# Threads

In Java, we can interact with threads in 2 ways.

1. Create an object that implements Runnable Interface.
   1. Object needs to implement public run() method.
   2. When then pass this object to the c’tor of thread. (i.e. new Thread(myRunnableObj))
2. Create an object that extends the Thread class.
   1. Need to override the run() method.
   2. Just need to call the new object c’tor directly (i.e. new MyThreadExtendedObj())

What are the pros and cons?

1. Java does not support multiple inheritance, therefore extending a thread class means the subclass cannot extend any other class. Implmenting Runnable interface means the class is still extendable.
2. A class might only be interested in being runnable, so inheriting the full Thread impl may be wasteful.

# Synchronization

Synchronize proves a locking mechanism around objects and methods. A method that is synchronized will block concurrent access for the same object instance. However, if they are separate instances, concurrent threads may run on the method. Locking is only for the instance of the object.

Static method synchronize on the class lock. Two threads could not simultaneously execute synchronized static methods on the same class even if one is calling foo and the other is calling bar.

## Locks

For more granular control, we can utilize a lock. A lock (or monitor) is used to synchronize access to a shared resource by associating the resource with a lock. A thread gets access to a shared resource by first acquiring the lock associated with the resource.

**Public class LockedATM {**

**Private Lock lock;**

**Private int balance = 100;**

**Public LockedATM() {**

**Lock = new ReentrantLock();**

**}**

**Public int withdraw(int value) {**

**Lock.lock();**

**// do something**

**Lock.unlock();**

**)**

Generally, using synchronized vs Locks are pretty comparable. The JVM can optimize synchronized, and if used with fewer threads, can often be more performant. However Lock provides extended tryLock API’s if we want to react to locking contention.

# NETWORKING

What happens behind the scenes when I type in a URL into a web browser?

In an extremely rough and simplified sketch, assuming the simplest possible HTTP request, no proxies and IPv4 (this would work similarly for IPv6-only client, but I have yet to see such workstation):

1. browser checks cache; if requested object is in cache and is fresh, skip to #9
2. browser asks OS for server's IP address
3. OS makes a DNS lookup and replies the IP address to the browser
4. browser opens a TCP connection to server (this step is much more complex with HTTPS)
5. browser sends the HTTP request through TCP connection
6. browser receives HTTP response and may close the TCP connection, or reuse it for another request
7. browser checks if the response is a redirect (3xx result status codes), authorization request (401), error (4xx and 5xx), etc.; these are handled differently from normal responses (2xx)
8. if cacheable, response is stored in cache
9. browser decodes response (e.g. if it's gzipped)
10. browser determines what to do with response (e.g. is it a HTML page, is it an image, is it a sound clip?)
11. browser renders response, or offers a download dialog for unrecognized types