Concurrency

Synchronized Block

Synchronized Access

- atomic classes protect single variable
- synchronized access protects series of commands (block)
- a structure called *monitor* (or *lock*) supports *mutual exclusion*
    - while the block is running, no other thread can interfere
- any object can be used as a monitor (existing or new one)
- when thread tries to run the block it first checks if any other thread is running it
    - if lock is not available, the thread will transition to `BLOCKED` state
    - after the thread "acquires the lock", the single thread will enter the block
    - while the block is executed all other threads will be prevented from entering

```
// synchronized block (syntax)

var lock = new Object();

synchronized(lock) {

  // code which needs to be executed

  // one thread at a time

}
```

lock can be any `Object` (existing or newly created)

synchronized block

```
// synchronized methods
// first way
void doSomething() {
  synchronized(this) {  current class (this) is used as a lock
    // work to be executed one thread at a time
  }
}


// alternative
syncrhonized void doSomething() {       method is marked as synchronized
   // work to be executed one thread at a time
}
```

ReentrantLock

- part of `Lock` interface which allows manual control over monitors
- for example, it's useful when we want to check if lock is available
  - and then maybe do something else in case it's not
- to protect a part of code* call `lock()` method

  *to make it unavailable to other threads while one thread is using it
- to make ti available to other threads call `unlock()` method

```
// using ReentrantLock
Lock myLock = new ReentrantLock();
try {
   myLock.lock();
   // work to be executed one thread at a time
} finally {
   myLock.unlock();
}

// this is equivalent to using synchronized block,
// but it gives you more control over the access
```

creating an instance of Lock

Lock Methods

Method	**Description**
`void lock()`	Requires lock and blocks until lock is acquired
`void unlock()`	Releases a lock
`boolean tryLock()`	Requests lock an returns immediately, returns `boolean` indicating if the lock was successfully acquired

`boolean tryLock(`
`  long Timeout,`
`  TimeUnit unit)`
	Requests lock and blocks for specified time or until lock is acquired,
	returned `boolean` indicating if the lock was successfully acquired

Keep in mind...

- you can release the lock the same number of times it is acquired
  - in other words lock/unlock always work in pairs
- if you try to obtain the lock twice, but release it only once, you'll create an error
- to make sure to avoid this error use `tryLock()` in combination with `unlock()`
  - only if `tryLock()` returns `true`, call `unlock()`