latex input: mmd-article-header Title: Java Concurrency Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Java, programming language, syntax, fundamentals CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2014 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

- The programmer must ensure read and write access to objects is properly coordinated (or "synchronized") between threads
    - Including preventing threads from accessing partially updated objects during modification by another thread
    - Use the built-in constructs
- Multiple processes implies multiple JVMs; here, we're referring to *multi-**threaded*** programming
- Rule of Thumb
    1. Try to use objects in `java.util.concurrent` instead of figuring out concurrency for yourself
    2. Try to use `synchronized` over using actual `Lock`/`Condition` objects
- Brian Goetz (author of *Java Concurrency in Practice*) coined the following "synchronization motto":
    - "If you write a variable which may next be read by another thread, or you read a variable which may have last been written by another thread, you must use synchronization."
- Definitions are hazy
    - **Synchronization** --- generally means sharing data between multiple processors or threads
    - **Concurrency** refers to a measure of (or the art of improving) how effectively an application allows multiple jobs required by that application (e.g. serving web page requests from a web server) to run simultaneously.
- Java is generally a good choice of language for multiprocessor applications, both because of in-built concepts fundamental to the language and because of a relatively rich concurrency library provided from Java 5 onwards.

Refs

- javamex

Thread Objects

- **Threads share the process's resources, including memory and open files**
- Every application has at least one thread called the *main thread*
- Each thread can be scheduled on a different CPU core
- Each thread is associated with an instance of the `class Thread`

- Threads are always in 1 of 6 states
    - New --- instantiated, but you haven't called `start()` yet
    - Runnable --- up to OS whether it is *actually* running
    - *Inactive*, i.e.
        - Blocked --- waiting to acquire an *intrinsic object lock*
        - Waiting --- waiting for a `java.util.concurrent` `Condition` or `Lock` object
        - Timed waiting --- waiting for `Thread.sleep(time)` or similar
    - Terminated --- either `run()` exited normally, or an uncaught exception terminated it
- The `stop()` and `suspend()` methods, which directly make a thread stop, have been *deprecated* because they could be stopped while the data structure they're operating upon is in an *inconsistent* state.

Two ways to start a thread

Provide your own Runnable to a Thread

This is the **preferred** method. "The general contract of the method `run` is that it may take any action whatsoever."

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from thread!");
    }
    public static void main(String[] args) {
        Runnable hello = new HelloRunnable();
        Thread thread = new Thread(hello);
        thread.start();
    }
}
```

Subclass Thread

The *other* way is *preferred*.

```
public class HelloThread extends Thread {
   public void run() { System.out.println("Hello from thread!"); }
   public static void main(String[] args) { new HelloThread().start(
}
```

Thread basics

- Wait for the completion of another thread with `myThreadInstance.join(timeout)`
- To *synchronize* threads, Java uses **monitors**, a mechanism allowing only one thread at a time to execute a region of code protected by the monitor
- You can give threads *priority* levels
    - `void setPriority(int priority)`
    - `Thread.MIN_PRIORITY = 1`, `MAX_PRIORITY = 10` (reverse order from Unix)
    - High priority threads
        - User interaction
        - Short tasks
    - Low priority
        - Long tasks
    - The VM *generally* runs *only* the highest-priority task, but not always
        - This makes it easy to accidentally *starve* low-priority tasks
- You can designate a thread to be a *daemon* meaning the JVM will exit if only *daemon* threads are still executing

Interrupting Threads

Say you have a "runnable" `Thread` instance:

```
Thread thr = new Thread(new MyRunnable());
thread.start();
```

Now it's off doing its thang, but you'd like to kindly ask it to stop when it gets a chance. So you call `interrupt()` on the instance:

```
thread.interrupt();
```

Now in `MyRunnable implements Runnable` you defined `run()`. Some of the methods your `Runnable` can call (e.g. `Thread.sleep(millisec)`) can `throw InterruptedException` which will happen after you called `interrupt()` above. At this point before you call `sleep()` or any other blocking call, you can check if you have been interrupted with

```
boolean Thread.currentThread().isInterrupted()
```

and perform actions before you actually throw the exception. But then you throw it, and you can catch it too, which is where you should probably interrupt yourself in whatever way is apropos.

Implementing Callbacks

This is based on Chapter 3 of *Harold, Elliotte Rusty (2013-10-04). Java Network Programming. O'Reilly Media. Kindle Edition.*

- The reason we create an "`instanceOfMe`" is to show what it looks like when the `callbackHandler()` is not `static`, which it *would* have to be if we didn't create an instance of *something* to call the `callbackHandler` on.
- The reason we don't spawn and start the other thread in the constructor is becase the thread could try to call back before the constructor has finished initializing the object!
- If many objects have the same callbacks, make an `interface` for them.

Code

```
public class Callbacker implements Runnable {
    private Thing forDoing;
    private ToCallback toCall;
    public Callbacker(Thing forDoing, ToCallback toCall) {
        this.forDoing = forDoing;
        this.toCall = toCall;
    }
    @Override public void run() {
        Info forYou = myRaisonDEtre(forDoing);
        toCall.callbackHandler(forYou);
    }
}

public class ToCallback {
    Thing something;
    public ToCallback(Thing something) {
        this.something = something;
    }
    public static void main(String[] args) {
        ToCallback instanceOfMe = new ToCallback(someThing);
        instanceOfMe.startThreadToDoThings();
    }
    public void startThreadToDoThings() {
        Callbacker cb = new Callbacker(something, this);
        new Thread(cb).start();
    }
    public void callbackHandler(Into receivedStuff) {
        doThingsWith(receivedStuff);
    }
}
```

Locks

See **package**

```
java.util.concurrent.locks
```

Interface lock

Use this when `synchronized` methods and blocks aren't going to cut it, e.g. if you want to use their associated `Condition` objects, or if you want something more sophisticated than "I wait for the lock, become sole owner, and release it when I'm done," e.g. the `ReadWriteLock` (see below), and the non- blocking `tryLock()` method (see interface methods).

Using a `synchronized` block with a `Lock` as a parameter just uses the `Lock` object's *associated* monitor just as usual. It is suggested not to do that because it's confusing.

We have (*none* are "optional"):

```
interface Lock {
    void lock()   // can't be interrupted while waiting, can cause d
    void lockInterruptibly()   // will wake up if interrupted while
    Condition newCondition()   // bound to this Lock instance
    boolean tryLock()          // only acquire if free

    /* e.g. myLock.tryLock(100, TimeUnit.MILLISECONDS);
     * this method can be interrupted while waiting  */
    boolean tryLock(long time, TimeUnit unit)  // give up after time

    void unlock()
}
```

"With this increased flexibility comes additional responsibility." In the docs, we are instructed to use the following idiom

```
Lock l = new ReentrantLock(); // could be *any* type of lock
l.lock();
try {
    // access protected resource
}
finally {       // IMPORTANT:
    l.unlock(); // still unlock even after Exception is thrown
}
```

Interface ReadWriteLock

> A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive. [-- Oracle]

Protect a concurrent object with this thing when there are many readers but few updates to it.

This is implemented in `class ReentrantReadWriteLock`

This interface does *not* `extend` anything.

```
interface ReadWriteLock {
    Lock readLock()     // return read-lock
    Lock writeLock()    // return write-lock
}
```

Class ReentrantLock

> A `ReentrantLock` has the same behavior & semantics as the *implicit monitor lock* invoked with the `synchronized` keyword, but has extended capabilities.

What makes it "reentrant" is that a single thread can repeatedly acquire a lock it already owns (which increases its `getHoldCount()`). If you do that, you must correspondingly `unlock()` over and over until your `hold count == 0`. This allows a single thread to acquire a lock, then call another method that also acquires that lock, without any issues.

If you construct a *fair* `ReentrantLock`, it will favor granting access to the longest-waiting thread. Otherwise, no particular access order is guaranteed. Don't make your lock `fair` unless you have a reason, because it makes your code much slower.

Note that the `Serializable` interface has no methods to bring to the table. Upon deserialization, a `Lock` will be *unlocked*, regardless of the state it was in when it was serialized.

```
class ReentrantLock implements Lock, Serializable {
    ReentrantLock(boolean fair)

    boolean isLocked()

    int     getHoldCount() // number of holds by current thread
    Thread  getOwner() // null if not owned

    Collection<Thread> getWaitingThreads(Condition) // waiting on C
    int getWaitQueueLength(Condition) // est. #threads waiting on C
    boolean hasWaiters(Condition)

    boolean hasQueuedThread(Thread) // is thread waiting to acquire
    boolean hasQueuedThreads()      // check if *anyone* is waiting

    boolean isHeldByCurrentThread()

    boolean isFair() // is fairness == true?
}
```

Interface Condition

- Use a `Condition` to make one thread wait for another thread's signal. The `Condition` variable facilitates this being done *atomically*. A `Condition` instance is *bound* to a `Lock`. You create the condition via

```
Condition c = myLock.newCondition()
```

- You might do this when you have acquired a lock, but still must wait before you can do useful work.
- You can only call `await()`, `signal()`, or `signalAll()` if you *have* the `Lock` that the `Condition` is attached to.
- When you call `await()` you relinquish the lock, and your `Thread` enters the *wait* state (see above). It is *not* made *runnable* again when the lock becomes available. It is *only* made runnable when someone calls `myCondition.signal()` and you are next-in-line on the `Condition`, or someone calls `myCondition.signalAll()`. At this point you can take the lock back and continue where you left off (returning from your call to `await()`). In general the code for this really ought to take the form

```
while(!(ok to proceed))
    condition.await();
```

    - This is because just because you got the lock back doesn't mean that whatever reason you stopped in the first place is not true anymore so you'll want to check again and keep blocking if the condition is still not met. This code does exactly that.
- If no one *ever* signals you, you can *deadlock*.

The interface looks like:

```
interface Condition {

    /* wait for another thread to call signal() on this Condition */
    void await()       // wait for Condition.signal() or Thread.inte
    void awaitUninterruptibly()  // wait for signal() but not interr

    /* wait until timeout */
    boolean await(long, TimeUnit)
    long    awaitNanos(long)
    void    awaitUntil(Date)

    void signal()               // wakeup one waiting thread
    void signalAll()            // wakeup *all* waiting threads
}
```

Synchronized

- Synchronization prevents *race conditions* (threads stepping on each other's toes, accessing corrupt shared data).
- Unless told otherwise (using a `synchronized` block or `volatile`), threads may work on locally cached copies of variables (e.g. `count`), updating the "main" copy when it suits them. For the reasons having to do with processor architecture, they may also *re-order reads and writes*, so a variable may not actually get updated when otherwise expected. However, *on entry to and exit from* blocks `synchronized` on a particular object, the entering/exiting thread also effectively *synchronizes* copies of all variables with "main memory" (aka. the Java heap, as seen by the JVM)

Block vs Method

A `synchronized` method

```
public synchronized void blah() {
  // do stuff
}
```

is semantically equivalent to a `synchronized (this)` block

```
public void blah() {
  synchronized (this) {
    // do stuff
  }
}
```

is semantically equivalent to using the *intrinsic lock* itself

```
public void blah() {
    this.intrinsicLock.lock();
    try {
        // do stuff
    }
    finally {
        this.intrinsicLock.unlock();
    }
}
```

We can wait on the `intrinsicLock`'s `intrinsicCondition` using `this.wait()` i.e. simply `wait()`, and `notifyAll()`.

synchronized(objInstance) block

- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock

synchronized method

Two Effects:

1. Only one thread can execute it at a time
2. Exiting the method establishes a happens-before relationship with subsequent invocations for the same object

Refs

1. Synchronized overview

Multi-*Process* Programming

Based on: `ProcessBuilder`'s Java Docs

1. Use `class ProcessBuilder` to create a new process in the OS
2. This is a child-process (in OS terms)
3. By default it inherits the current process's environment (accessible for itself via `System.getEnv()`)

4. The working dir defaults to same as current process, but is modifiable (see e.g. below)
5. By default, `STD|IN|OUT|ERR` are redirected to the current process, and can be accessed via streams obtained using methods `get(Output|Input|Error)Stream()`
6. The `Process` object does *not* get killed when there are no references to it left, it just keeps executing
7. You can wait until the process terminates with `proc.waitFor()`
8. To kill the `Process` via its reference: `proc.destroy()`

Basic usage pattern

```
/* create the ProcessBuilder as you'd expect */
ProcessBuilder procBldr = new ProcessBuilder("myCmd", "arg1", "arg2'

/* modify environment variables */
Map<String, String> env = procBldr.environment();
env.put("VAR1", "value");

/* change working dir */
procBldr.directory(new File("a_Location/inner"));

/* redirect IO streams (makes them inaccessible to current proccess
procBldr.redirectOutput(...);
procBldr.redirectErrorStream(...);

// make all IO locations the same as the current process
procBldr.inheritIO();

/* to capture the output you can EITHER do */

// this
procBldr.redirectOutput(ProcessBuilder.Redirect.INHERIT);

// or this for finer control and piping etc.
InputStream is = p.getInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(is)
String s;
while ((s = reader.readLine()) != null)
    System.out.println(s);
is.close();

/* start the process */
Process proc = procBldr.start();
```

Example of printing the output of "ls"

Based on Art of Coding (and verified to work)

```
ProcessBuilder procBldr = new ProcessBuilder("ls");
Process p = procBldr.start();
p.waitFor();
System.out.println("Process exited with code = "+p.exitValue());

// To send STDERR to STDOUT
procBldr.redirectErrorStream(true);

// this can be simplified, see my notes above
InputStream is = p.getInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(is)
String s;
while ((s = reader.readLine()) != null)
    System.out.println(s);
is.close();
```

Other

- Constructors cannot be `synchronized`
- Every object has an intrinsic lock associated with it

Java Memory Model

- On modern platforms, code is frequently reordered by the compiler, the processor and the memory subsystem to achieve maximum performance
- **The Java Memory Model (JMM) defines** *when such reorderings are possible*
    - Execution-time constraints on the relationship between threads and main memory to achieve consistent and reliable applications
    - Makes it possible to reason about code execution in the face of optimizations
- JVMs must observe **within-thread as-if-serial** semantics
    - NB: *as-if-serial* semantics do *not* prevent different threads from having different views of the data.
    - It *does* mean that everything that happens before the release of a lock will be seen to be ordered before and visible to everything that happens after a subsequent acquisition of that same lock.

Cache coherence:

- After we exit a `synchronized` block, we *release* the monitor
- This flushes the cache to main memory, making writes made by this thread visible to other threads.
- So now, before we can enter a `synchronized` block, we *acquire* the monitor, invalidating the local processor cache, so that variables will be reloaded from

main memory, so are now able to see all of the writes made by the previous release.

Volatile Fields

- Indicates that a variable's value will be modified by different threads.
- Lock-free mechanism for synchronizing access to an instance field
- Declaring a `volatile` Java variable means:
  - The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory"
  - This shows what it does

    ```
    volatile int i;
    ...
    i += 5;
    ```

    Is basically equivalent to:

    ```
    // Note that we can't literally synchronize on an int p
    int temp;
    synchronized (i) { temp = i; }
    temp += 5;
    synchronized (i) { i = temp; }
    ```

    Note! Although it may look it, **the operation i += 5 is *not atomic***. If that's what you want, you should probably use `AtomicInteger` instead.
- The value of a `volatile` field becomes *visible to all **readers*** (other threads in particular, aka *consistent*) *after a **write** operation completes* on it
  - Without `volatile`, readers could see some non-updated value

Category	`synchronized(thing)`	`volatile`
works on **primitive** types	no	yes
can **block**	yes	no (no lock involved)
can be `null`	no	yes
provides **atomicity**	no	no

Refs

- Volatile Tutorial
- Jamex's Common Volatile Bugs

Final variables

- Basically a `const` in C++
- Remember the whole double-checked locking thing? Well, apparently, "`final` can be used to make sure that when you construct an object, another thread accessing that object doesn't see that object in a partially-constructed state, as could otherwise happen"

Refs

- Final Tutorial

Atomics

- `AtomicInteger` `AtomicLong` `AtomicIntegerArray` `AtomicReferenceArray`
- In package `java.util.concurrent.atomic`, they have atomic methods `in/decrementAndGet()`, meaning you can safely use e.g. `AtomicInteger` as a *shared counter* without any synchronization.

Concurrent Data Structures

You want to stay away from the above low-level constructs whenever posssible, and use higher-level structures implemented by concurrency experts with a lot of time to sink into it and experience to debug it. Queues are often the right choice for multithreading scenarios; you have producer(s) that insert in and consumer(s) that retrieve out. *Blocking queues* block when you try to insert into a full queue or remove from an empty one, until that operation because feasible.

```
interface BlockingQueue<E> {

    /* insert */
    void add()      // Exception if it's full
    bool offer()    // block while full, false on timeout
    void put()      // block while full, no timeout

    /* return head element */
    E element()     // Exception if empty
    E peek()        // null if empty

    /* remove and return head element */
    E remove        // Exception if empty
    E take()        // block while empty
}
```

You'll also find the following implementations:

Queues

```
ArrayBlockingQueue  // fixed max size on creation
LinkedBlockingQueue // no fixed upper bound on size
PrioirityBlockingQueue // removed by priority, unbounded size
LinkedTransferQueue // SE 7, producer's insert blocks till consumer
```

Other

```
ConcurrentHashMap       // shared cache, ≤ 16 *simult* writers by de
ConcurrentSkipListMap
ConcurrentSkipListSet
ConcurrentLinkedQueue

CopyOnWriteArrayList
CopyOnWriteArraySet
```

ConcurrentHashMap

Feature	`Collections.synchronizedMap(Map)`	ConcurrentHashMap
Allows multiple concurrent readers	yes	yes
Must synchronize while iterating through	yes	no
Allows (tunable) multiple concurrent writers	no	yes

- In cases in which multiple threads are expected to access a common collection, "Concurrent" versions are normally preferable.
- Updates to these things are **not *atomic***, but we can provide atomicity in three simple ways
    - Use the `putIfAbsent(K,V)` method
    - `synchronize` the whole updating method

- Use the `replace()` method like a sort of *compare-and-swap* like the following real-live piece of "optimistic concurrency via compare-and-swap"

```
private void incrementCount(String q) {
Integer oldVal, newVal;
    do {
      oldVal = queryCounts.get(q);
      newVal = (oldVal == null) ? 1 : (oldVal + 1);
    } while (!queryCounts.replace(q, oldVal, newVal));
}
```

How does it work?

- Has a field `Segment[]` whose size is the 3rd constructor parameter ("`concurrencyLevel`")
  - Each `Segment` contains a `HashEntry<K,V>[] hashTable` whose size is \(\frac{\mathrm{totalSize}}{(1^{st}param)}}{\mathrm{numSegments}}{(3^{rd}param)}}\)
- When you go to write it, you *first* hash into the correct `Segment` and *only lock that one*

Refs

1. Java2Blog Tutorial
2. Oracle's Package Concurrent Summary
3. ConcurrentHashMaps on Jamex

Collections.synchronized(Collection|List|Map|Set|SortedMap|SortedSet)()

- See table nearby for comparison with `ConcurrentHashMap`
- Wrapping a `Map` with Collections.synchronizedMap(myMap) makes it *safe* to access the map concurrently: each call to `get()`, `put()`, `size()`, `containsKey()` etc will synchronize on the map during the call
- Note that **this doesn't make updates to the `Map` *atomic***, but it does make it *safe*. That is, concurrent calls to update will never leave the `Map` in a *corrupted state*. But they might 'miss a count' from time to time.
  - For example, two threads could concurrently read a current value of, say, 2 for a particular query, both independently increment it to 3, and both set it to 3, when in fact two queries have been made.
  - The same is true of the [above] `ConcurrentHashMap`, though we can fix that (above)

Refs

1. ConcurrentHashMaps Jamex

Callables, Futures, Thread Pools

- `Runnable` --- runs a task asynchronously, no params, no return value
    - You implmement `public void run()` which gets called via `new Thread(runnable).start()`
- `Callable<V>` --- runs a task asynchronously, no params, with return value
    - You implement `public V call()`
- `Future<V>` --- holds result of asynchronous computation
    - no callbacks
    - blocking `V get()`
    - non-blocking `boolean isCancelled()`

Thread Pools

- Use these if your program use a large number of short-lived threads.
- When a thread dies, it gets the next task instead of getting deallocated & reallocated.
- `Executors` --- contains *static factory methods* for constructing thread pools

```
newFixedThreadPool(size)    // fixed size pool
newCachedThreadPool()       // creates new threads when nece
newScheduledThreadPool(nThreads)    // run periodically / af
newSingleThreadScheduledExecutor()  // see above
```

    - `Future<T> submit(Callable<T>/Runnable[, T result])` --- submit to pool
        - `Callable` is the easiest way to get the return value. Just call `get()` on the `Future<T>`
    - `shutdown()` --- shuts down pool nicely (call when you're done so you program can terminate)
    - `shutdownNow()` --- cancels all current tasks and shuts down

Other

There's a lot more advanced stuff in here that I will not take notes on

- Controlling groups of tasks with `ExecutorCompletionService`
- The fork-join framework, allowing `RecursiveTask<T>`,\
- Synchronizers --- pre-implemented patterns for common multithread synchronization patterns, e.g. `Exchanger`, `CyclicBarrier` (neural nets?), and

Semaphore