11 - Concurrency

- The ScheduledExecutorService is an ExecutorService that can schedule commands to run after a given delay, or to execute periodically. The schedule methods create tasks with various delays and return a task object that can be used to cancel or check execution. The scheduleAtFixedRate and scheduleWithFixedDelay methods create and execute tasks that run periodically until cancelled. There is no FixedDelayExecutorService!
- A class implementing the Callable interface, must implement the method call():

```
public interface Callable<V>{
  V call() throws Exception;
}
```

- Callable is a task that returns a result and may throw an exception. Implementers define a single method with no arguments called call. The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.
- Future is an interface, so new Future ("Data from callable") will not compile!
- A Callable should return actual data object instead of wrapping the data into a Future. It is the job of ExecutorService.submit() method to return a Future that wraps the data returned by Callable.call(). `Future result = Executors.newSingleThreadExecutor().submit(new MyTask())`
- `Future.get()` will block until there is a value to return or there is an exception. Therefore, the program will block at and will print the return value of the given Callable once it is done. If you don't want to block the code, you may use `Future.isDone()`, which returns a boolean without blocking.
- `List<Runnable> shutdownNow()` is a method of the ExecutorService Interface. This method attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution. This method does not wait for actively executing tasks to terminate. Use awaitTermination to do that.
- Iterators retrieved from a ConcurrentHashMap are backed by that ConcurrentHashMap, which means any operations done on the ConcurrentHashMap instance may be reflected in the Iterator.
- entrySet() returns a Set view of the mappings contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.
- The Runnable interface doesn't have a method start(). The class Thread does!
- java.util.concurrent.Executor is an interface that has only one method: void execute(Runnable command) Java concurrent library has several implementations for this interface such as ForkJoinPool, ScheduledThreadPoolExecutor, and ThreadPoolExecutor.
- CopyOnWriteArrayList guarantees that the Iterator acquired from its instance will never get a ConcurrentModificationException. This is made possible by creating a copy of the underlying array of

the data. The Iterator is backed by this duplicate array. An implication of this is that any modifications done to the list are not reflected in the Iterator and no modifications can be done on the list using that Iterator. Calls that try to modify the iterator will get UnsupportedOperationException.
- The synchronized keyword cannot be applied to variables.
- You can add null elements in CopyOnWriteArrayList (as well as in a regular ArrayList). Remember that HashMap supports adding null key as well as null values but ConcurrentHashMap does not.
- Multiple threads can safely add and remove objects from a CopyOnWriteArrayList simultaneously.
- The forEachOrdered method processes the elements of the stream in the order they are present in the underlying source, even if the stream is a parallel stream.

# A few important classes in java.util.concurrent package:

1. ExecutorService interface extends Executor interface. While Executor allows you to execute a Runnable, ExecutorService allows you to execute a Callable.
2. Executors is a utility class that provides several static methods to create instances of ExecutorService. All such methods start with new e.g. newSingleThreadExecutor(). You should at least remember the following methods: newFixedThreadPool(int noOfThreads), newSingleThreadExecutor(), newCachedThreadPool(), newSingleThreadScheduledExecutor(), newScheduledThreadPool(int corePoolSize).

# fork/join framework:

- This is the general logic of how the fork/join framework is used:
    1. First check whether the task is small enough to be performed directly without forking. If so, perform it without forking.
    2. If no, then split the task into multiple small tasks (at least 2) and submit the subtasks back to the pool using `invokeAll(list of tasks)`.

- The number of threads created depends on how the ForkJoinPool is created. By default, it uses the number processors available.
- A ForkJoinPool differs from other kinds of ExecutorService mainly by virtue of employing work-stealing. This means that if a worker thread is done with a task, it will pick up a new task irrespective of which thread created that task. In a fork/join framework, any worker thread may spawn new tasks and it is not necessary that the tasks spawned by a thread will be executed by that particular thread. They can be executed by any available thread.
- The order of join() and compute() is critical. Remember that fork() causes the sub-task to be submitted to the pool and another thread can execute that task in parallel to the current thread. Therefore, if you call join() on the newly created sub task, you are basically waiting until that task finishes. This means you are using up both the threads (current thread and another thread from the pool that executes the subtask) for that sub task. Instead of waiting, you should use the current thread to compute another subtask and when done, wait for another thread to finish. This means, both the threads will execute their respective tasks in parallel instead of in sequence.
- There is a cost associated with forking a new task. If the cost of actually finishing the task without

forking new subtasks is less, then there is not much benefit in breaking a task into smaller units. Therefore, a balance needs to be reached where the cost of forking is less than direct computation. THRESHOLD determines that level. THRESHOLD value can be different for different tasks. THRESHOLD is a standard concept used while subdividing a big task. It ensures that a task is not divided so much that the cost of creation of sub tasks becomes higher than directly computing the task itself.

- The whole objective is to divide the task such that multiple threads can compute it in parallel. Therefore, it is better if two sub tasks are equal in terms of cost of computation, otherwise, one thread will finish earlier than the other thereby reducing performance.
- This is a standard way of using the Fork/Join framework: You create a RecursiveTask or RecursiveAction (depending on where you need to return a value or not) and in that RecursiveTask, you subdivide the task into two equal parts. You then fork out one of the halfs and compute the second half.
- There is no need to submit the sub tasks to the pool using the submit method. Once you fork a subtask using the fork method, it is automatically executed by the pool.
- RecursiveAction/Task is not an interface. It is an abstract class. So you need to subclass RecursiveAction/Task instead of implementing it.
- The worker threads in the ForkJoinPool extend java.lang.Thread and are created by a factory. By default, they are created by the default thread factory but another factory may be passed in the constructor.
- ForkJoinPool implements Executor and not the threads in the pool.
- RecursiveAction is used when a task does not return a value. In this case, we want it to return an integer value so it should extend RecursiveTask.
- (Math.random(1, 10) is not correct. There is no such method named random that takes two arguments in Math class. The correct signature is `public static double random()` and it returns a pseudorandom double greater than or equal to 0.0 and less than 1.0.
- When `Math.random()` is first called, it creates a single new pseudorandom-number generator, exactly as if by the expression new java.util.Random(). This method is properly synchronized to allow correct use by more than one thread. In other words, Math.random() is a synchronized method, so it can be used by multiple threads but at the cost of performance.

# ReadWriteLock:

- From a ReadWriteLock, you can get one read lock (by calling `lock.readLock()`) and one write lock (by calling `lock.writeLock()`). Even if you call these methods multiple times, the same lock is returned.
- A read lock can be locked by multiple threads simultaneously (by calling `lock.readLock().lock()`), if the write lock is free. If the write lock is not free, a read lock cannot be locked.
- The write lock can be locked (by calling `lock.writeLock().lock()`) only by one thread and only when no thread already has a read lock or the write lock.
- In other words, if one thread is reading, other threads can read, but no thread can write. If one thread

is writing, no other thread can read or write.
- ReadWriteLock interface does not have lock and unlock methods. It has only two methods - readLock and writeLock. Neither one takes any argument.

# ReentrantLock:

- The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. As the name says, ReentrantLock allow threads to enter into lock on a resource more than once.
- `Lock.lock()` returns void. `Lock.tryLock()` returns boolean.

# java.util.concurrent.ConcurrentMap:

- It is a Map providing additional atomic putIfAbsent, remove, and replace methods.
- default methods in Map interface:
  - `V putIfAbsent(K key, V value)`
  - `boolean remove(Object key, Object value)`
  - `boolean replace(K key, V oldValue, V newValue)`

# java.util.concurrent.atomic.AtomicInteger:

- `incrementAndGet()` atomically increments the current value by 1 and returns the new value.
- `addAndGet(1)` atomically adds the given value to the current value and returns the new value.
- `getAndIncrement()` atomically increments the current value by 1 but it will return the old value.
- `getAndSet(6)` will set the variable to 6 but it will return the old value
- AtomicInteger is not a wrapper class.
- `public final boolean compareAndSet(int expect, int update)` - Atomically sets the value to the given updated value if the current value == the expected value.

  - Parameters: expect - the expected value update - the new value
  - Returns: true if successful. False return indicates that the actual value was not equal to the expected value.

- Using `if(oldstatus == status.get()) status.set(newstatus);` is valid code but is not thread safe. The value can potentially change after comparison and just befor it is set again.

# public class ThreadLocalRandom extends Random:

- A random number generator isolated to the current thread. Like the global Random generator used by the Math class, a ThreadLocalRandom is initialized with an internally generated seed that may not otherwise be modified. When applicable, use of ThreadLocalRandom rather than shared Random objects in concurrent programs will typically encounter much less overhead and contention.
- Use of ThreadLocalRandom is particularly appropriate when multiple tasks (for example, each a ForkJoinTask) use random numbers in parallel in thread pools.
- Usages of this class should typically be of the form:

`ThreadLocalRandom.current().nextX(...)` (where X is Int, Long, etc). When all usages are of this form, it is never possible to accidently share a ThreadLocalRandom across multiple threads.

< Previous | Next >