Advanced Concurrency Topics

We talked about low-level concepts and tools to manage concurrency in java.

The Thread class is placed in the java.lang package.

There are some High-level APIs to help us manage concurrency that were added since Java5 and are available in the **java.util.concurrency** package

These APIs are usually a better approach to handle concurrency compared to those old, low-level tools that we saw in the first document:

* They **usually** have a better performance, they don’t simply make everything synchronized for example.
* They can make a better use of modern multi-core CPUs that can perform parallel processing
* They are easier to use
* They offer different solutions to different use cases and for example you’re not left with only the synchronized keyword to handle concurrency.

# Thread-Safety and Thread-Safe Classes

Instances of classes that are thread safe can be used as a shared resource in multiple threads in a safe way without causing a synchronization problem.

To use the instances of these classes you won’t need to use locks, synchronized block/methods, etc.

They are either inherently thread-safe like immutable classes, or all the precautions are built in these classes and we won’t be thinking about them like ConcurrentHashmap.

**Every time you want to use a thread-safe, utility class you have to read the documentation and use the methods that are thread-safe because these classes can have some methods that don’t do their job in a thread-safe way.**

**You must also check for how they acquire locks for example, do they block on read operations?**

Examples of such classes are:

|  |  |
| --- | --- |
| Non-Thread-Safe Class | The Thread-Safe Equivalent |
| ArrayList | Vector |
| HashMap | ConcurrentHashMap |
| StringBuilder | StringBuffer |

**Some notes:**

* As you know the String class is thread-safe cause it’s immutable. The StringBuffer and StringBuilder are mutable classes for manipulating strings. The StringBuffer class makes its instances thread-safe by making its methods synchronous. However, there **are still some non-thread safe methods in this class that aren’t synchronized**.
* So, pay attention to the source code of the methods you use on this kind of thread-safe classes
* Some of the synchronized methods are an overridden form of their parent interface that don’t have the synchronized key-word in their signature. This was interesting

## Thread-Safe Classes or Normal Classes, which One to Use?

Why don’t we just always use thread-safe classes?

* First of all, we should know that immutable classes like *String* or *Integer* are inherently thread-safe since they’re always read not modified. So, in that sense they are not different from normal classes
* But when it comes to classes that provide a support for concurrency by for example synchronized blocks or other approaches, they have a worse performance due to these precautions
* So, if you don’t need to use an object as a shared resource where multiple threads will work with it, you better off using the normal classes

# Concurrent Collections

Side notes:

Maps are not subtypes of the Collection interface but I think they’re all part of the Java Collection API, I’m like 10% sure about this.

* A simple way to make a class thread-safe is to make all the methods synchronized but this certainly won’t have a good performance
* For example, you could only use one of the methods at a time as they all need to acquire the lock to this
* The java.util.concurrency offers data-structures that are thread-safe, and implement this thread-safety in an efficient way with decent performance in concurrent applications
* for example, they may acquire/release locks in a more efficient way

some of the examples are: **ArrayBlockingQueue, CopyOnWriteArrayList, ConcurrentHashMap**

## BlockingQueue

There are multiple implementations of this interface like *ArrayBlockingQueue* that uses an array as the underlying data structure and LinkedBlockingQueue that uses linked lists.

They are basically queues that are thread-safe an provide some additional handy features that you will get when you use the ***put()*** and ***take()*** methods to insert and remove data:

If you try to read/remove from an empty queue the thread waits until the queue has a value in it

If you try to insert into a full queue, the thread waits until there is room.

So basically, this will help you solve the Producer/Consumer Problem with ease.

## ConcurrentHashMap

In my quick investigation that **needs more attention and time**, I realized that concurrent hash maps don’t acquire locks on retrievals(it needs more introspection) so they’re a highly-concurrent implementation, on the other hand, HashTables simply lock the entire table for reads and writes so you may want to use ConcurrentHashMaps where you need a stricter synchronization

Watched till 00:21’:00”…

# Synchronizers

A synchronizer object is used to control the execution of multiple threads.

These objects have an inner state which is used to manage the involved threads to let them wait or resume execution for example.

There are some utility classes to facilitate this process that I think are alternatives to methods like wait() and notify():

## Semaphore

* It controls access to shared resources **through using a counter**
* You set an **initial** limit to the number of permits to access the shared resource
* the counter maintains the number of available permits
* When a thread wants to access the shared resource, it calls the acquire() method, if the counter is greater than 0, a permit is granted and the counter reduces by 1
* When a thread is done with the shared resource, it calls release() method, the counter is incremented by1, and the permit will be available for others to acquire
* If the counter number is **not** greater than 0, a thread will be blocked upon trying to acquire a permit until a permit can be acquired

## CountDownLatch

## Exchanger

## CyclicBarrier

## Phaser

# Atomic Operations and Classes