Concurrency in Java

I skipped the theory part in the beginning of the lecture 17 of Java Cup series.

# The Thread Class and How to Create New Threads

You can either

* Extend the Thread class and override its run() method and call .start() on its instances

Or

* Implement the Runnable interface and feed it as a target to new instance of the Thread class and then call the .start() method on that instance

How it works:

By default, the Thread class in its run method calls the run method of its target which is of type runnable. If the target is null then it’ll return with no action. So, you need either to override this method, or feed the Thread instance with an implementation of Runnable.

* The run() method in Runnable and Thread defines the codes that will run in the new thread, it doesn’t create a new thread itself.
* The start() method is a low-level method that creates a new thread and calls the run() method.

The interface approach is preferable and is most commonly implemented since:

* You can extend another class when you are implementing Runnable but you can not when you extend Thread
* I guess: Other approaches and tools to create threads and shit are based on this approach

**So, no matter the approach there will be a new instance of the Thread class when creating a new thread whose start() method will be called.**

## Other Methods In the Thread class

### Instance methods:

* We talked about **run()** and **start()**
* **Join():**

when you call this method on a thread instance, the current thread that’s invoking this method will stop execution until the thread **on which the join method has been invoked has finished.**

It’s useful for example when you have already started a child thread, and done some things in the parent thread and at some point, you want to make sure that the child thread has finished and then proceed.

* **get/setPriority():**

is a number between 1 and 10 and the higher the **number** the more CPU time will be allotted to that thread and the OS runs that thread more:

thread.setPriority(Thread.MAX\_PRIORITY);thread.start();

**Default**: Thread.NORM\_PRIORITY = 5:

Thread.MIN\_PRIORITY = 1

Thread.MAX\_PRIORITY = 10

* **setDaemon()**

What are daemon threads:

* They run in the background and they usually**(?)** **serve other threads.** example: **the garbage collector**
* They don’t mean anything **by themselves** so:

If all the non-daemon threads have finished, the JVM closes the daemon threads and the program ends.

* By default, threads are not daemon and if you want them to be daemon threads, you have to explicitly set this property

thread = new Thread >> thread.setDaemon(true)>> thread.start()

* **getId()**

returns the unique identifier of the thread

### Static Methods

* + **Thread.currentThread():**

returns the current thread that its code is running right now

* + **Thread.sleep(m, n):**

**the current thread** will stop running for m milliseconds and n nanoseconds

# How Threads Consume Resources

* for each one of the threads one call stack is created, so threads have their own stacks
* if you put a breakpoint on one of the threads, by default Intellij will stop other threads from running. But you can configure the breakpoints to not block other threads and you can even run each thread step by step independent of other threads.

* **Objects are stored in Heap and local variables are stored on Stack**. **Each Thread has its own stack but all the threads share the Heap**.
* **If you have 2 threads that are calling the same method, each one of these threads has its own stack to store the local variables of that method but they both can be using the same objects in heap**

## Critical Sections and Synchronization

Sharing the Heap could cause a set of problems like:

* when a thread is reading an object that is being modified by another thread
* or two threads are updating the same object
* or one thread is working with a file and in the meantime another thread closes the file

The resources that are used by multiple threads are called **Shared Resources**. Variables, files, objects, devices , etc.

A situation where multiple threads try to access a shared resource and **at least one of them** wants to **make changes** on the resource is called a **Race Condition**

* The official definition of a Critical Section says that:

A **Critical Section** is a segment of the code that must be executed by only one thread at a time **to produce the expected results**. When more than one thread is allowed to execute this code segment, it could produce unpredictable results. So, If one thread is executing the critical section, the second thread must be stopped from executing it until the first thread is done.

But as I understood and I’m 99% certain about it, a critical section is a segment of each thread’s code that is accessing a shared resource, two threads can have critical sections that are not exactly the same piece of code but executing it could cause a race condition so caution must be taken so two threads can not execute parts of their routine that are prone to causing a race condition. So as per my own definition, a critical section is the code segment can lead to a race condition.

**Mutual Exclusion(Mutex):** When one thread enters the critical section, other threads must be stopped from executing the critical section. **Multiple threads should not execute the critical section.( or as per my own definition of a critical section, two threads should not execute the critical sections of their own routines which can cause a race condition on a particular shared resource, at the same time)**

## How Java Lets You Mange Race Conditions

### Locks and Monitors

* ATTENTION:

I searched a lot about the difference between locks and monitors and here are the results. I still might be slightly wrong though:

* There are articles talking about locks and monitors as two complementary entities where locks provide mutual exclusion and monitors make the cooperation between threads possible like what you have with wait and notify
* But the Oracle documentation on concurrency talks about an entity called the **Intrinsic Lock.** It says it’s also called the **Monitor Lock** and it says that the API specification sometimes refers to this simply as a **Monitor**. But I’ve seen it also refer to it as just **Lock**.
* **So, we continue by agreeing on the fact that all these terms are the same until proven otherwise**
* there is an intrinsic lock associated with every object, and if a thread wants to have an exclusive access to an object, it has to first acquire this lock and then release it when it’s done with it. But the thing is, other threads that access the object directly without acquiring a lock will still be able to modify the object( I tested it). However, if a thread is holding the lock of an object and another thread wants to acquire the lock, the second thread will be blocked.

### How the Locks are Acquired and Released

* When a thread calls a synchronized **instance** **method**, it automatically acquires the lock for that method’s object ( ***this* basically**), and will release the lock after the method returns or throws and exception causing the termination of that method.

If two threads are trying to invoke the same synchronized method on two different objects, there will be no blockage going one.

* When you call a **static** synchronized **method**, the thread acquires the lock for the *class* object associated with the class. So no other thread can call this method as they try to acquire the same lock.
* It’s possible for one thread to acquire the lock that it already owns. It’s helpful for example when a thread directly or indirectly calls a synchronized method that contains another synchronized method. This is called **Reentrant Synchronization** and makes a thread not block itself.
* If a thread is running a synchronized method and another thread is trying to run another synchronized method from the same object, it will block because it can’t acquire the lock to *this*.
* We also have **Synchronized blocks** that are helpful for more fine-grained control on what locks are acquired. You can pass **any** object to the synchronized blocks so the thread acquires the lock associated to that object before executing the block.
* So it’s the responsibility of the developer to demarcate the boundaries of critical sections where the locks should be acquired and released, and determine the object whose lock is to be acquired. Java will take care of the rest.
* It doesn’t make sense for constructors to be synchronized and it’s a syntax error as only the thread that is constructing the object should have access to it while being constructed.

You must not do dumb things like adding *this* to a list that other threads have access to. Oracle calls this premature leakage of the object.

* So critical sections are marked with the ***synchronized*** keyword
* **You can read about where you should use the synchronized methods or blocks later but as of now, it doesn’t have a point.**

……watched until the 01:05’:20”

# Inter-Thread Communication

* Sometimes a thread needs to wait for another thread to notify it like when a thread displaying the viruses waits for the thread finding the viruses.
* Notify and wait methods are implemented in the Object Class, They are final and use low-level, native implementations
* How they work: when a **thread** calls the wait method on an object, that thread stops and waits until **another thread** calls the notify method on the **same object**.