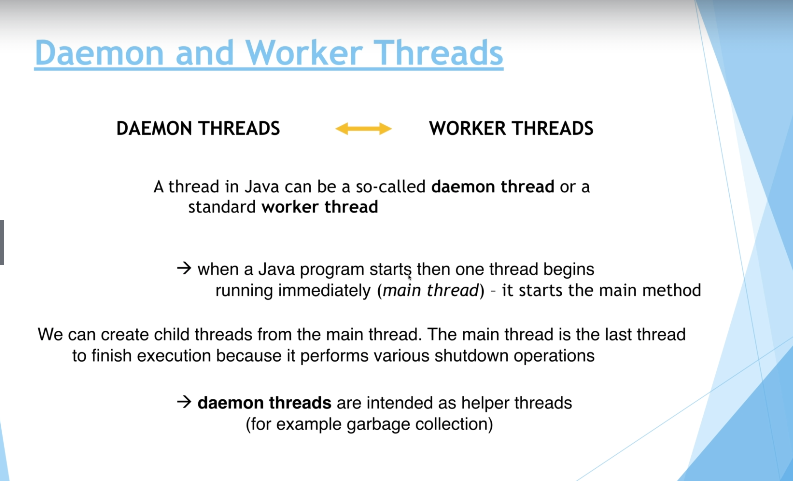
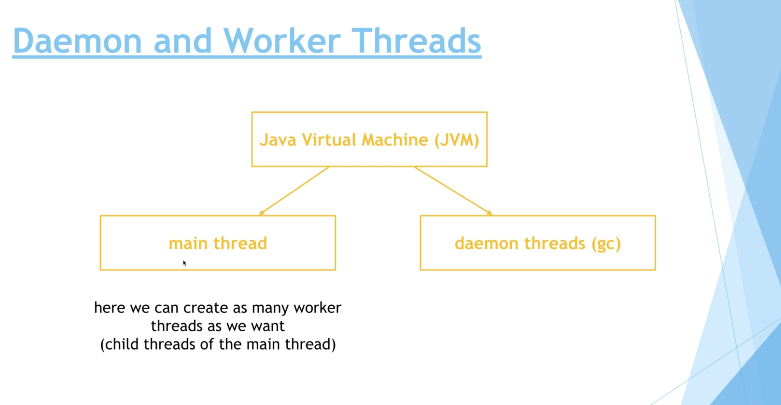
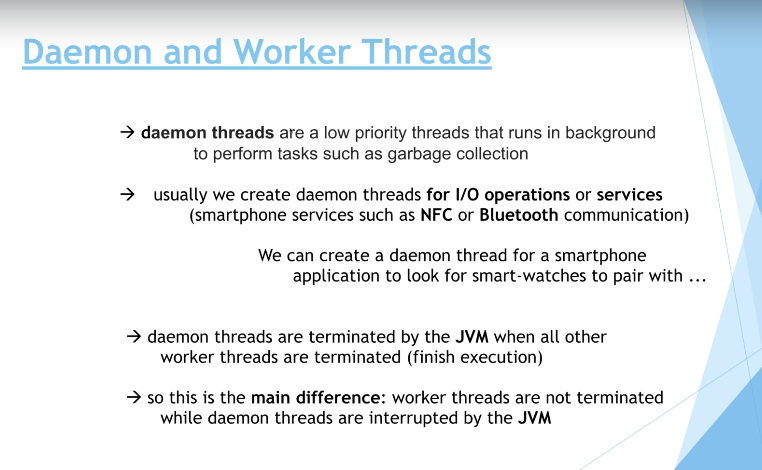
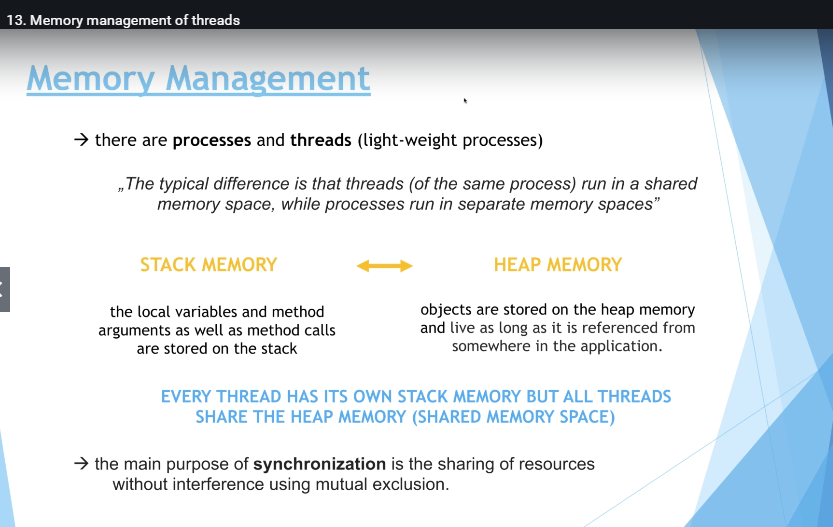
# Java multithrheading

# Threads manipulation



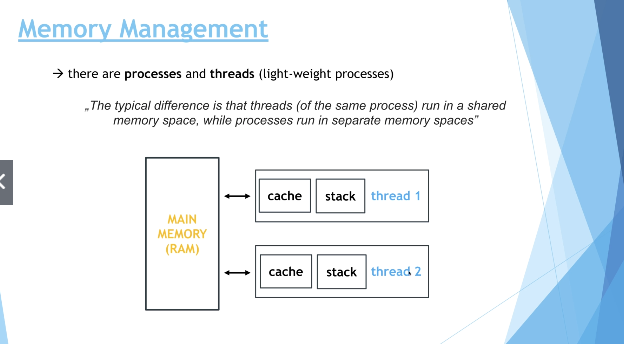


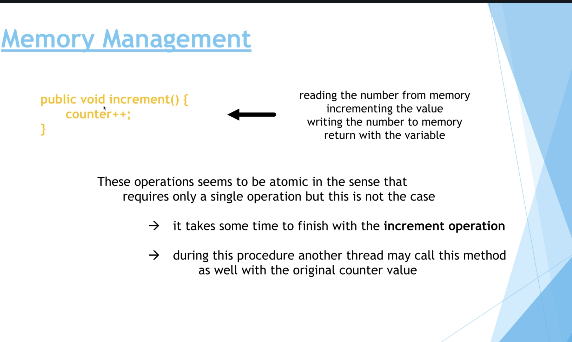


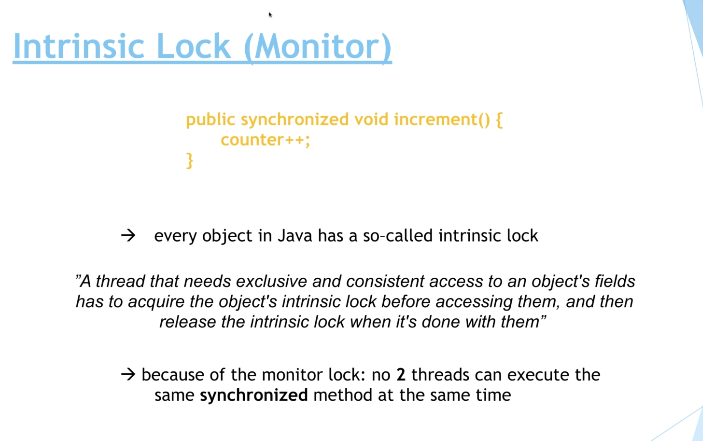


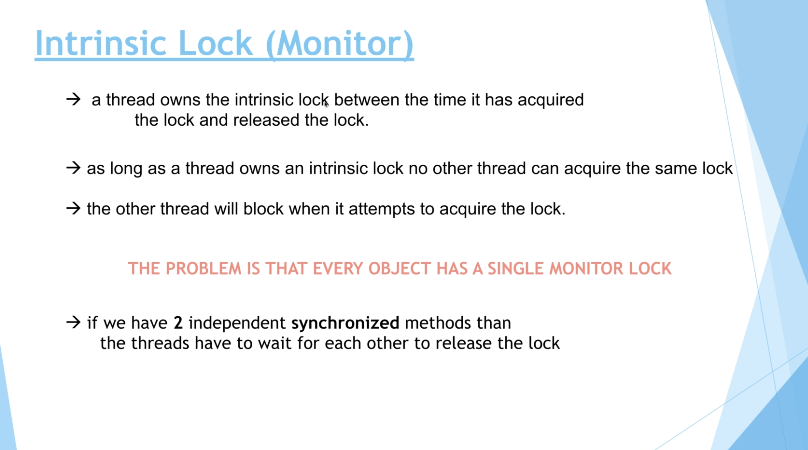
Stack memory is small and fast.

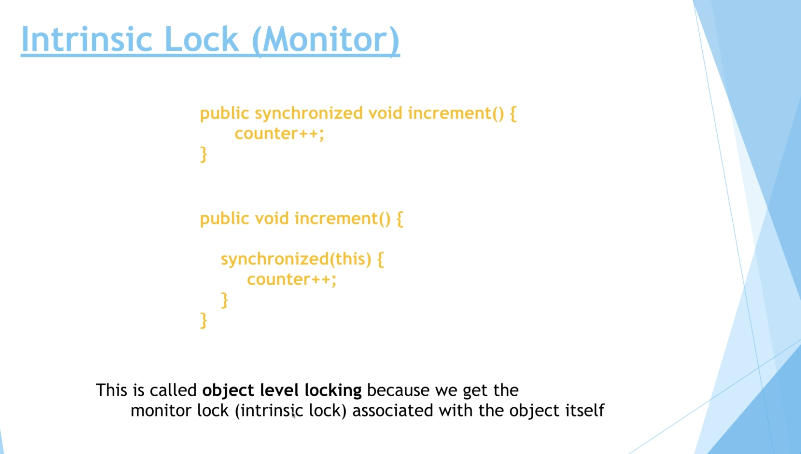
Heap memory is large and slow, objects.

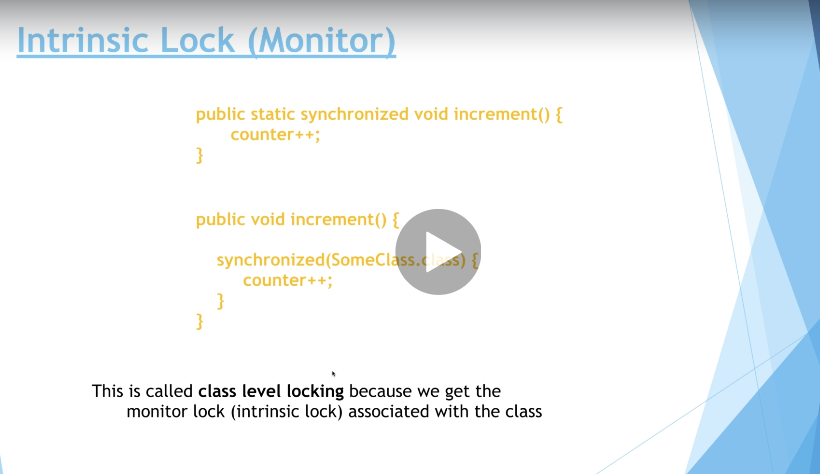












DIFFERENCE BETWEEN OBJECT LEVEL LOCKING AND CLASS LEVEL LOCKING!!! IF WE HAVE SYNCHRONIZED METHOD WE HAVE TO USE SYNCHRONIZED ON IN THE SYNCHRONIZATION BLOCK. IF WE DONT HAVE SYNCHRONIZED METHOD WE DONT HAVE TO DO THAT !!!!!!!

ITS BETTER TO USE SYNCHRONIZED BLOCKS THAT METHODS

What are re-entrant locks?

In the previous lectures we have been talking about locks (**intrinsic locks** or monitor locks).

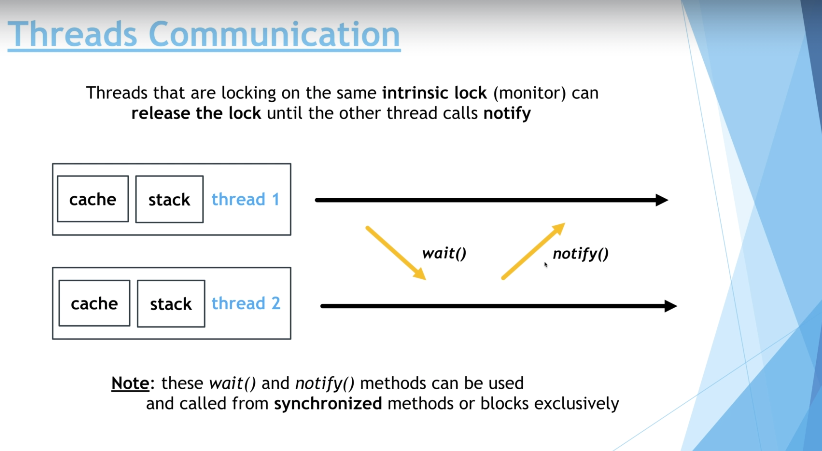
* there is a single intrinsic lock associated with every object or class in Java
* a given  thread that needs exclusive and consistent access to an object's fields

has to acquire the object's intrinsic lock before accessing them,

* and then the thread releases the intrinsic lock when it's done with them

Ok so a thread **cannot acquire a lock owned by another thread**. But a given thread **can acquire a lock that it already owns**. Allowing a thread to acquire the same lock more than once is called *re-entrant synchronization.*And this is exactly what is happening with Java - the same thread may acquire the lock more than once.

**For example**: let's consider recursive method calls. If a given thread calls a recursive and synchronized method several times then it is fine (note that in this case the same thread "enters" the synchronized block several times). There will be no deadlock because of re-entrant synchronization.

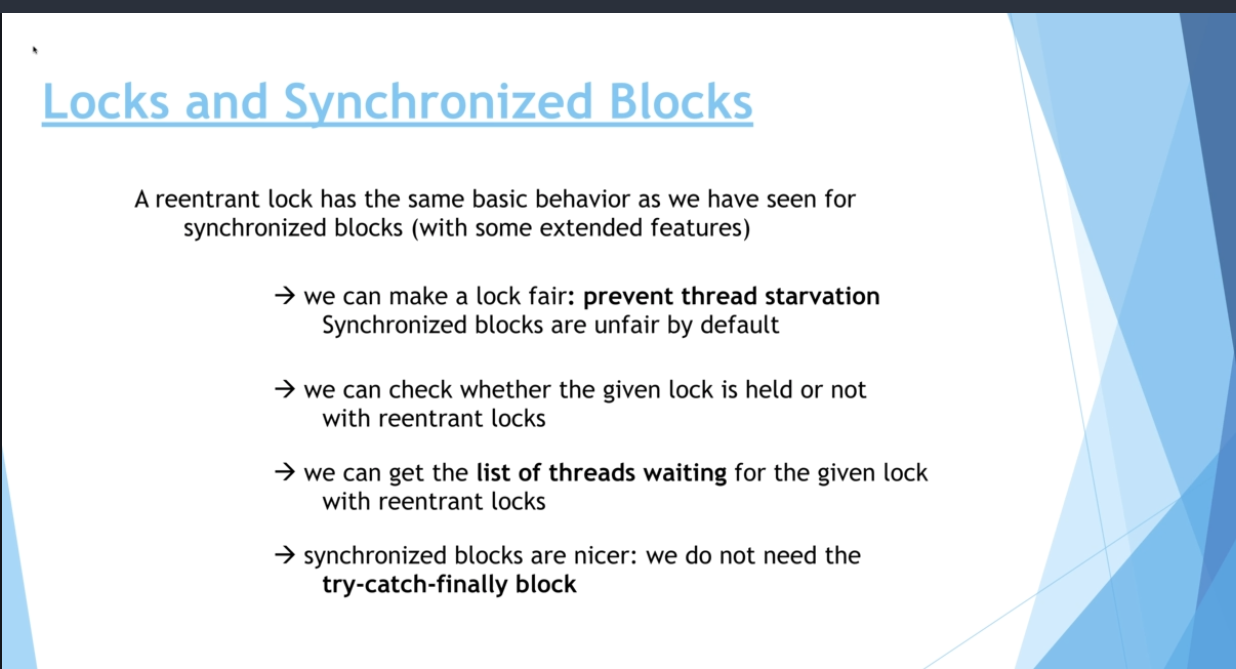


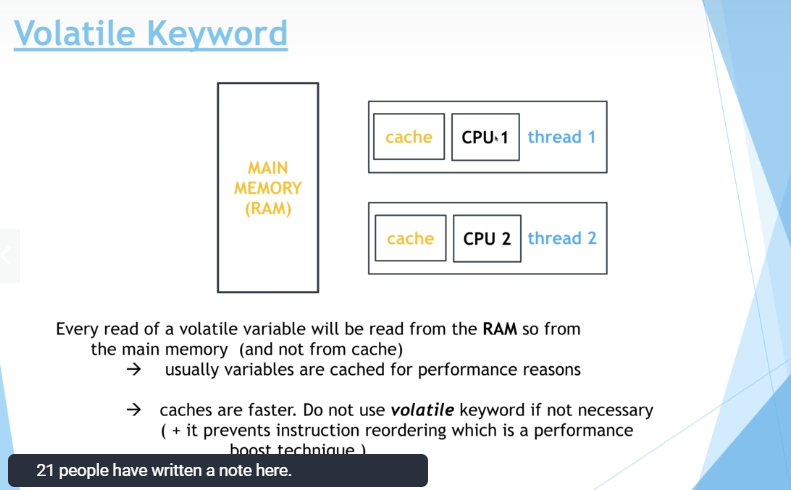
What is the difference between wait and sleep?

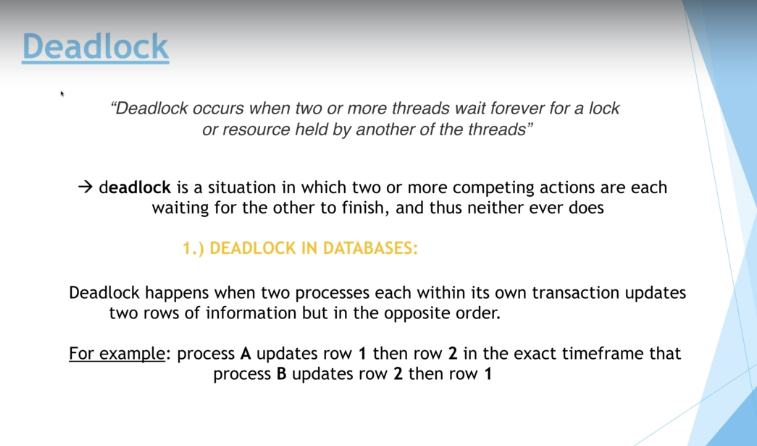
Let's discuss the difference between sleep and wait. They may seem to be very similar but there are fundamental differences between them.

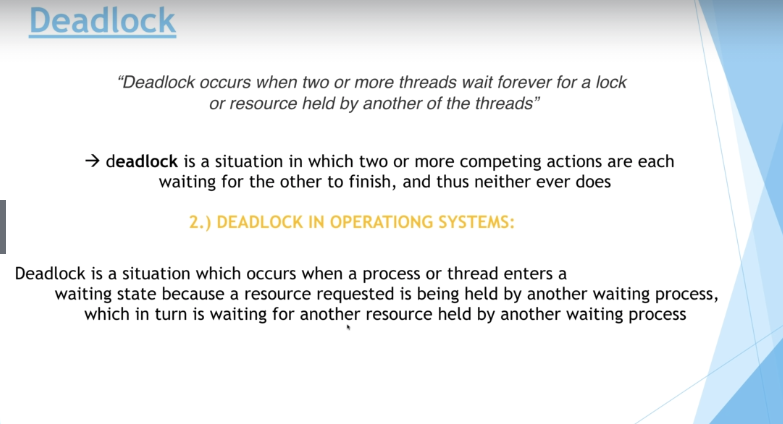
* you call wait on the **Object**while on the other hand you call sleepon the **Thread**itself
* wait can be interrupter (this is why we need the *InterruptedException*) while on the other hand sleep can not
* wait (and notify) must happen in a synchronized  block on the monitor object whereas sleep does not
* sleep operation does not release the locks it holds while on the other hand wait releases the lock on the object that wait() is called on

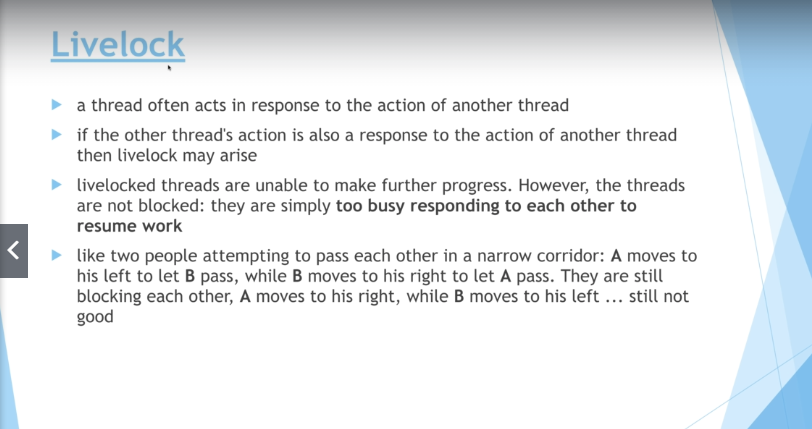
So as you can see there are some differences between wait and sleep operations!

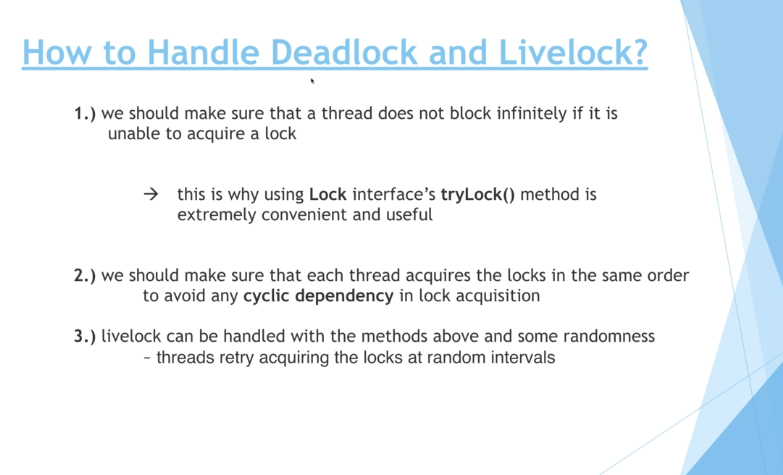


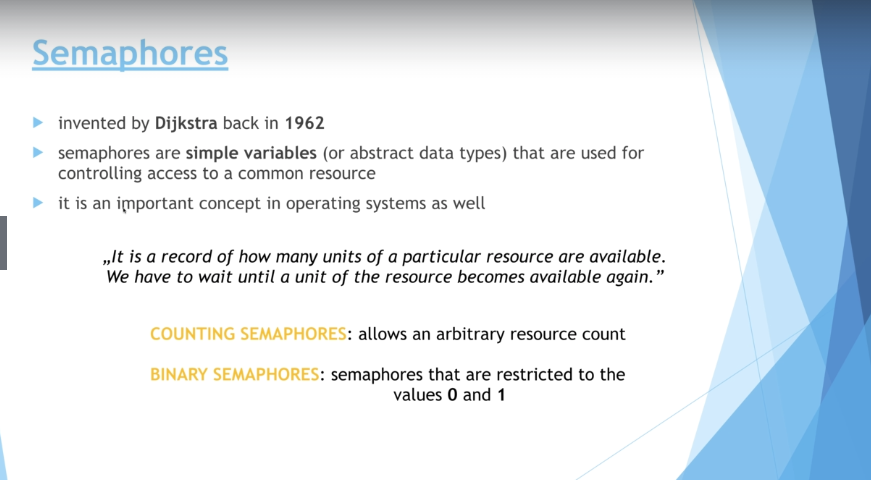


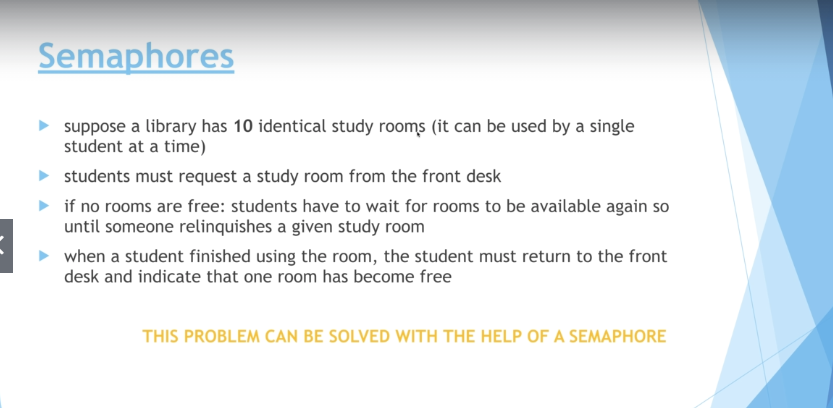


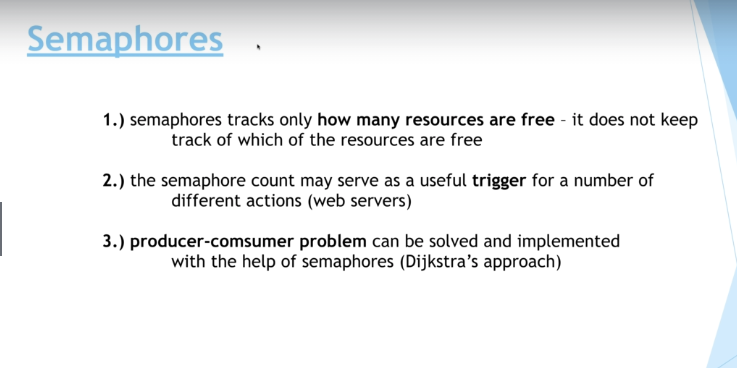


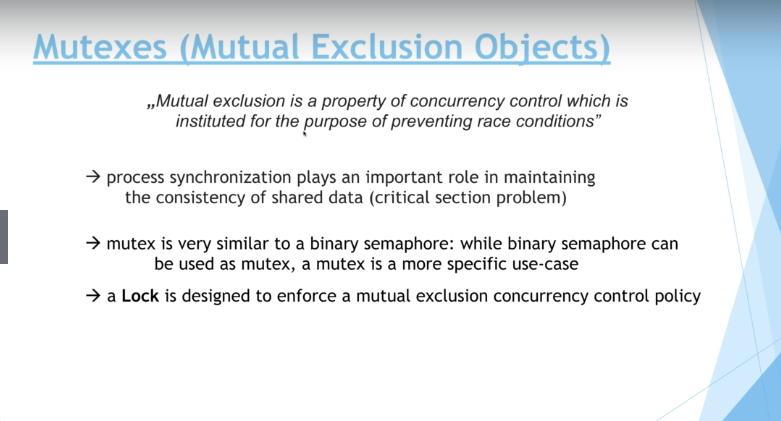


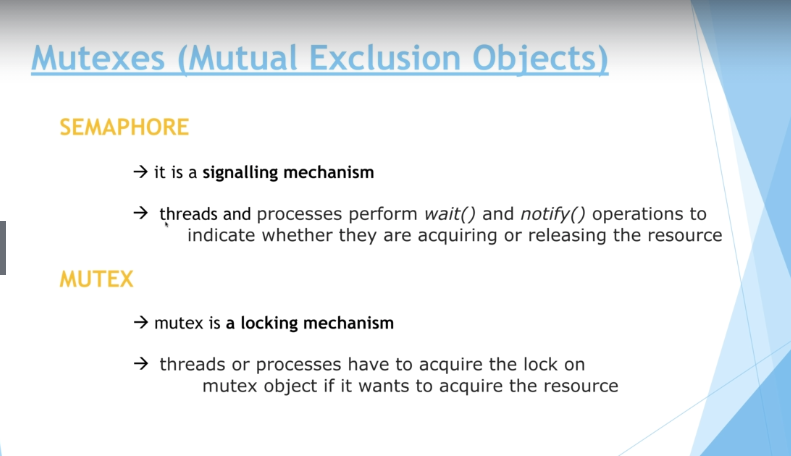


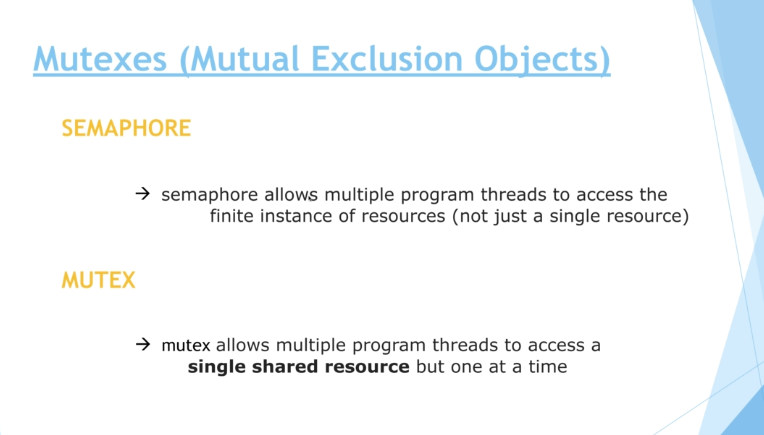


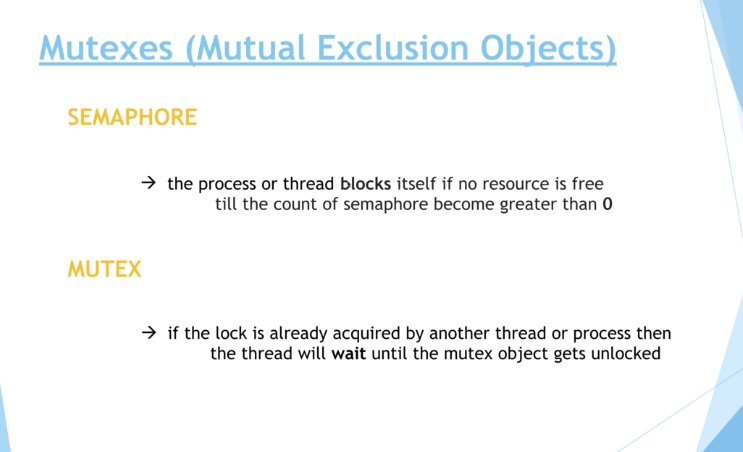












Mutex and binary semaphore

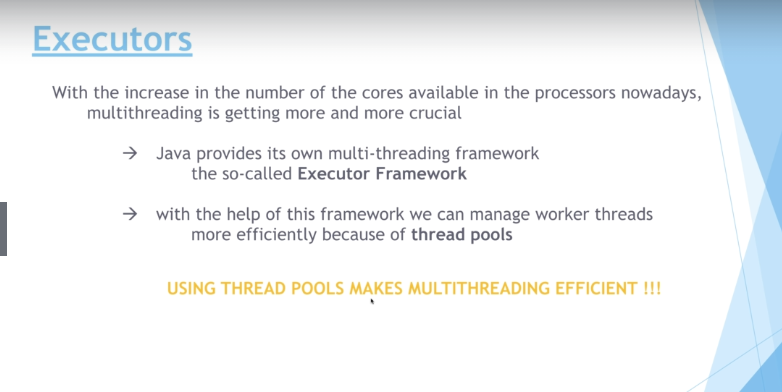
In the previous lectures we have been talking about binary semaphores and muteness. We have pointed out the fact that there are subtle differences between them. Let's consider the most critical difference again, the so-called **principle of ownership**.

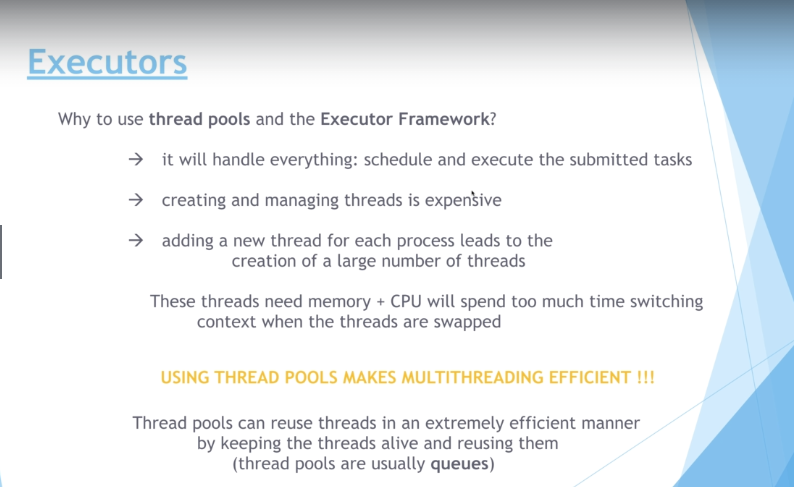
*"Ownership is the simple concept that when a task locks (acquires) a mutex only it can unlock (release) it"*

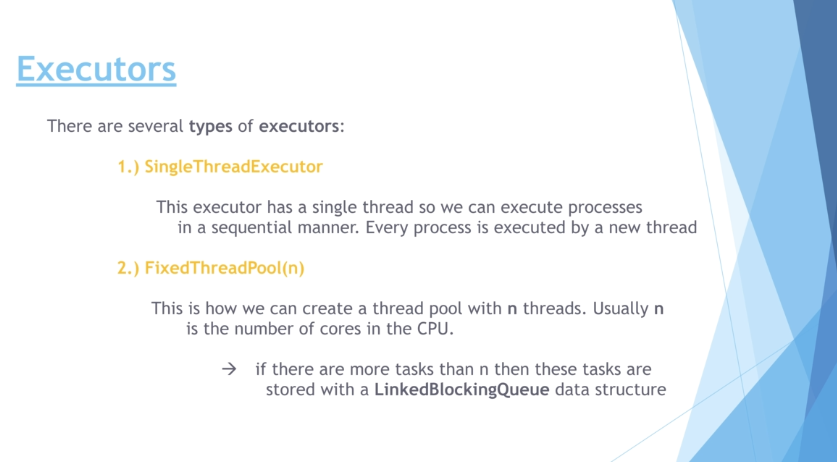
* a mutex can be owned by at most one thread at any given time while on the other hand binary semaphore has no concept of ownership
* if a task tries to unlock a mutex it hasn’t locked (thus doesn’t own) then an error condition is encountered and, most importantly, the mutex is not unlocked. If the mutual exclusion object doesn’t have ownership then, irrelevant of what it is called, it is not a mutex.

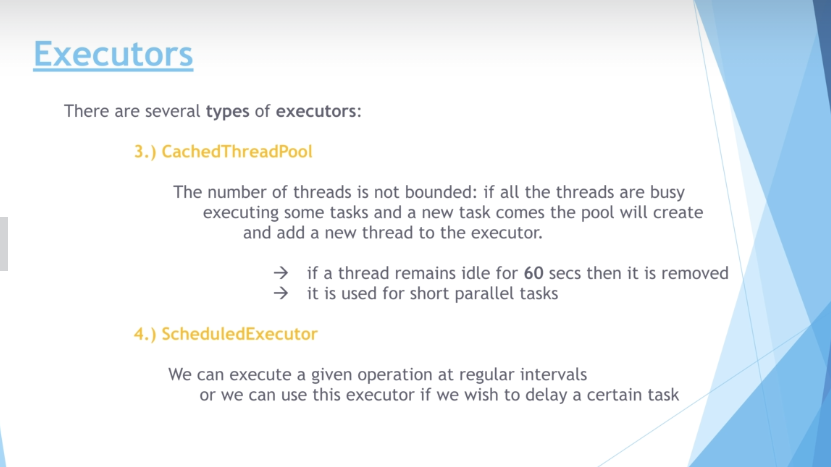
I just wanted to point out the crucial difference between binary semaphores and muteness! Hope you find it useful!

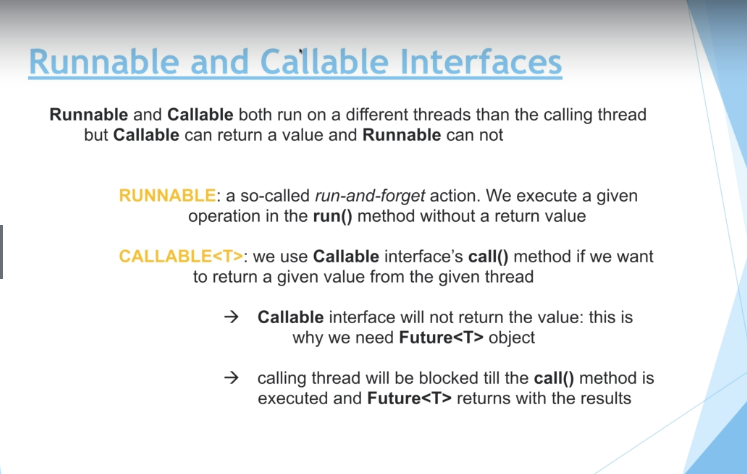
# Executors

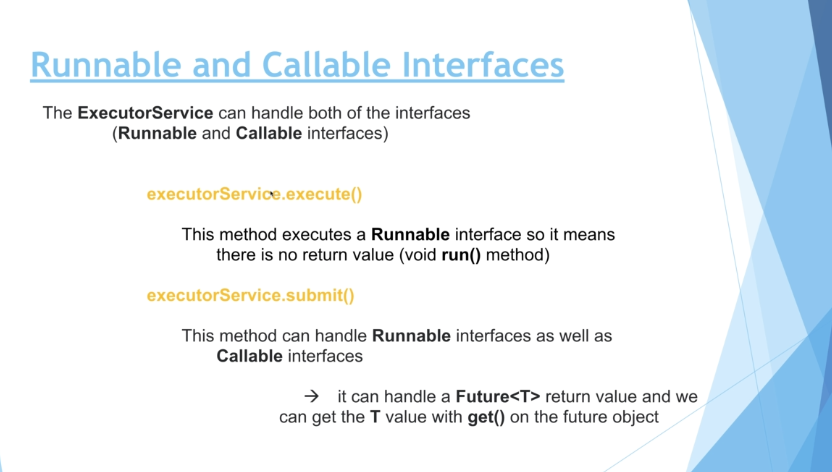












# Conccurent Data Structures

