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

* interrupt()

we’ll talk about this method later on in this document.

### 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 or **any other** synchronized method because 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.

It makes sense for a synchronized block to acquire the lock of the object it wants to have an exclusive access to or potentially modify.

* Putting the content of a method inside a synchronized block with *this* as the lock, is equivalent to that method being synchronized

Here is an example of a synchronized block:

List<Integer> names = ….

Void addToNames(){

Synchronized(names){

…….

}

}

In the above method, two threads can enter the method if they are talking about 2 different instances of names. But they can’t if they are working with the same names object instance.

* 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.**

# Inter-Thread Communication

## Wait and Notify

Sometimes a thread needs to wait for another thread to notify it. like when a thread displaying the viruses waits for the thread that is finding the viruses.

It’s most obvious case is when a thread wants to do something with an object but the object’s sate is not where it should be, so the first thread calls **wait** on the object and waits for another thread that modifies the object to **notify** it that the object is in the state where the first thread can perform its operation.

Like when some consumers wait for a producer to produce something.

Notify and wait methods:

* Are implemented in the Object Class
* They are final so no one can override their behavior
* they use low-level, native implementations

How they work is: 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**.

* Threads must have the lock to the object they want to invoke wait() or notify() on
* If they try to invoke these methods on objects that they don’t have to lock to, they’ll get an *IllegalMonitorStateException*
* Each object can have a list of threads that are waiting on it not just one.
* When a thread calls wait() on an object, it stops and releases the lock to that object so other threads can acquire it and potentially call the notify method on it.
* **The thread will only release the lock to the object that it’s waiting on** and if it possesses another lock, they will be kept while the thread is waiting
* When the notify() method is invoked on an object, **one of** the threads is arbitrarily chosen to be awakened, it will not be able to continue until the thread calling the notify method releases the lock and the awakened thread acquires it. I add that I think(99% sure) for other waiting threads to continue, another thread must call the notify method again.
* **The awakened thread must again compete to acquire the object’s monitor like any other thread that is trying to do so and waiting threads have no special privilege in being the next thread that acquires the lock**
* You can call notifyAll() to awaken all the waiting threads
* You can also set a timeout for the waiting, if the timeout has been exceeded, this thread stops waiting for another thread to notify it and the thread will again compete to acquire the lock and if it gets the lock, it’ll continue execution. If the time out is set to zero the thread will wait until the notify method is called. The wait() method invokes wait with zero timeout
* A thread that is waiting can also stop waiting if another thread interrupts it
* I’m 99.8% sure that if a thread stops waiting because of being notified, a time out or interruption, in either situation it has to acquire the lock again to be able to continue.
* **But is it able to go to the catch block of the InterrupedException?**

Example:

Synchronized method(){

Wait()-> is calling wait on this

}

Synchronized(obj){

obj.notify() -> obj in this thread corresponds to the this in the first thread.

}

## Interrupts

Another concept that makes the interaction between threads possible is the concept of Interrupt.

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is **very common for the thread to terminate.** Here’s an example:

for (int i = 0; i < importantInfo.length; i++) {

// Pause for 4 seconds

try {

Thread.sleep(4000);

} **catch (InterruptedException e) {**

**// We've been interrupted: no more messages.**

**return**;

}

// Print a message

System.out.println(importantInfo[i]);

}

**So, interruption of a thread doesn’t necessarily mean that it will be terminated.**

### The Interrupt status Flag

The interrupt mechanism is implemented using an internal flag known as the *interrupt status*.:

***private volatile boolean interrupted;***

**it’s read/written directly by the JVM and indicates whether a thread has been interrupted.**

When a thread checks for an interrupt by invoking the **static** method **Thread.interrupted()**:

* The interrupt flag of the **current** thread is looked up
* If the interrupt flag is cleared(false), a false will be returned
* If it’s set (true), it will be cleared(set to false) and a true will be returned

The **non-static** isInterrupted method, which is **used by one thread** **to query** the interrupt status of **another**, does not change the interrupt status flag.

**I add**: And it makes sense, because the static method returns the interrupt status of the current thread, so if it’s looking it up it means it’s started working or something like that but the instance method is called but other threads so it doesn’t mean that the thread has come out of the interrupt state.

### The interrupt() method

If a thread is in a block state due to invocation of:

* the wait() method on an object
* the join() method on another thread
* Or Thread.sleep()

The ***interrupt()*** method can be invoked on this thread and as a result the thread will be out of the block state and an ***InterrupedException*** which is a checked exception, is thrown by the above methods.

**The interrupt method seems to set the interrupt status flag, however, by convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.**

* so, if you call the interrupt method in the above cases, the interrupt status class will be **cleared** and an ***InterruptedException*** will be thrown.

Note:

The behavior of this method is different for blockage due to I/O operations or nio channel selectors and as I checked will set the interrupt flag but I don’t get much into it now.

* If none of the above conditions, the first three and the io stuff are the case, the interrupt flag will be set
* Interruption of a thread that’s not alive(*the conditions of alive threads come in the next section*) is reported by isInterrupted and interrupted methods. but it doesn’t have any effect right?

# A Thread’s Life Cycle

A thread is alive if it has been started and has not yet died.

*/\*\*  
 \* A thread state. A thread can be in one of the following states:  
 \* <ul>  
 \* <li>{@link #NEW}<br>  
 \* A thread that has not yet started is in this state.  
 \* </li>  
 \* <li>{@link #RUNNABLE}<br>  
 \* A thread executing in the Java virtual machine is in this state.  
 \* </li>  
 \* <li>{@link #BLOCKED}<br>  
 \* A thread that is blocked waiting for a monitor lock  
 \* is in this state.  
 \* </li>  
 \* <li>{@link #WAITING}<br>  
 \* A thread that is waiting indefinitely for another thread to  
 \* perform a particular action is in this state.  
 \* </li>  
 \* <li>{@link #TIMED\_WAITING}<br>  
 \* A thread that is waiting for another thread to perform an action  
 \* for up to a specified waiting time is in this state.  
 \* </li>  
 \* <li>{@link #TERMINATED}<br>  
 \* A thread that has exited is in this state.  
 \* </li>  
 \* </ul>  
 \*  
 \* <p>  
 \* A thread can be in only one state at a given point in time.  
 \* These states are virtual machine states which do not reflect  
 \* any operating system thread states.  
 \*  
 \* @since 1.5  
 \* @see #getState  
 \*/*public enum State {  
 */\*\*  
 \* Thread state for a thread which has not yet started.  
 \*/  
 NEW*,  
  
 */\*\*  
 \* Thread state for a runnable thread. A thread in the runnable  
 \* state is executing in the Java virtual machine but it may  
 \* be waiting for other resources from the operating system  
 \* such as processor.  
 \*/  
 RUNNABLE*,  
  
 */\*\*  
 \* Thread state for a thread blocked waiting for a monitor lock.  
 \* A thread in the blocked state is waiting for a monitor lock  
 \* to enter a synchronized block/method or  
 \* reenter a synchronized block/method after calling  
 \* {@link Object#wait() Object.wait}.  
 \*/  
 BLOCKED*,  
  
 */\*\*  
 \* Thread state for a waiting thread.  
 \* A thread is in the waiting state due to calling one of the  
 \* following methods:  
 \* <ul>  
 \* <li>{@link Object#wait() Object.wait} with no timeout</li>  
 \* <li>{@link #join() Thread.join} with no timeout</li>  
 \* <li>{@link LockSupport#park() LockSupport.park}</li>  
 \* </ul>  
 \*  
 \* <p>A thread in the waiting state is waiting for another thread to  
 \* perform a particular action.  
 \*  
 \* For example, a thread that has called {@code Object.wait()}  
 \* on an object is waiting for another thread to call  
 \* {@code Object.notify()} or {@code Object.notifyAll()} on  
 \* that object. A thread that has called {@code Thread.join()}  
 \* is waiting for a specified thread to terminate.  
 \*/  
 WAITING*,  
  
 */\*\*  
 \* Thread state for a waiting thread with a specified waiting time.  
 \* A thread is in the timed waiting state due to calling one of  
 \* the following methods with a specified positive waiting time:  
 \* <ul>  
 \* <li>{@link #sleep Thread.sleep}</li>  
 \* <li>{@link Object#wait(long) Object.wait} with timeout</li>  
 \* <li>{@link #join(long) Thread.join} with timeout</li>  
 \* <li>{@link LockSupport#parkNanos LockSupport.parkNanos}</li>  
 \* <li>{@link LockSupport#parkUntil LockSupport.parkUntil}</li>  
 \* </ul>  
 \*/  
 TIMED\_WAITING*,  
  
 */\*\*  
 \* Thread state for a terminated thread.  
 \* The thread has completed execution.  
 \*/  
 TERMINATED*;  
}

# The Producer/Consumer Problem

READ ABOUT CPU CACCING, AND OS RAM and Their Role in Concurrency

https://jenkov.com/tutorials/java-concurrency/java-memory-model.html