Name

**public** **class** MainClass {

**public** **static** **void** main(String[] args) {

System.***out***.println("Main thread Starting");

Thread t1 = **new** Thread1("Thread 0");

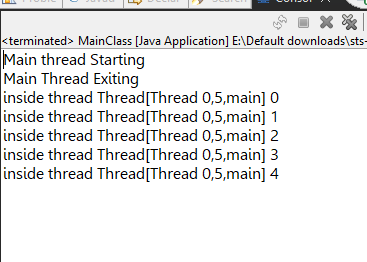
t1.start();

System.***out***.println("Main Thread Exiting");

}

It totally depends on the JVM in which order we will get the output for this program.

}



Parent Thraed

Thread Priority

Thread Name

Thread By Implementing a Runnable Interface:-

We try to implement a Runnable Interface in our custom thread class and override the run method of the Runnable Interface.

**NOTE:**

Even the actual java **Thread** class also implements Runnable Interface

**public** **class** Thread2 **implements** Runnable{

@Override

**public** **void** run() {

**for**(**int** i=0;i<5;i++) {

System.***out***.println("Thread running "+Thread.*currentThread*()+" "+i);

}

}

}

**public** **class** MainClass {

Giving name to the user-defined Thread class and it is optional

**public** **static** **void** main(String[] args) {

System.***out***.println("Main thread Starting");

System.***out***.println("Main Thread Exiting");

Thread t2 = **new** Thread(**new** Thread2(),"Thread 1");

t2.start();

}

}

While creating a Thread by extending a Thread class, we can directly instantiate our custom Thread class(Thread1) as it is a direct sub-class of the Thread class.

But while creating a Thread class by implementing the Runnable interface, we need to pass the Runnable object(or our Thread 2 class instance) as argument to the constructor of the Thread class as below:

Thread t2 = **new** Thread(**new** Thread2(),"Thread 1");

**Which is better for creating threads - Extending a Thread Class or implementing a Runnable Interface:**

As we know, Java does not support Multiple Inheritance, but through interfaces, we can achieve multiple inheritance as we can implement more than one interface in Java.

So, it provides more flexibility in code structure.

Also, Java 8 intoduced Lambda Expressions which are more easier to use with Runnable Interface and also, it is more concise in use.

**Generally, we use Lambda expression while creating a thread as follows:-**

//Using Lambda Expression instead of creating another class

Thread t3 = **new** Thread(()->{

**for**(**int** i=0;i<5;i++) {

System.***out***.println("Thread Working "+Thread.*currentThread*()+" "+i);

}

},"Thread 2");

Instead of creating another class which implements the Runnable Interface and there, we write our logic inside the run method.

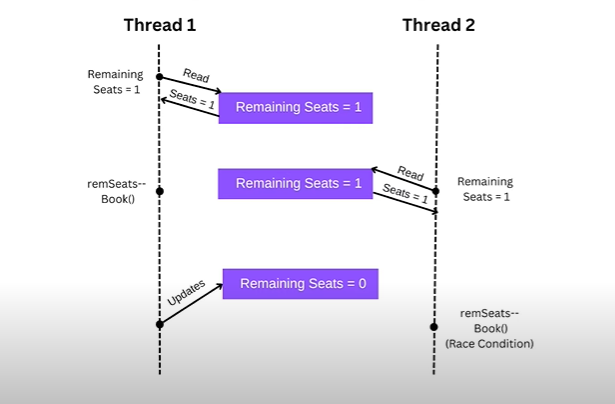
We can directly use the Lambda Expression as above which will make our overall code more concise

**Synchronization in Java Multi-Threading:-**

* It is the process by which we control the accessability of multiple threads to a particular shared resource.
* As threads share the same memory space i.e. they can share resources.
* But sometimes there is some critical situations where it is desirable that we can use only one thread at a time and only one thread has access to a shared resource.

Suppose, take an example of a MovieBooking Application, there is only 1 seat left,

Now I have booked the seat when say, only 1 seat was left . So, while one thread was operating my booking update which may take some time as the whole operation is heavy, there might be another thread for another user which comes and still see, the remaining seat is 1 as the substraction operation is still going at the backend and yet not subtracted. So, this thread will also try to access the resource at the same time which will lead to **data inconsistency**



**Race Condition Problem:-**

A race condition in Java occurs when two or more threads access shared resources or variables at the same time, and the outcome of the program depends on the timing or order of their execution i.e. the output becomes unpredictable or inconsistent.

This can lead to unpredictable results or errors because the threads may interfere with each other's operations, causing unexpected behavior.

**Solving the Problem:-**

To solve the issue, we basically use the keyword **synchronized.**

We can have both **synchronized blocks and methods.**

**Example Code:**

**When a thread enters a synchronized block or method, it must acquire the lock associated with the object specified in the synchronized keyword**

**package** com.multithreading.practice;

**public** **class** SynchronizationPractice {

**private** **int** counter = 0;

**private** **int** counter = 0;

Using synchronized block

**private** **final** Object lock = **new** Object();

**public** **void** increment() {

**synchronized**(lock) {

counter++;

System.***out***.println(Thread.*currentThread*().getName()+" incrementCount:"+ counter);

**try** {

Thread.*sleep*(1000);

}**catch**(InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **static** **void** main(String[] args) {

SynchronizationPractice s = **new** SynchronizationPractice();

Thread t1 = **new** Thread(()->{

**for**(**int** i=0;i<5;i++) {

s.increment();//this is like we are implementing this part inside the run() method

}

},"Thread-1");

Thread t2 = **new** Thread(()->{

**for**(**int** i=0;i<5;i++) {

s.increment();//this is like we are implementing this part inside the run() method

}

Thread-1 incrementCount:1

Thread-1 incrementCount:2

Thread-1 incrementCount:3

Thread-1 incrementCount:4

Thread-1 incrementCount:5

Thread-2 incrementCount:6

Thread-2 incrementCount:7

Thread-2 incrementCount:8

Thread-2 incrementCount:9

Thread-2 incrementCount:10

},"Thread-2");

t1.start();

t2.start();

}

}

**When the synchronized block or method exits (either by reaching the end of the block or by encountering a return statement), the lock is released.** This allows other threads to acquire the lock and enter the synchronized block or method.

For using synchronized method above, we just need to change the increment method as:-

**public** **class** SynchronizationPractice {

Internally it executes everything inside

**synchronized block as**

**synchronized(this){ … }**

**private** **int** counter = 0;

//private final Object lock = new Object();

**public** **synchronized** **void** increment() {

counter++;

System.***out***.println(Thread.*currentThread*().getName()+" incrementCount:"+ counter);

**try** {

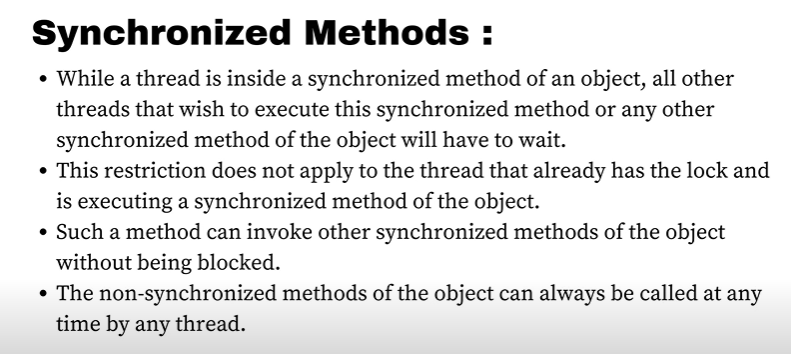
Thread.*sleep*(1000);

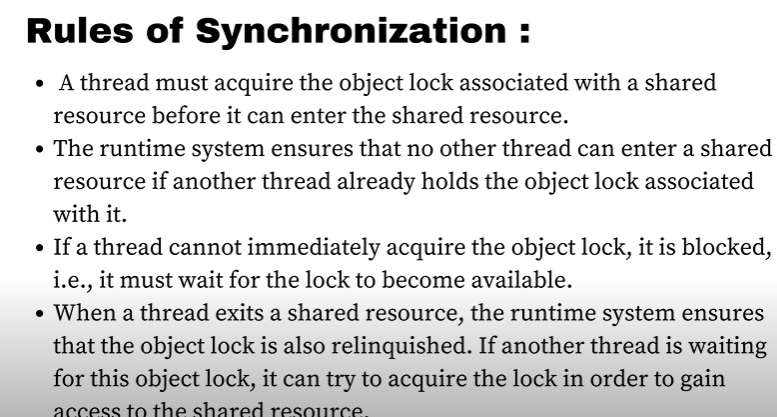
}**catch**(InterruptedException e) {

e.printStackTrace();

}

}





**How do we synchronize static methods as the above method was a non-static method:**

Because in a static method, we know we can’t have any instantiation of the class involved. In that case, so we synchronize on the class object itself**.**

We can **do** **this** by using the .**class** syntax, which represents the **class** object

**Example:-**

**public** **class** SynchronizationPractice {

**private** **static** **int** *count* = 0; // Shared static variable

**public** **static** **void** increment() {

**synchronized** (SynchronizationPractice.**class**) {

*count*++;

System.***out***.println(Thread.*currentThread*().getName() + " increments count to " + *count*);

**try** {

Thread.*sleep*(1000); // Simulate some processing time of 1sec

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **static** **void** main(String[] args) {

// Create multiple threads accessing the synchronized method

Thread thread1 = **new** Thread(() -> {

**for** (**int** i = 0; i < 5; i++) {

SynchronizationPractice.*increment*();

}

}, "Thread-1");

Thread thread2 = **new** Thread(() -> {

**for** (**int** i = 0; i < 5; i++) {

SynchronizationPractice.*increment*();

}

}, "Thread-2");

thread1.start();

thread2.start();

}

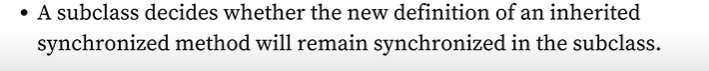
}

**If we use here the synchronize keyword in the method itself instead of using synchronize block, it automatically locks the** SynchronizationPractice.**class object internally.**

**Note:**

Static Synchronized method and Non-Static Synchronized method are independent of each other in terms of execution which means there could be 2 threads where one can have access to the static method and another thread having access lock of the non-static method

When a thread accesses a static synchronized method, it acquires the lock associated with the class object (**SynchronizationExample.class** in our example). On the other hand, when a thread accesses a non-static synchronized method, it acquires the lock associated with the object instance (**this**).Both the locks are different

****

**Thread Safety:-**

Thread safety in Java refers to the ability of a piece of code or a data structure to function correctly and reliably in a multi-threaded environment, where multiple threads may access and modify shared data concurrently

Thread Safety can be achieved by the following ways:-

1. **Synchronization**
2. Use Thread-Safe Classes such as **ConcurrentHashMap, AtomicInteger, CopyOnWriteArray**
3. **Volatile** keyword
4. **Locks**
5. **Atomic Operations.**

**Volatile keyword:-**

Suppose, we have 2 threads, **Thread T-0** & **Thread T-1.** Threads interacts with the CPU and in turn, our CPU interacts with the RAM. Now, normally it is efficient for our CPU usage optimization that our CPU access data from the cache then from the RAM or main memory.

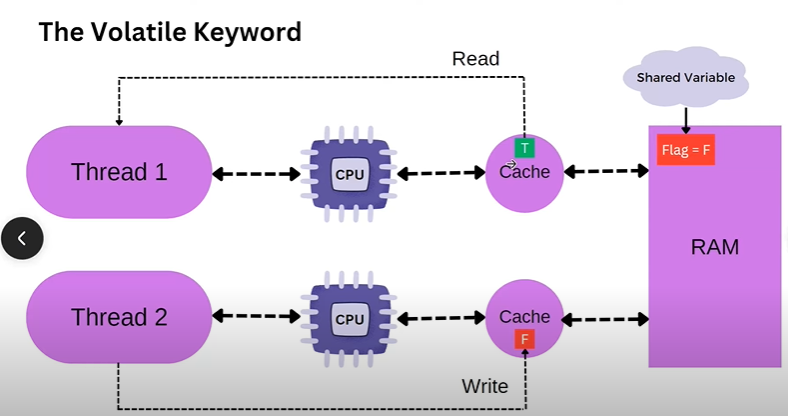
Now, if **Thread 2** updates the value of the shared variable to false in it’s local cache,

But at the same time **Thread 1** still has the value as true in it’s local cache.

And also, it takes some time to update the value in **RAM** as false of the shared variable.

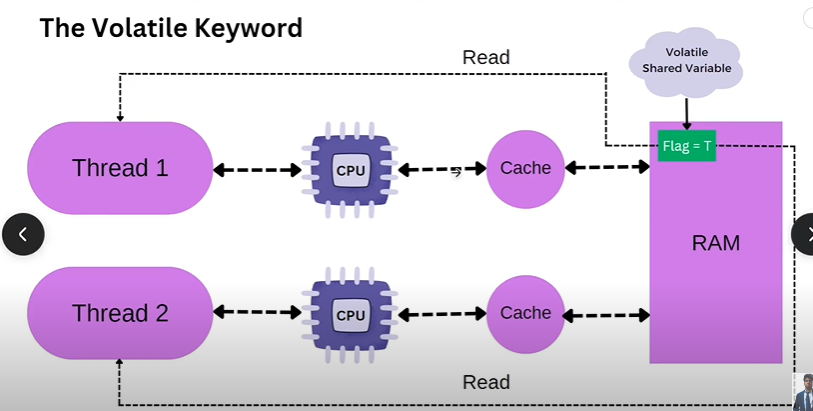
Not Updated Here

But in case of **Thread 1,** there is literally no visibility to get the updated shared variable value.

****

So, to solve the issue, we use the **volatile** keyword.

* Without the **volatile** keyword, changes made to a variable by one thread may not be visible to other threads immediately or at all.
* This is due to optimizations performed by the Java Virtual Machine (JVM), such as caching variables in thread-local memory.
* When a variable is declared as **volatile**, the JVM ensures that changes to that variable are immediately visible to all threads.
* When a shared variable is volatile, the threads reads it directly from the main memory rather than reading it from the local Cache memory
* Thread 2 in this case writes the updated value directly in the main memory as the variable which is updated is a volatile variable



**This is also important because in case of a Singleton Design Pattern principal,** if we are trying to implement it in a multi-threaded environment. It will not work because of the same reason as 2 threads may create 2 instances of the class , if we don’t use **volatile** keyword.

**Producer-Consumer Pattern Problem:-**

**package** com.multithreading.practice;

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** BlockingQueue {

**private** Queue<Integer> q;

**private** **int** capacity;

**private** BlockingQueue(**int** cap) {

q = **new** LinkedList<>();

capacity = cap;

}

**public** **boolean** add(**int** e) {

**synchronized**(q) {

**if**(q.size()==capacity) {

//do some work

System.***out***.println("Queue is full. Waiting for consumer to consume");

}

q.add(e);

System.***out***.println(Thread.*currentThread*()+" is working");

System.***out***.println("Produced:"+e);

**return** **true**;

}

}

**public** **int** remove() {

**synchronized**(q){

**if**(q.size() == 0) {

//do some work

System.***out***.println("Queue is empty. Waiting for producer to produce"+Thread.*currentThread*());

}

**int** element = q.poll();

System.***out***.println(Thread.*currentThread*()+" is working");

System.***out***.println("Consumed:"+element);

**return** element;

}

}

**public** **static** **void** main(String[] args) {

BlockingQueue queue = **new** BlockingQueue(5);

Thread producer = **new** Thread(() -> {

**for**(**int** i=0;i<5;i++) {

**try** {

queue.add(i);

}**catch**(Exception e) {

System.***out***.println("Interrupted Exception:"+e);

}

}

},"Thread-Producer");

Thread consumer = **new** Thread(() -> {

**for**(**int** j=0;j<5;j++) {

**try** {

queue.remove();

}**catch**(Exception e) {

System.***out***.println("Exception:"+e);

}

}

},"Thread-Consumer");

producer.start();

consumer.start();

}

}

**Here, the problem is it totally depends on the JVM which thread will start working first. So, suppose our Thread-Consumer starts working and have the lock, so no other Thread can work currently. So, Thread-Consumer calls the remove method and tries to remove an element when there is no element currently in the Queue. And also, no other Thread can add element to the Queue as Thread-Consumer has the lock.It will lead to an exception.**

**So, to solve this issue we change the code in the following way:**

**public** **boolean** add(**int** e) {

**synchronized**(q) {

**if**(q.size()==capacity) {

System.***out***.println("Queue is full. Waiting for c onsumer to consume");

**try** {

q.wait();

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

}

q.add(e);

System.***out***.println(Thread.*currentThread*()+" is working");

System.***out***.println("Produced:"+e);

q.notifyAll();

**return** **true**;

}

Adding these 2 methods solves the earlier issue

}

**public** **int** remove() {

**synchronized**(q){

**if**(q.size() == 0) {

System.***out***.println("Queue is empty. Waiting for producer to produce"+Thread.*currentThread*());

**try** {

q.wait();

} **catch** (InterruptedException e) { e.printStackTrace();

}

}

**int** element = q.poll();

System.***out***.println(Thread.*currentThread*()+" is working");

System.***out***.println("Consumed:"+element);

q.notifyAll();

**return** element;

}

}

We are adding here the wait method so that if any thread comes and finds that the queue is already empty we are sending it to the waiting state so that it releases the lock and other thread can add elements in the Queue. And after adding elements the other thread notifies using the **notifyAll()** method to notify all the waiting threads to start working again. So, it will resume it’s working after the line from where the wait() method on the thread was called and will remove the elements

**Now, the problem with the above code is Suppose, there are 2 adder threads working and the second thread after seeing the queue size == capacity,also goes to the waiting state along with the first adder thread.**

**Now suppose, the remover thread gets the lock and removes 1 item from the queue and sends the notifyAll() method. So, the adder1 thread comes and starts from the line where we called the wait() method and adds an element.**

**Now, the capacity is full for the queue. But here, the adder2 also will get notified and will start from the wait() method and will then without checking if the capacity is full, will try to add the element.This will lead to an exception. So, we want the adder2 thread to check if the capacity is full, if full, go to the waiting state.**

**For that we just need to change the if to while loop. This will solve our issue.**

This was earlier the if-block

**public** **int** remove() {

**synchronized**(q){

**while**(q.size() == 0) {

System.***out***.println("Queue is empty. Waiting for producer to produce"+Thread.*currentThread*());

**try** {

q.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**int** element = q.poll();

System.***out***.println(Thread.*currentThread*()+" is working");

System.***out***.println("Consumed:"+element);

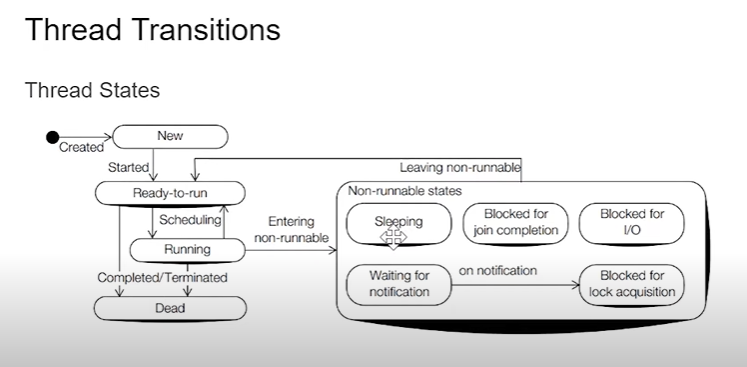
q.notifyAll();

**return** element;

}

}

**Note**: Every object has a waitSet where all the waiting threads are being put by the JVM.



Thread.**sleep**(1);

**Note:** Thread doesn’t release it’s lock on using the sleep() method as it does on using the wait() method.

Object.wait()

