**Producer/Consumer Model**

Imagine a model where you have one thread creating items and adding them to a pool and another thread removing things from the pool to use.

Assume also that the pool has a capacity.

How might you coordinate things so that the producer doesn’t overfill the pool and the consumer properly removes things once they are ready?

We’ve already seen something similar to this with the toy programs that used the bounded counter.
**Producer/Consumer Code (I)**

```java
private static final int THINGS_TO_PRODUCE = 100045;
private static final int THINGS_TO_CONSUME = 100030;
private static final int POOL_SIZE_LIMIT = 100;

private static final LinkedList<Integer> poolOfValues =
    new LinkedList<Integer>();
private static final Random rnd = new Random();

public static void main(String[] args) throws InterruptedException {
    Thread T1 = new Thread(new Runnable() {
        public void run() {System.out.println("Produced: "+ Producer());}
    });
    T1.start();

    Thread T2 = new Thread(new Runnable() {
        public void run() {System.out.println("Consumed: "+ Consumer());}
    });
    T2.start();
    T1.join(); T2.join();
    System.out.println("Left in the pool: "+ poolOfValues.size());
}
```

**Producer/Consumer Code (II)**

```java
private static int Producer() {
    int thingsProduced = 0;
    for (int i=0; i<THINGS_TO_PRODUCE; i++) {
        synchronized(poolOfValues) {
            while (poolOfValues.size() == POOL_SIZE_LIMIT) {
                try {poolOfValues.wait();} catch (InterruptedException e) {} 
            }
            poolOfValues.add(rnd.nextInt());
            thingsProduced++;
            poolOfValues.notifyAll();
        }
    }
    return thingsProduced;
}

private static int Consumer() {
    int thingsConsumed = 0;
    for (int i=0; i<THINGS_TO_CONSUME; i++) {
        synchronized(poolOfValues) {
            while (poolOfValues.size() == 0) {
                try {poolOfValues.wait();} catch (InterruptedException e) {} 
            }
            poolOfValues.remove();
            thingsConsumed++;
            poolOfValues.notifyAll();
        }
    }
    return thingsConsumed;
}
```
Semaphores

The previous example can be somewhat simplified and potentially made more efficient by the use of a Dijkstra-styled counting semaphore...

There are several ways to use these, but what they support is the ability to increase a counter (release()) and decrease a counter (acquire()) in such a way that attempting to decrease a counter that has reached 0 will block and increasing a counter will automatically wake a thread that is blocked on it (if one exists).

```java
public static void main(String[] args) throws InterruptedException {
    Thread T1 = new Thread(new Runnable() {
        public void run() {
            System.out.println("Produced: "+ Producer());
        }
    });
    T1.start();
    Thread T2 = new Thread(new Runnable() {
        public void run() {
            System.out.println("Consumed: "+ Consumer());
        }
    });
    T2.start();
    T1.join(); T2.join();
    System.out.println("Left in the pool: "+ poolOfValues.size());
}
```
Producer/Consumer w/Semaphores Code (II)

```java
private static int Producer() {
    int thingsProduced = 0;
    for (int i=0; i<THINGS_TO_PRODUCE; i++) {
        try {
            poolAvailableSpace.acquire();
            synchronized(poolOfValues) {
                poolOfValues.add(rnd.nextInt());
                poolAvailableValues.release();
            }
            thingsProduced++;
        } catch (InterruptedException e) {} /* Ignore interrupt */
    }
    return thingsProduced;
}

private static int Consumer() {
    int thingsConsumed = 0;
    for (int i=0; i<THINGS_TO_CONSUME; i++) {
        try {
            poolAvailableValues.acquire();
            synchronized(poolOfValues) {
                poolOfValues.remove();
                poolAvailableSpace.release();
            }
            thingsConsumed++;
        } catch (InterruptedException e) {} /* Ignore interrupt */
    }
    return thingsConsumed;
}
```

Borrow/Return via Semaphores

If you have a scenario that fits a borrow and return metaphor, a single Semaphore could be used. Set the Semaphore to the number of items available to borrow. When someone wants to borrow it, they do an .acquire() which will block until one is available. They have to .release() it so that others can borrow it.

Note that this is different from the pool scenario.
The CountDownLatch synchronizer

The use of a `CountDownLatch` allows you to prime several threads to begin but has them wait for a common start signal. The start signal can come from one place or have to come from multiple places.

It also allows you to effectively “join” on all of them in via a single statement in the original thread that launches the others.

```java
class SynchronizedStart {
    private static final int HOW_MANY_THREADS = 7;

    private static class DoWork implements Runnable {
        CountDownLatch starterDelay, joinerLatch;
        int myCode;

        DoWork(CountDownLatch startLatchIn, CountDownLatch stopLatchIn, int myCodeIn) {
            starterDelay = startLatchIn; joinerLatch = stopLatchIn; myCode = myCodeIn;
        }

        @Override
        public void run() {
            System.out.println("Preparing to run: " + myCode);
            try {
                starterDelay.await();
                System.out.println("Running: " + myCode);
                Thread.sleep(5000); //pretend to do some work
                joinerLatch.countDown();
            } catch (InterruptedException e) {
            }
        }
    }

    public static void main(String args[]) throws InterruptedException {
        CountDownLatch startSignal = new CountDownLatch(1);
        CountDownLatch stopSignal = new CountDownLatch(HOW_MANY_THREADS);
        for (int i=0; i<HOW_MANY_THREADS; i++) {
            new Thread(new DoWork(startSignal, stopSignal, i)).start();
        }
        System.out.println("Wait for it...");
        Thread.sleep(5000);
        System.out.println("Start!"); stopSignal.countDown();
        try { stopSignal.await(); } catch (InterruptedException e) {
        }
        System.out.println("Done");
    }
}
```
public class LaunchTest {
    private static final int GO_NOGO_PEOPLE = 15;

    private static class GoNoGoWorker implements Runnable {
        CountDownLatch launchLatch; int myCode;
        GoNoGoWorker(CountDownLatch launchLatchIn, int myCodeIn) {
            launchLatch = launchLatchIn; myCode = myCodeIn;
        }
        public void run() {
            try {
                Thread.sleep((new Random()).nextInt(5000));
                System.out.println("Go: " + myCode); launchLatch.countDown();
            } catch (InterruptedException e) {} } }
    public static void main(String args[]) throws InterruptedException {
        CountDownLatch launchDelay = new CountDownLatch(GO_NOGO_PEOPLE);
        for (int i = 0; i < GO_NOGO_PEOPLE; i++) {
            new Thread(new GoNoGoWorker(launchDelay, i)).start();
        }
        try {
            launchDelay.await();
            System.out.println("Go for launch!");
        } catch (InterruptedException ex) {} }
}

**Slow on-demand producers...**

What if your producer works on-demand but also takes some time to do its work and you don’t want consumers blocking on their requests to the producer since if they have multiple requests you want the producers to get started on all of them.

If you get them all started in different threads then they could potentially finish faster than if the requests were made in serial.

Imagine a producer that took some time to generate an integer (maybe due to computation, maybe due to a network-based request)…
**Callable<T>**
Similar to how we can use the `Runnable` interface to create a class that can then be “launched” in another thread, the `Callable` interface provides this ability with one difference – something that is callable will provide a return value.

**FutureTask<T> and Callable<T>**
The `FutureTask<T>` class is an interesting concept; provide the ability to start something running off in its own thread with a promise that when it ends in the future, you’ll know and be able to use some information as a result of it completing.
The main method

```java
ArrayList<FutureTask<Integer>> myValues = new ArrayList<FutureTask<Integer>>();

long startTime = System.currentTimeMillis();
for (int i=0; i<NUMBERS_NEEDED; i++) {
    myValues.add(new FutureTask<Integer>(){
        new Callable<Integer>() {
            public Integer call() {
                return producer();
            }
        }));
    new Thread(myValues.get(i)).start();
}

int sum = 0;
for (int i=0; i<NUMBERS_NEEDED; i++) {
    sum += myValues.get(i).get(); //blocks if return value not ready
}
long endTime = System.currentTimeMillis();
System.out.println("Elapsed time: " + (endTime-startTime));
```

FutureTask<T> and Runnable

If you have a fixed item that you’d like to return at a later date, or a reference to a mutable object, you can use Runnable rather than Callable.

```java
private class Sleeper implements Runnable {
    public void run() {
        //pretend like we're actually cooking...
        Thread.sleep(5000);
    }
}

public Future<Integer> generateInt() {
    FutureTask<Integer> intToReturn =
        new FutureTask<Integer>(new Sleeper(), 8);
    Thread intMaker = new Thread(intToReturn);
    intMaker.start();
    return intToReturn;
}

The caller of generateInt will perform .get() on return value, which unblocks when Integer is ready
```