THREAD POOLS
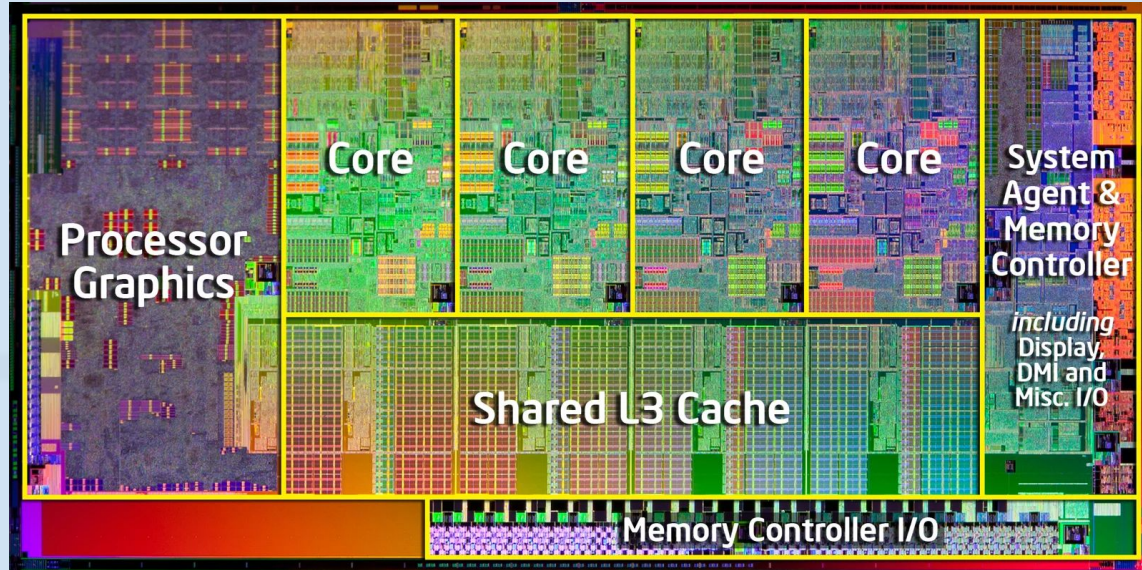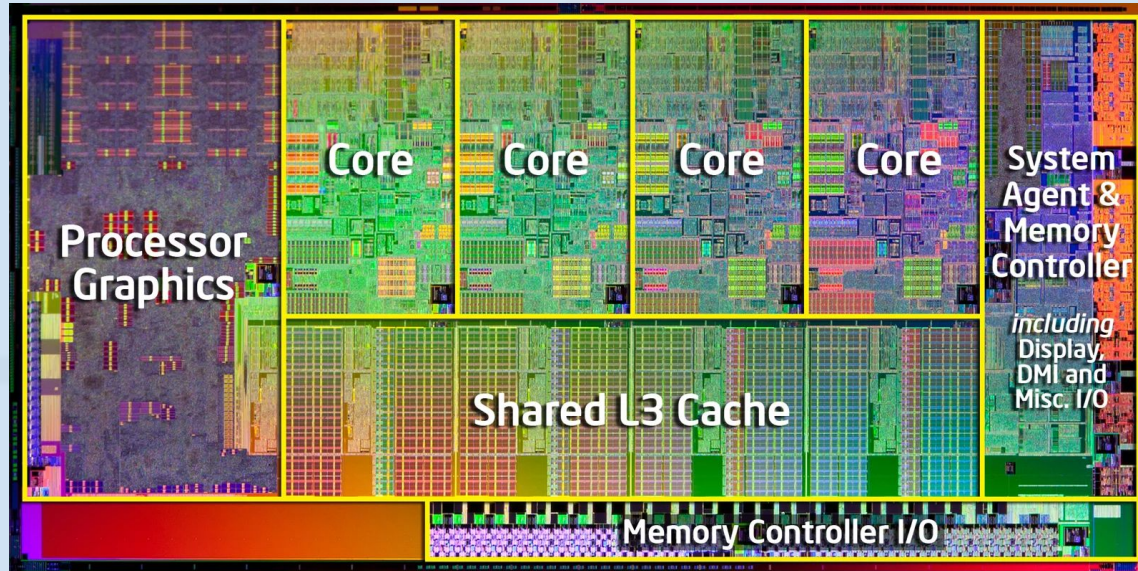
Now days, most computer processing units (CPUs) have multiple processors also called cores.

These processors can perform operations at the same time as the other processors to improve the efficiency of the CPU.

As computer programmers, we can use Threads to code the instructions we want the computer's cores to perform at the same time.

```
Thread t1 = new Thread(()->calculateStuff());
Thread t2 = new Thread(()->reconfigureThings());
Thread t3 = new Thread(()->modulateData());
Thread t4 = new Thread(()->processStructures());
Thread t5 = new Thread(()->computeComputations());
Thread t6 = new Thread(()->performComplexFucntion());
```

It would be nice and simple if we could create a new thread for every set of instructions that we want to happen concurrently.

```java
Thread t1  = new Thread(()->calculateStuff());
t1.start();
Thread t2  = new Thread(()->reconfigureThings());
t2.start();
Thread t3  = new Thread(()->modulateData());
t3.start();
Thread t4  = new Thread(()->processStructures());
t4.start();
Thread t5  = new Thread(()->computeComputations());
t5.start();
Thread t6  = new Thread(()->performComplexFucntion());
t6.start();
Thread t7  = new Thread(()->calculateStuff());
t7.start();
Thread t8  = new Thread(()->reconfigureThings());
t8.start();
Thread t9  = new Thread(()->modulateData());
t9.start();
Thread t10 = new Thread(()->processStructures());
t10.start();
Thread t11 = new Thread(()->computeComputations());
t11.start();
Thread t12 = new Thread(()->performComplexFucntion());
t12.start();
Thread t13 = new Thread(()->calculateStuff());
t13.start();
Thread t14 = new Thread(()->reconfigureThings());
t14.start();
Thread t15 = new Thread(()->modulateData());
t15.start();
Thread t16 = new Thread(()->processStructures());
```
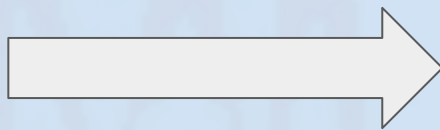
Unfortunately, this isn't always practical.

The process of getting the operating system to create a new thread is actually a relatively slow process, even if it may not be a whole lot of code.

```java
public static void main(String[] args) {
    Thread t = new Thread(()->{});
    t.start();
}
```

```asm
_Z12runPv:
.LFB11:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    %rdi, -8(%rbp)
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE11:
        .size   _Z12runPv, .-_Z12runPv
        .globl  main
        .type   main, @function
main:
.LFB12:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        leaq    -16(%rbp), %rax
        movl    $0, %ecx
        movl    $_Z12runPv, %edx
        movl    $0, %esi
        movq    %rax, %rdi
        call    pthread_create
        movl    $0, %eax
        movq    -8(%rbp), %rdx
        xorq    %fs:40, %rdx
        je      .L5
        call    __stack_chk_fail
```

And creating too many threads at one time will actually make the program slower or even cause it to crash.

!!!

`Exception in thread "main" java.lang.StackOverflowError` ■ ■ ■
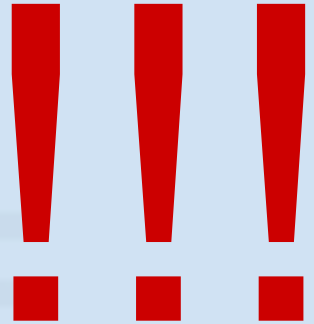
A better option might be to create a safe number of threads at the beginning of the program and then put them to sleep. This will be our pool of threads.

Thread 1	zzzzzzz
Thread 2	zzzzzzz
Thread 3	zzzzzzz
Thread 4	zzzzzzz
Thread 5	zzzzzzz
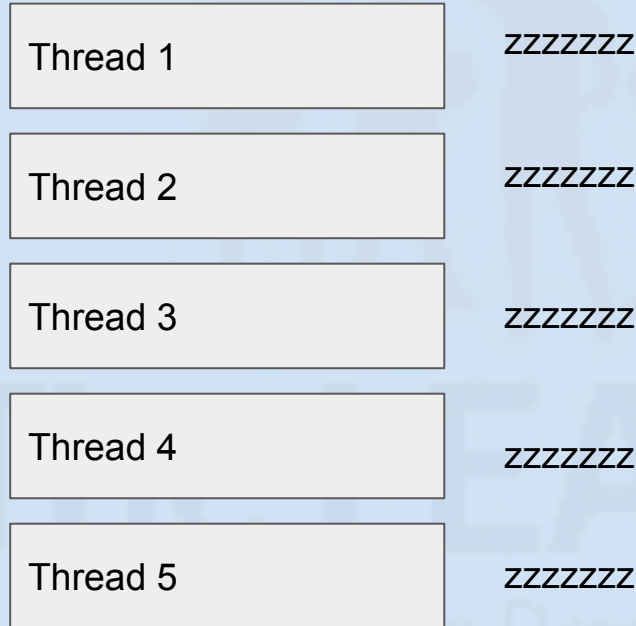
Then we store all the code we want to perform concurrently into a queue.

Job 1
Job 2
Job 3
Job 4
Job 5
Job 6
Job 7
Job 8
Job 9

When we want the threads to execute the code that was stored in the queue, we wake them up and have them perform jobs from the queue until the queue is empty.

| Thread 1 |
| Thread 2 |
| Thread 3 |
| Thread 4 |
| Thread 5 |

| Job 1 |
| Job 2 |
| Job 3 |
| Job 4 |
| Job 5 |
| Job 6 |
| Job 7 |
| Job 8 |
| Job 9 |
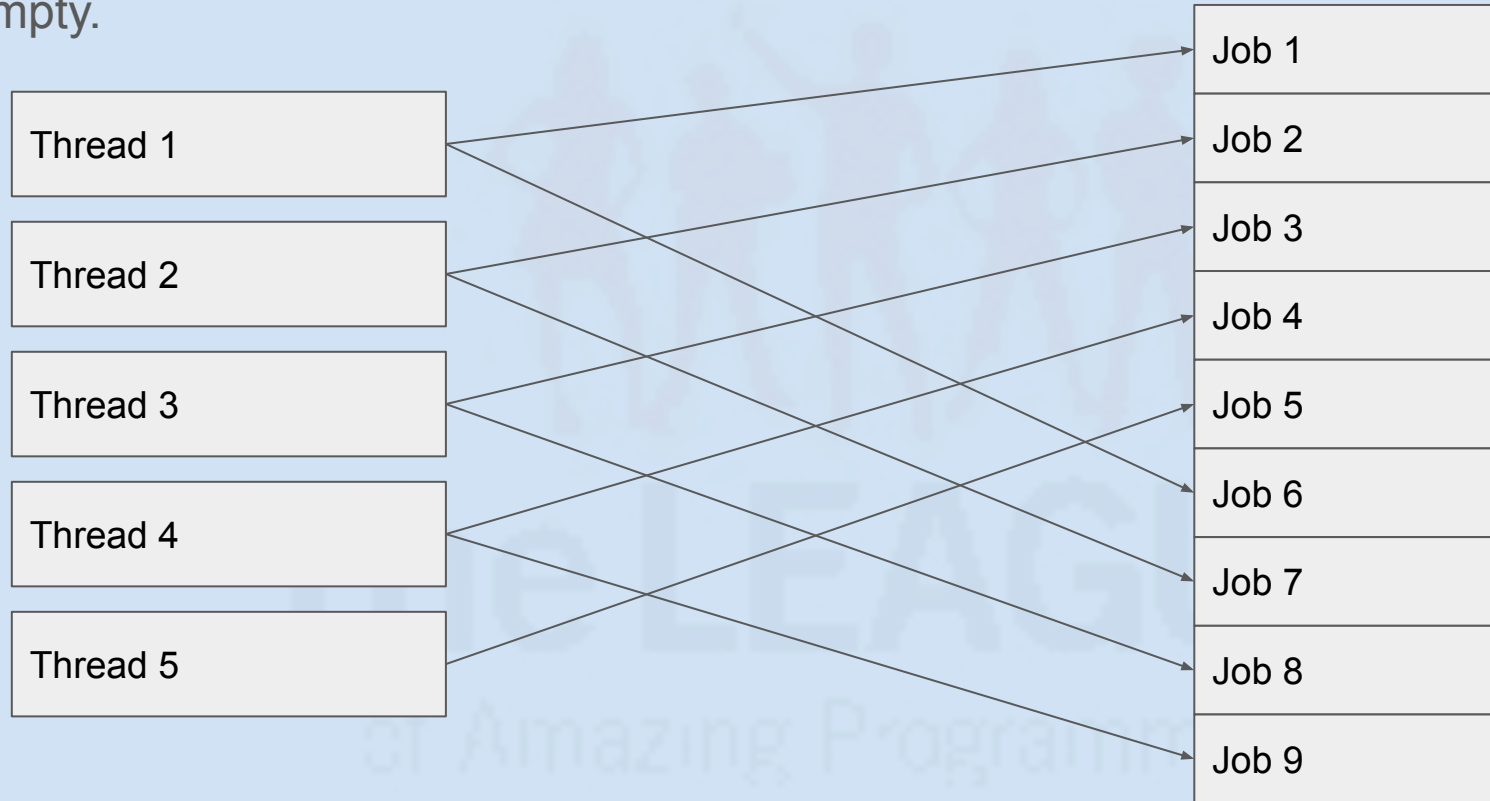
Then the threads go back to sleep.

Thread 1	zzzzzzz
Thread 2	zzzzzzz
Thread 3	zzzzzzz
Thread 4	zzzzzzz
Thread 5	zzzzzzz

The process of waking a sleeping thread is much faster than creating a new one. A thread pool shouldn't overload the operating system by accidentally creating too many threads.
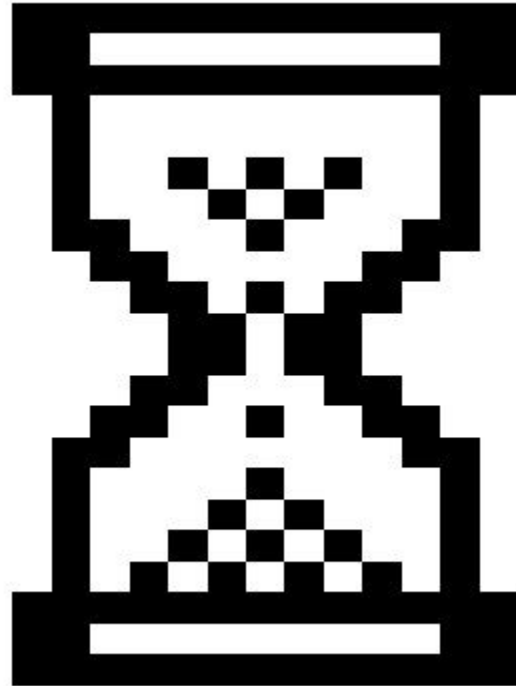
While the threads are asleep, the operating system can use the free processor power to perform any background operations that may need to be performed.

Background processes (66)					
adb (32 bit)	0%	2.0 MB	0 MB/s	0 Mbps	0%
Antimalware Service Executable	0.1%	195.9 MB	0 MB/s	0 Mbps	0%
Application Frame Host	0%	6.9 MB	0 MB/s	0 Mbps	0%
Calculator (2)	0%	0.9 MB	0 MB/s	0 Mbps	0%
COM Surrogate	0%	2.4 MB	0 MB/s	0 Mbps	0%
Component Package Support S...	0%	1.4 MB	0 MB/s	0 Mbps	0%
CTF Loader	0%	26.2 MB	0 MB/s	0 Mbps	0%
Device Association Framework ...	0%	5.3 MB	0 MB/s	0 Mbps	0%

Let's get started building our own thread pool in Java.

Create a new class called WorkQueue. This class will be our thread runner so be sure to implement the Runnable interface and add the *run* method.

```java
public class WorkQueue implements Runnable{
    public void run() {

    }
}
```

This class will need some private member variables. For now, we'll just make an array of threads. Don't initialize the array yet because we don't know how many threads to create.

```java
public class WorkQueue implements Runnable{
    private Thread[] threads;

    public void run() {

    }
}
```

How many threads should we use for our thread pool?

It really depends on the specific use of the program. However, a good starting point would be to create one thread for every core on the CPU minus one. The leftover processor will allow the main thread to add more work to the queue while the other threads are performing jobs.
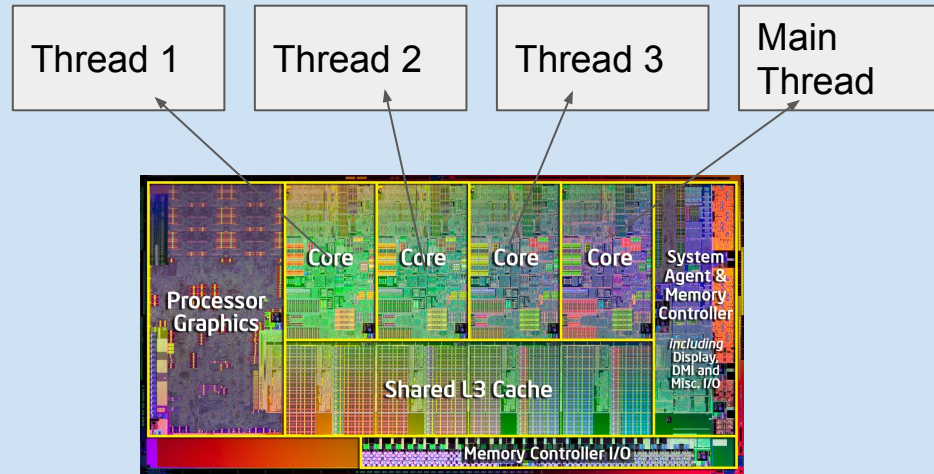
We can even use the main thread to help complete some of the jobs when it is no longer adding new ones to the queue.
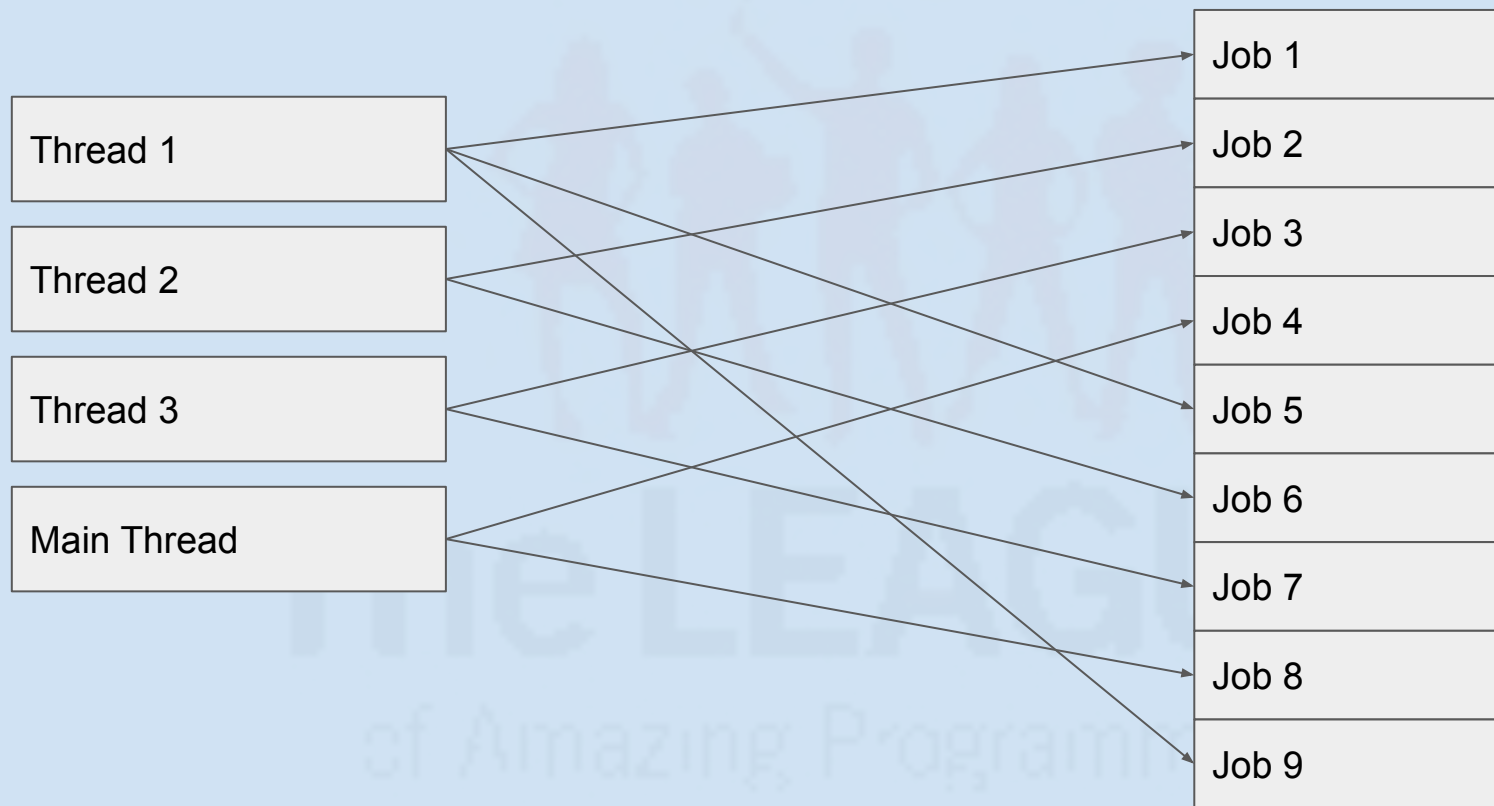
In Java, we can get the number of cores the CPU has with:

```
Runtime.getRuntime().availableProcessors()
```

Let's make a constructor for our WorkQueue class that will initialize our thread array with the proper number of threads.

```java
public class WorkQueue implements Runnable{
    private Thread[] threads;

    public WorkQueue() {
        int totalThreads = Runtime.getRuntime().availableProcessors() - 1;
        threads = new Thread[totalThreads];
    }

    public void run() {

    }
}
```

Make a method that returns the length of the thread array. This will come in handy later when we test our code.

```java
public int getThreadCount() {
    return threads.length;
}
```

After initializing the thread array, we can now go ahead and initialize each thread. We'll use **this** as the Runnable parameter.

```java
for(int i = 0; i < threads.length; i++) {
    threads[i] = new Thread(this);
}
```

We can also start the threads at this point, although it will have no effect since we haven't programmed the them to do anything yet.

```
threads[i].start();
```

We don't want our threads to terminate instantly, so we'll put a loop in the run method to keep them going until we want them to stop. In the **WorkQueue** class, create a private boolean member variable named *isRunning*. Initialize it to **true**.

```java
public class WorkQueue implements Runnable{
    private Thread[] threads;
    private boolean isRunning = true;

    public WorkQueue() {
        int totalThreads = Runtime getRuntir
```

Now let's put that loop in the run method so that it will repeat as long as *isRunning* is equal to **true**.

```java
public void run() {
    while(isRunning) {

    }
}
```

We need a way to shut down our threads so they don't loop forever. Create a public void method called **shutdown** that simply changes *isRunning* to equal **false**.

```java
public void shutdown() {
    isRunning = false;
}
```

For now, we'll put a print statement inside the while loop of the *run* method just to make sure our threads are working. You can get an identification number for whichever thread is currently operating with:

Thread.*currentThread*().getId()

```java
public void run() {
    while(isRunning) {
        System.out.println("Output from thread #" + Thread.currentThread().getId());
    }
}
```

It's time to test what we have so far. Create a runner class with a **main** method. In the **main** method, create an object of your **WorkQueue** class and use it to print the number of threads created and call the **shutdown** method.

```java
public class ThreadPoolTester {
    public static void main(String[] args) {
        WorkQueue wq = new WorkQueue();
        System.out.println("Total threads: " + wq.getThreadCount());
        wq.shutdown();
    }
}
```

Your output will probably look similar to this:

```
Output from thread #11
Output from thread #14
Total threads: 7
Output from thread #13
Output from thread #12
Output from thread #14
Output from thread #15
Output from thread #11
```

However, it will most likely not be exactly the same. It all depends on your computer's CPU, the operating system you are using, what other programs are running, and various other things. It is even possible that there won't be any output at all from the threads. As long as you don't get any errors, exceptions, or infinite loops; you are on the right track.

If you have that working, then it's time to start making it so that our main thread can provide work to our worker threads. For this, we'll create a new interface.

Job 1
Job 2
Job 3
Job 4
Job 5
Job 6
(empty)
(empty)
(empty)

Main Thread
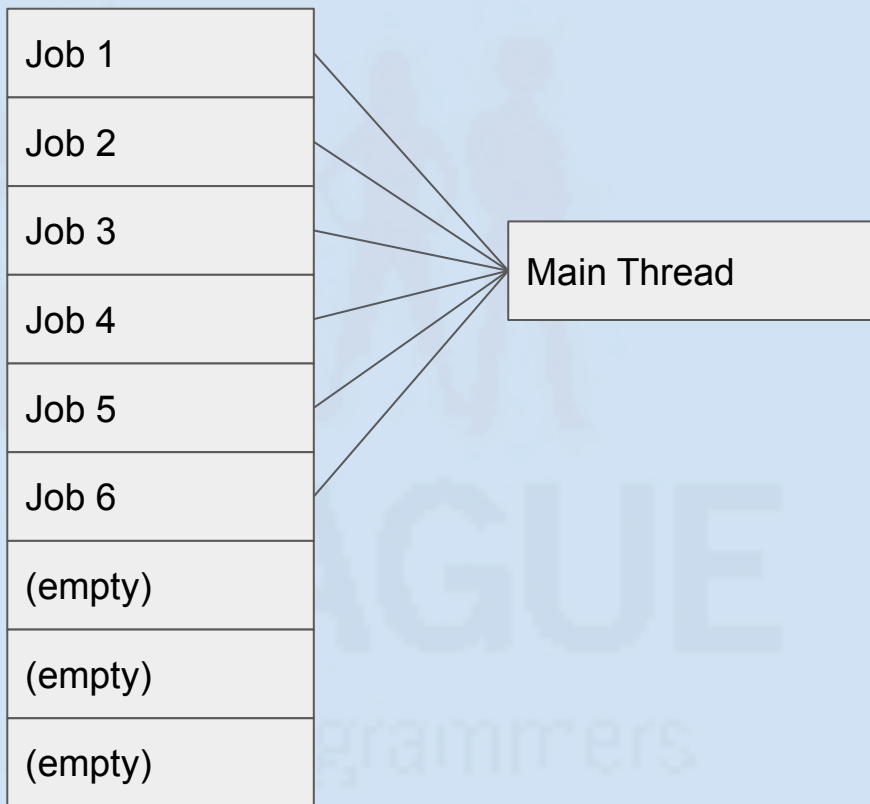
Create a new Interface called **Job**. This will be a functional interface so it will only have one abstract method. Call it **perform** and we won't have it take in any parameters for now.

```java
public interface Job {
    void perform();
}
```

In the **WorkQueue** class, we'll create and initialize an **ArrayDeque** of **Job** objects called *jobQueue*.

```java
public class WorkQueue implements Runnable{
    private ArrayDeque<Job> jobQueue = new ArrayDeque<Job>();
    private Thread[] threads;
```

Create a new method called addJob that takes in a Job object and adds it to the queue.

```java
public void addJob(Job j) {
    jobQueue.add(j);
}
```

Now that we are successfully creating our queue, we want to put the threads to sleep until they are needed to run some code for us.

Thread 1	zzzzzzz
Thread 2	zzzzzzz
Thread 3	zzzzzzz
Thread 4	zzzzzzz
Thread 5	zzzzzzz

How do we put the threads to sleep?

One option could be to use `Thread.sleep(milliseconds)`. But how would we know what value to put for `milliseconds`? We can't predict how long they will need to sleep.

Thread 1	zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz...
Thread 2	zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz...
Thread 3	zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz...
Thread 4	zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz...
Thread 5	zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz...

A better solution may be to use wait and notify. It kind of works like this:

Thread 1

Thread 2

We're going to be sleeping. Let us know when you want us to do something.

Thread 3

Thread 4

Thread 5

Main Thread

Okay!
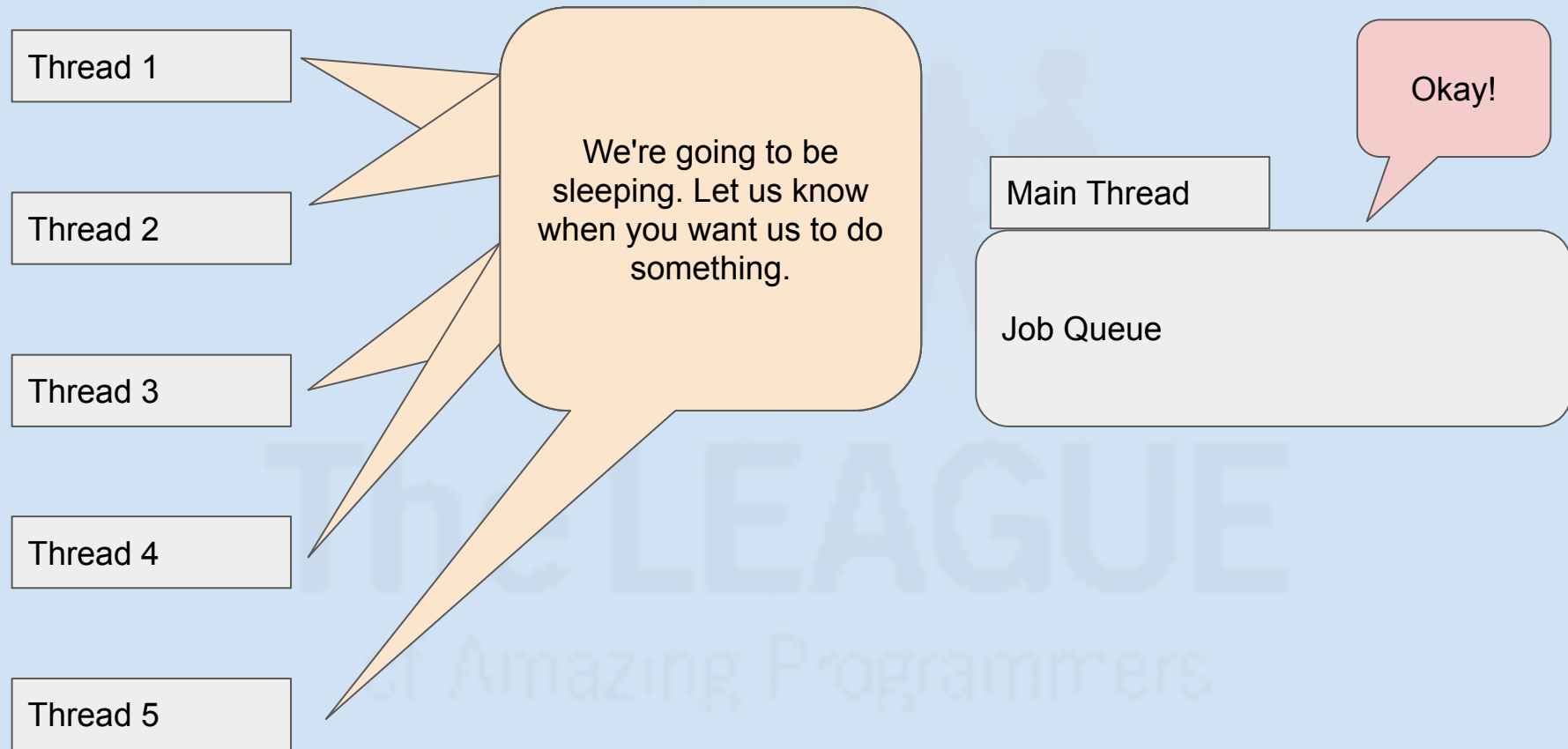
Job Queue

At this point, the main thread will add jobs to the queue and tell the queue how many threads to signal:

Thread 1   zzzzz...

Thread 2   zzzzz...

Thread 3   zzzzz...

Thread 4   zzzzz...

Thread 5   zzzzz...

Job 1

Job 2

Job 3

(empty)

(empty)

(empty)

(empty)

(empty)

(empty)

Hey! I need 3 threads.

Main Thread

Okay!
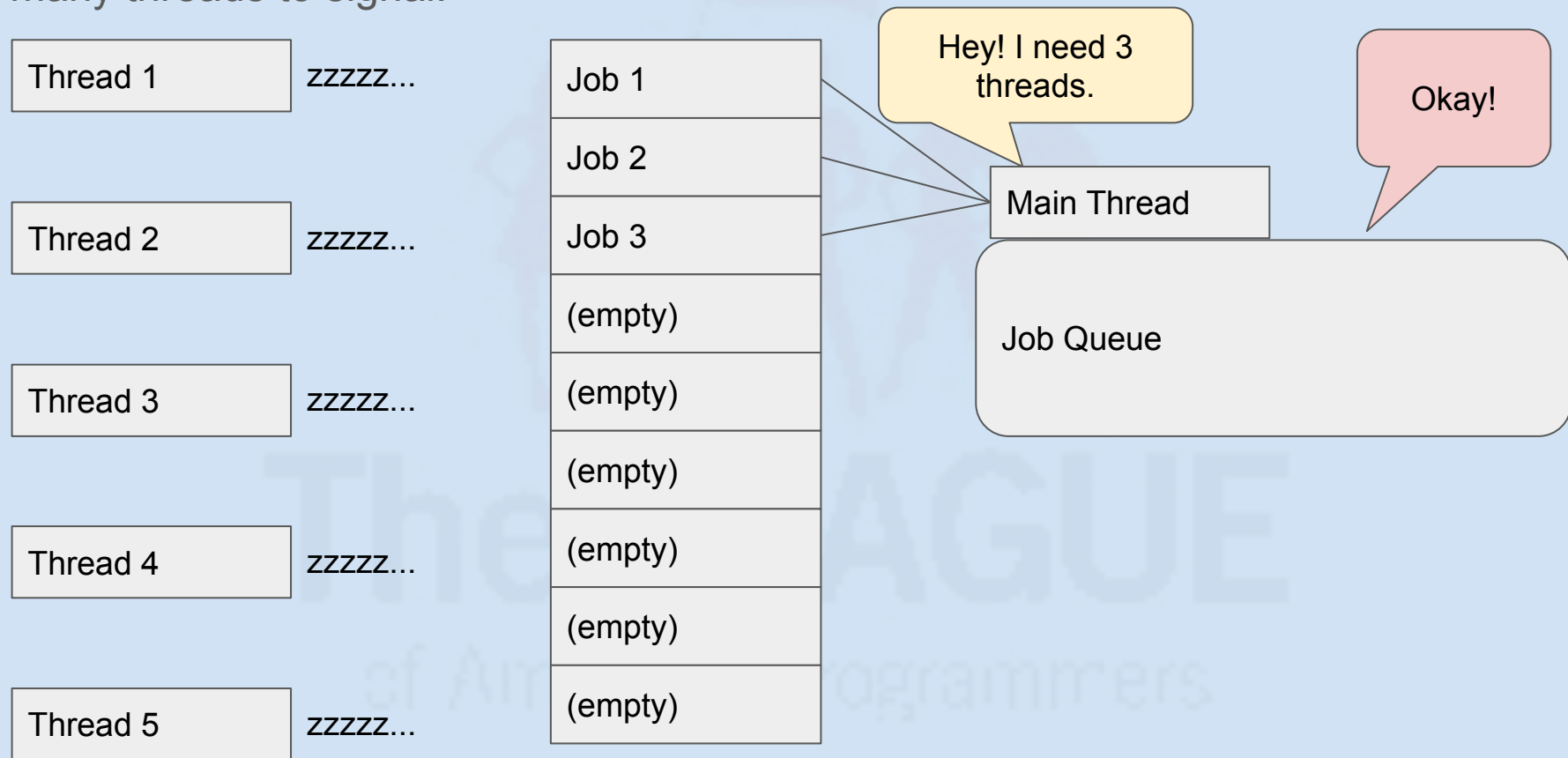
Job Queue

Then the queue will signal the sleeping threads:

Thread 1

Thread 2

Thread 3 zzzzz...

Thread 4 zzzzz...

Thread 5

We need three threads! Who's in?

I'll do it!

Main Thread
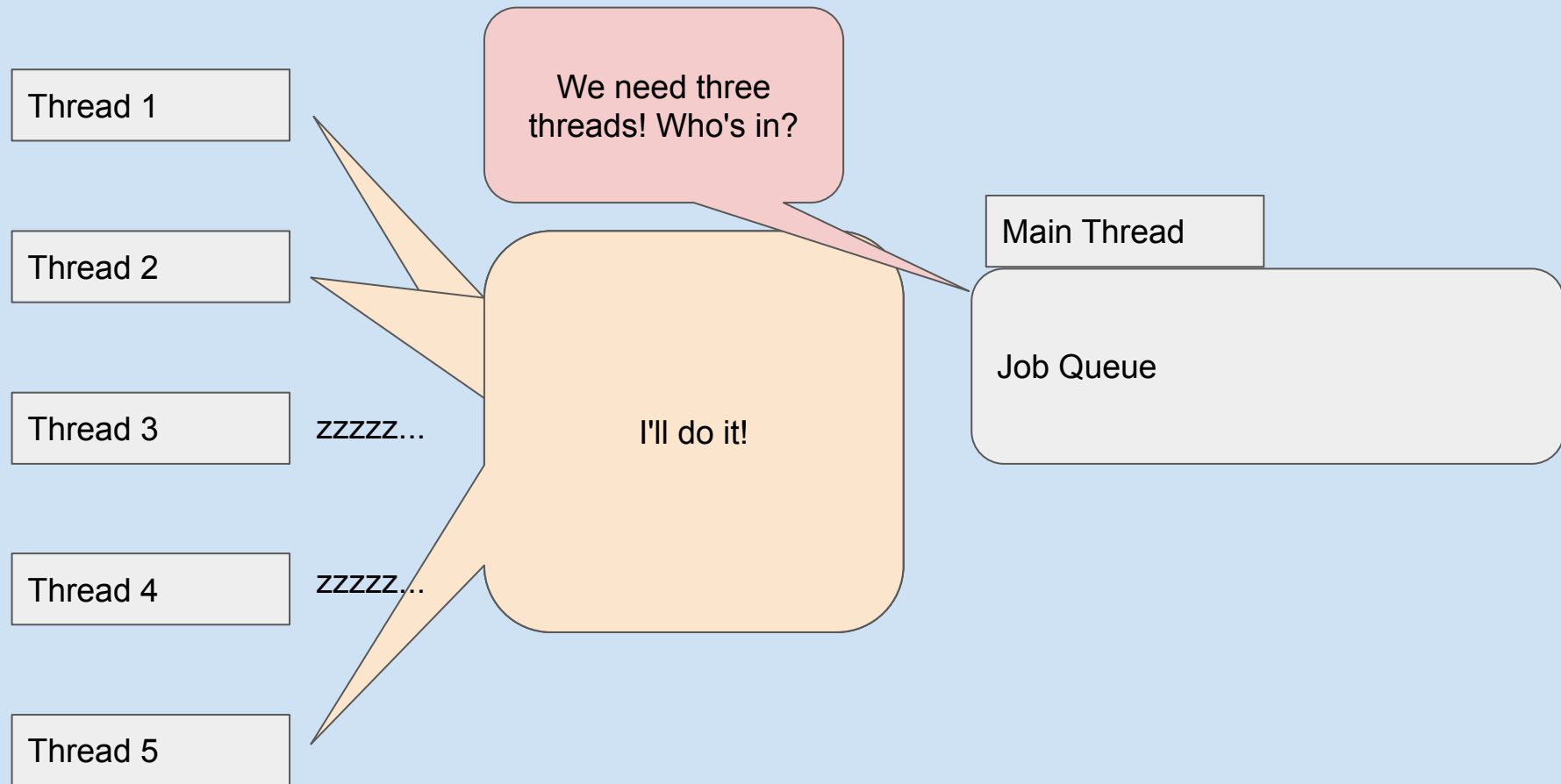
Job Queue

Let's now change the addJob method so that the job queue will be synchronized and notify a sleeping thread that a job is ready to be performed.

```java
public void addJob(Job j) {
    synchronized (jobQueue) {
        jobQueue.add(j);
        jobQueue.notify();
    }
}
```

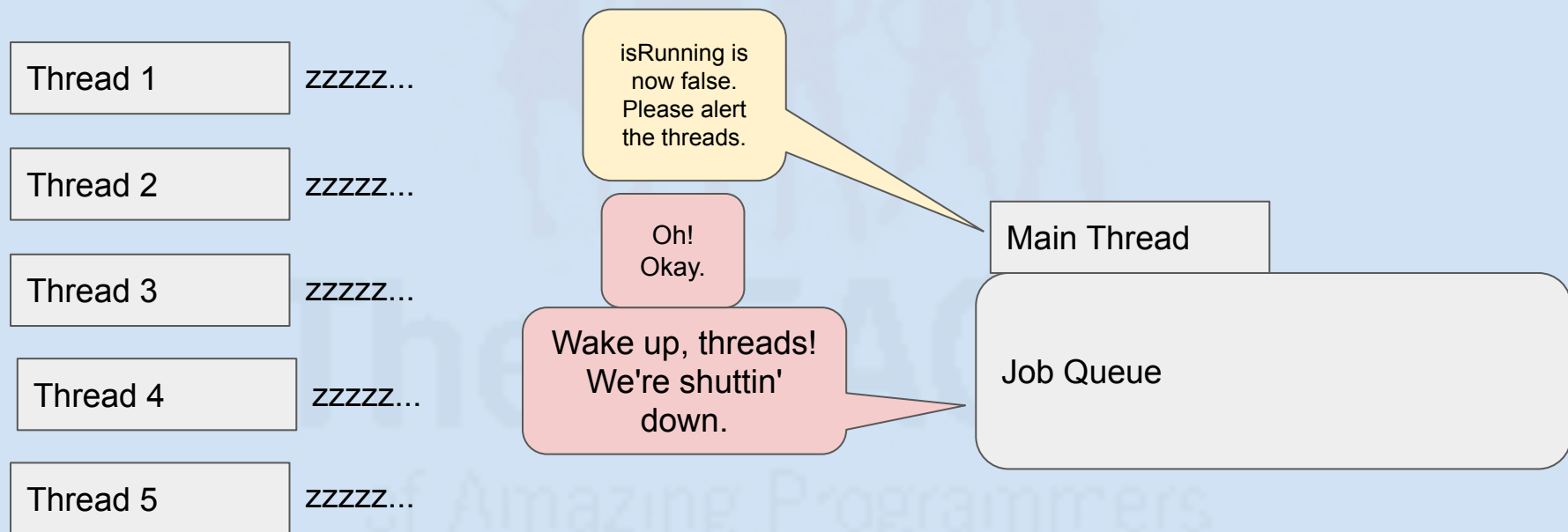Let's now modify the *run* method so that it will actually put the thread to sleep. After the print statement, synchronize the job queue and use it to call the *wait* method. You'll need to catch the **InteruptedException**.

```java
public void run() {
    while (isRunning) {
        System.out.println("Output from thread #" + Thread.currentThread().getId());
        synchronized (jobQueue) {
            try {
                jobQueue.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

The threads should now be going to sleep until the queue signals them to wake up. When we shut down our thread pool, we want to signal all of the threads to wake up so that they can shut down properly.

Thread 1   zzzzz...

Thread 2   zzzzz...

Thread 3   zzzzz...

Thread 4   zzzzz...

Thread 5   zzzzz...

isRunning is now false. Please alert the threads.

Main Thread

Oh! Okay.

Wake up, threads! We're shuttin' down.

Job Queue

In the shutdown method, synchronize the job queue and use it to call the *notifyAll()* method. This will tell all the waiting threads to wake up. They should notice that **isRunning** is now false and then terminate properly.

```java
public void shutdown() {
    isRunning = false;
    synchronized (jobQueue) {
        jobQueue.notifyAll();
    }
}
```

Since the *isRunning* variable is being read by multiple threads, add the keyword **volatile** to its declaration. Here is a link that details what the volatile keyword does: https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html

Or feel free to ask your teacher to tell you about it.

It is usually a good idea to mark any variable that will be accessed by multiple threads as **volatile**. In short, it will help prevent unpredictable outputs.

```java
private volatile boolean isRunning = true;
```

We're almost ready to test what we have so far. We just need to make sure that the main thread is giving the worker threads enough time to start. In your main method, add `Thread.sleep(1000)` before shutting down the work queue. Be sure to catch the **InterruptedException**.

```java
public static void main(String[] args) {
    WorkQueue wq = new WorkQueue();
    System.out.println("Total threads: " + wq.getThreadCount());
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    wq.shutdown();
}
```

Run your program. Your console should now have a print statement for every thread that was created. Each print statement should also have a unique ID. The order of the print statements does not matter.
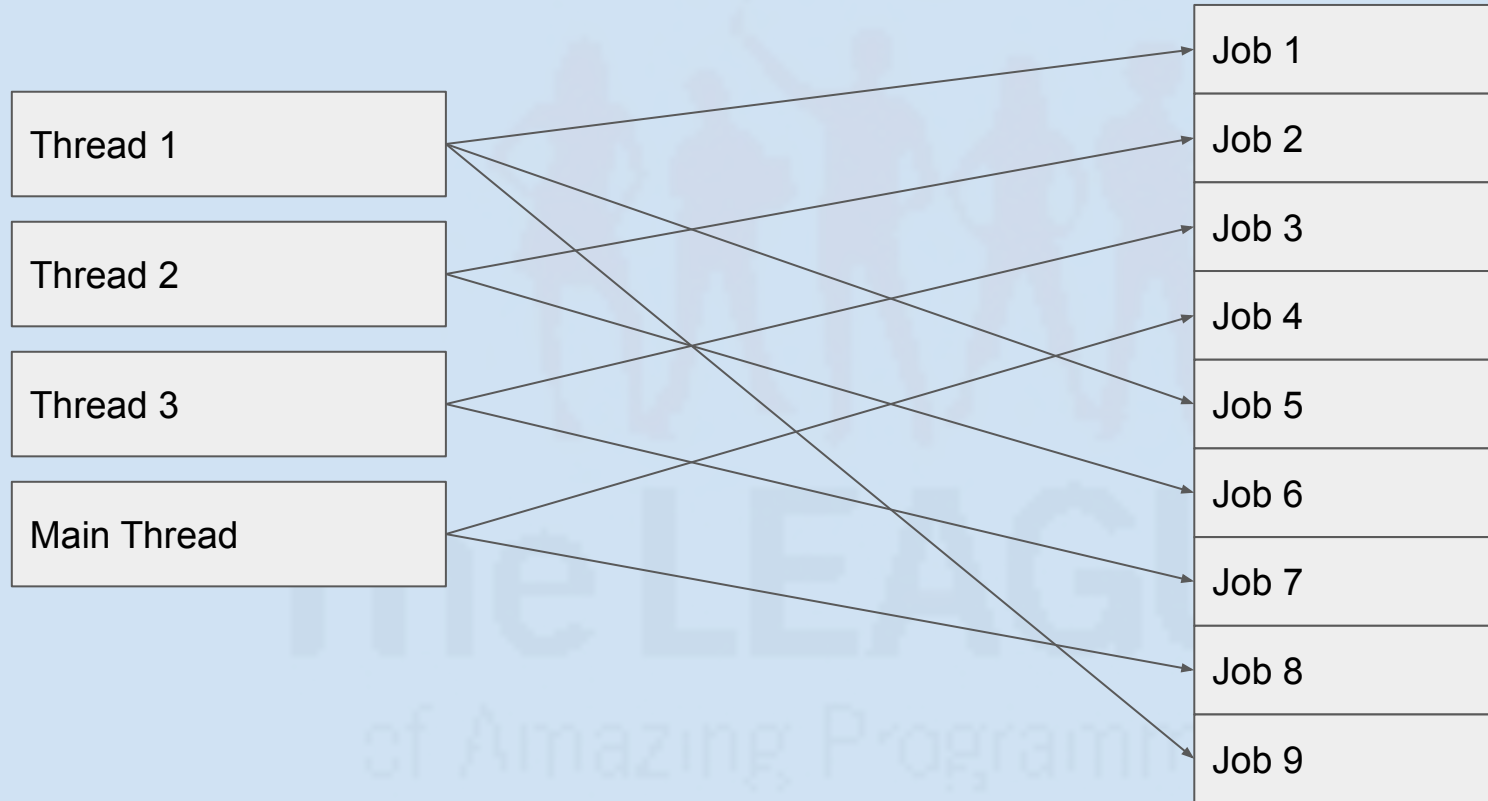
```
Total Threads: 7
Output from thread #12
Output from thread #13
Output from thread #14
Output from thread #11
Output from thread #15
Output from thread #16
Output from thread #17
```

Also, make sure that your program terminates after one second.
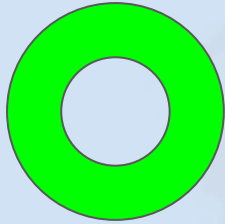
Now that we can add things to our queue, we need to be able to remove Jobs and perform their tasks.

Create a method in the WorkQueue class called *performJob* that returns a boolean.

```java
public boolean performJob() {
    return false;
}
```

If the *performJob* method successfully completes a job, then it'll return true. Otherwise, it will return false. The threads will use this to determine whether or not they should go to sleep. If a thread gets a job, then it will complete it and try to do another one. If it does not receive a job from the queue, then it will go to sleep.
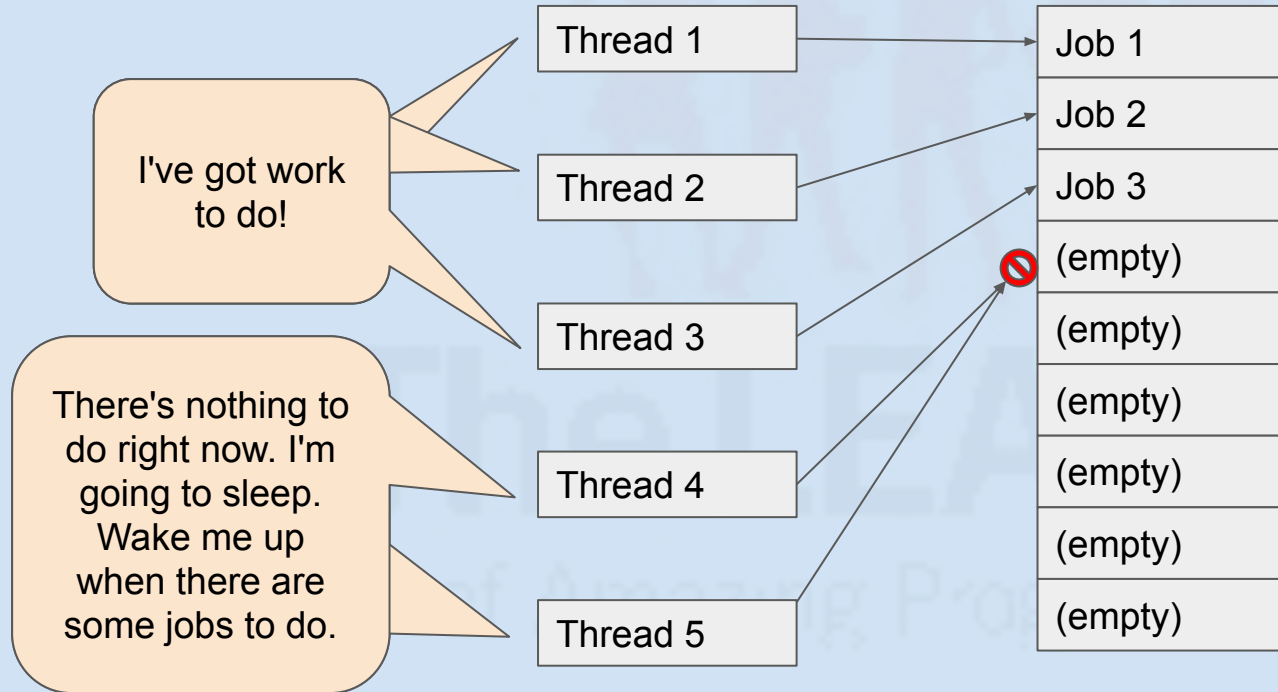
If the queue is not empty, then we want to remove a job from the queue, perform the job, and return true. Otherwise, return false.

```java
public boolean performJob() {
    if(!jobQueue.isEmpty()) {
        Job j = jobQueue.remove();
        j.perform();
        return true;
    }else {
        return false;
    }
}
```

While the code on the previous slide may work. There is a good chance that it may not. That is because that code is not "thread safe". If multiple threads run that method at the same time, it is possible that two or more threads will get the same job.

Thread 1

Thread 2	Uh oh!

Thread 3

Main Thread

Job 1
Job 2
Job 3
Job 4
Job 5
Job 6
Job 7
Job 8
Job 9
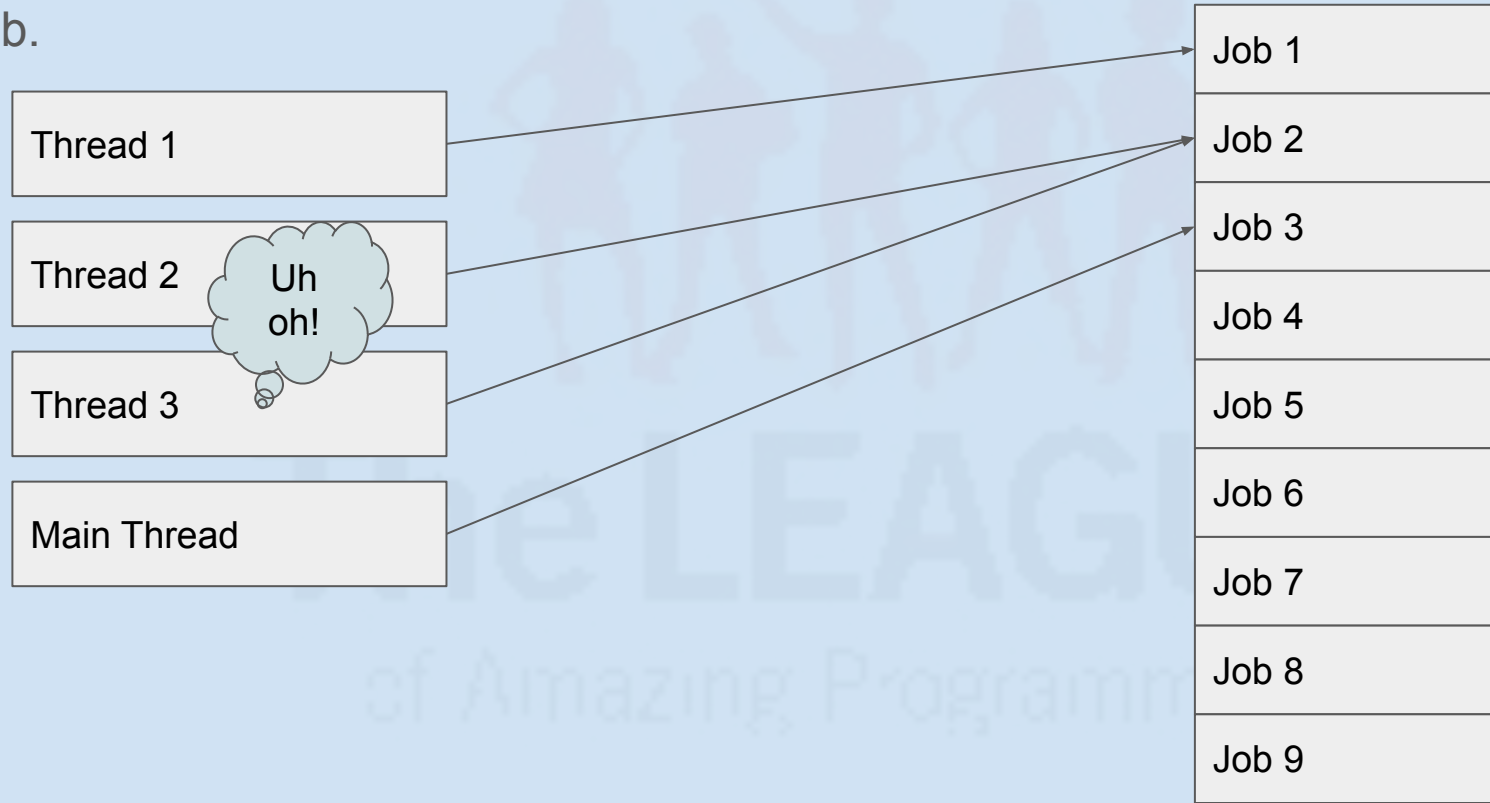
It's also possible that a thread might think the queue has jobs in it when it really doesn't. This could cause a crash.
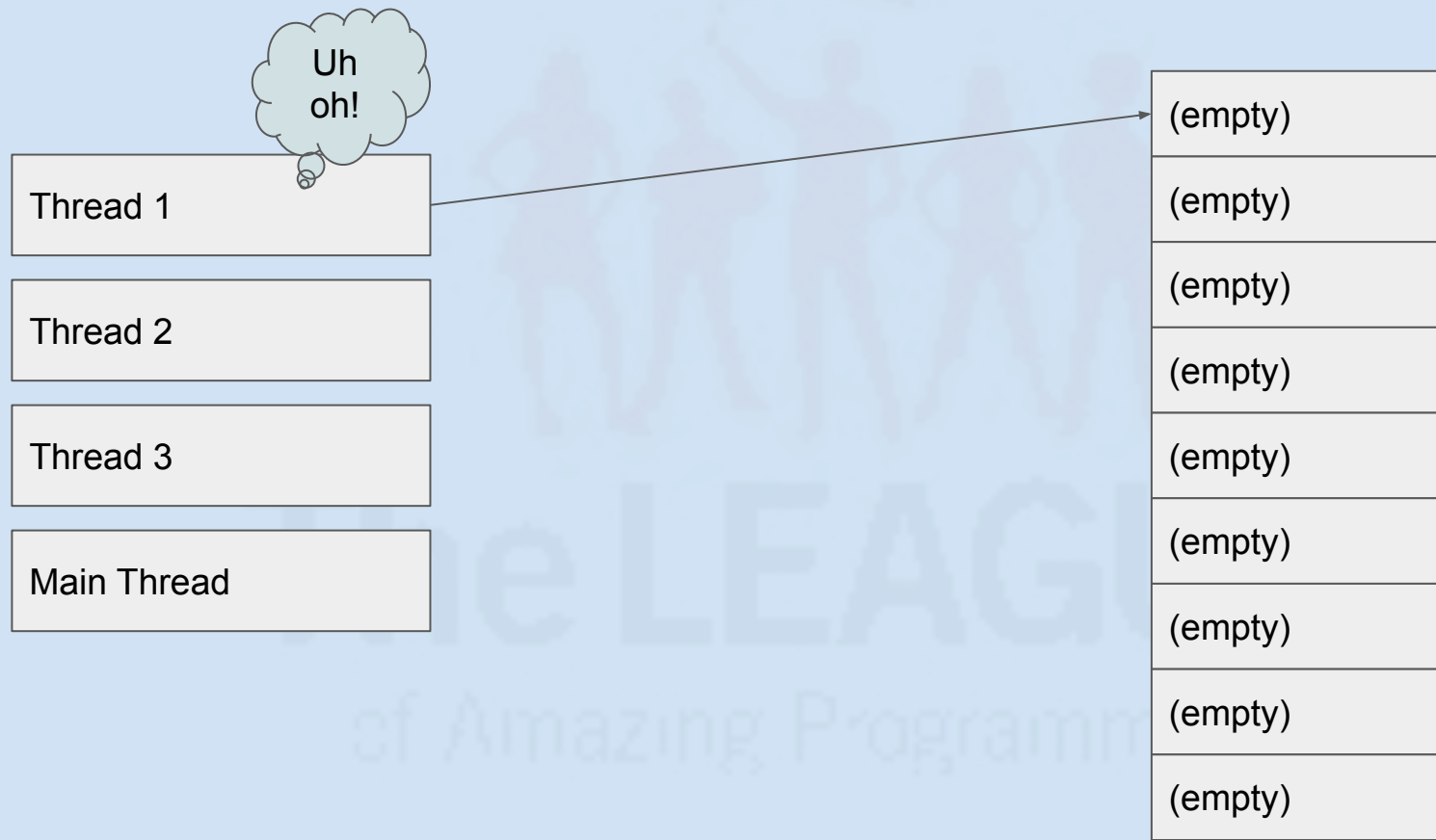
To fix this, we'll need to put a "lock" on the queue so that only one thread can check and remove from it at a time. We can do this with a *synchronized* block on our queue.

```java
Job j = null;
synchronized (jobQueue) {
    if (!jobQueue.isEmpty()) {
        j = jobQueue.remove();
    }
}
```

Now to check if we were successful in getting a job from the queue, we just have to see if our Job object is still **null**. If not, perform the job.

```java
if(j != null) {
        j.perform();
        return true;
}else {
        return false;
}
```
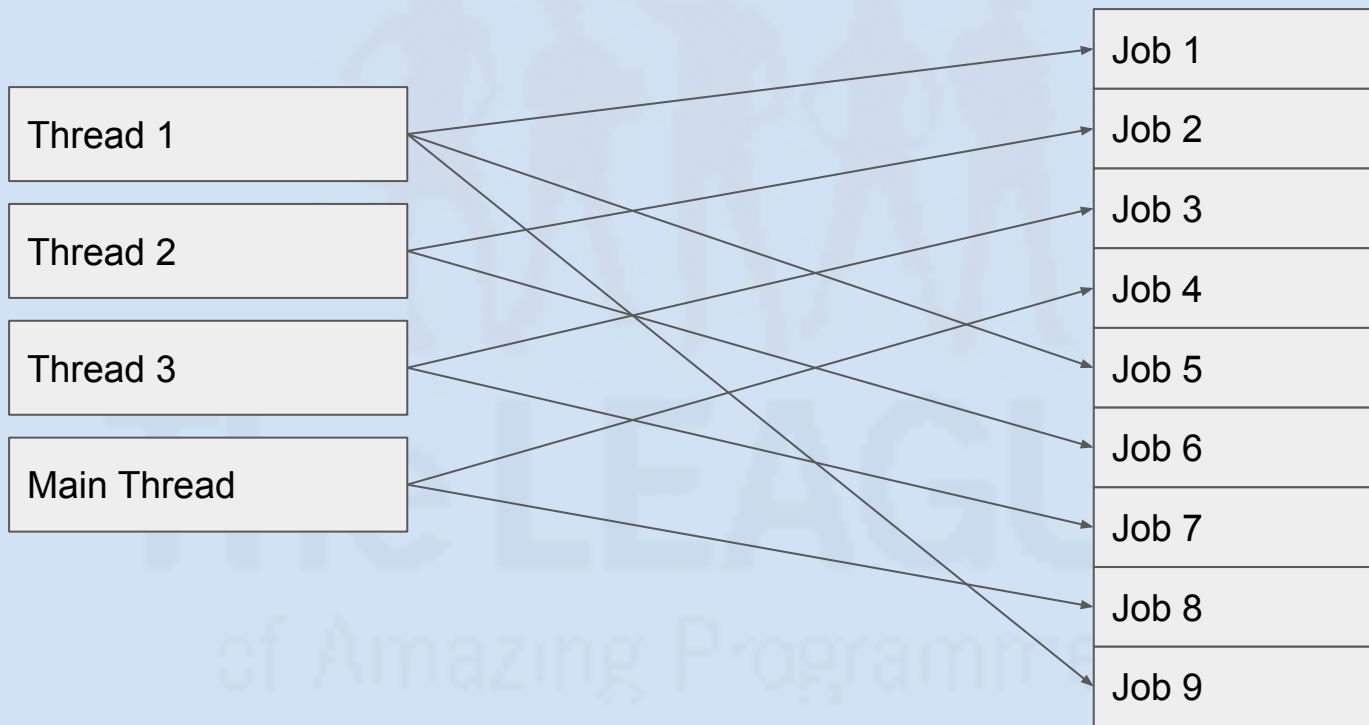
The whole *performJob* method should look something like this:

```java
public boolean performJob() {
    Job j = null;
    synchronized (jobQueue) {
        if(!jobQueue.isEmpty()) {
            j = jobQueue.remove();
        }
    }
    if(j != null) {
        j.perform();
        return true;
    }else {
        return false;
    }
}
```

We can now modify our *run* method to only sleep when it is unable to perform a job.

```java
public void run() {
    while (isRunning) {
        if (!performJob()) {
            synchronized (jobQueue) {
                try {
                    jobQueue.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

We're almost finished. We just need a way to pause the main thread so that the workers can finish all the jobs. Or even better, have the main thread attempt to complete some jobs of its own while it is waiting.

| Thread 1 |
| Thread 2 |
| Thread 3 |
| Main Thread |

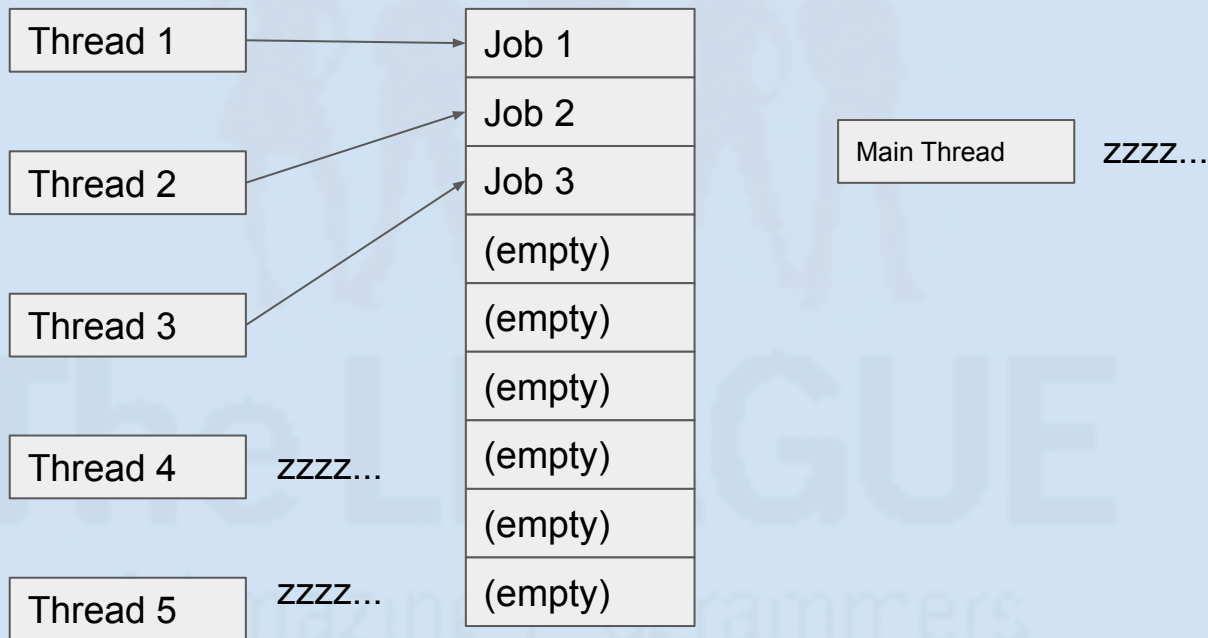| Job 1 |
| Job 2 |
| Job 3 |
| Job 4 |
| Job 5 |
| Job 6 |
| Job 7 |
| Job 8 |
| Job 9 |

Create a void method in your WorkQueue class called completeAllJobs. This method simply calls *performJob* in a loop as long as the **jobQueue** is not empty.

```java
public void completeAllJobs() {
    while (!jobQueue.isEmpty()) {
        performJob();
    }
}
```

We don't need to worry about locking the queue in this case since we already made *performJob* handle that for us.

If there are no jobs for the main thread to do, then we want the main thread to hold on here until the other threads have finished their jobs. We could figure out a way to put the main thread to sleep until the other threads are done, but we don't want to make things complicated unless we need to.

Thread 1

Thread 2

Thread 3

Thread 4	zzzz...

Thread 5	zzzz...

Job 1
Job 2
Job 3
(empty)
(empty)
(empty)
(empty)
(empty)
(empty)

Main Thread	zzzz...

For now, let's just keep the main thread in a loop until all the threads are waiting for a new job. We can check the state of a thread with the *getState()* method. Let's modify our *completeAllJobs()* method to loop the main thread until the other threads are no longer in the **WAITING** state.

```java
public void completeAllJobs() {
    while(!jobQueue.isEmpty()) {
        performJob();
    }

    for(int i = 0; i < threads.length; i++) {
        if(threads[i].getState() != State.WAITING) {
            i = -1;
        }
    }
}
```

We want to make sure that all the jobs have been completed before shutting down. Add a call to *completeAllJobs* at the top of the *shutdown* method. This will also prevent any threads from being stuck in the waiting (or sleep) state after shutting down.

```java
public void shutdown() {
    completeAllJobs();
    isRunning = false;
    synchronized (jobQueue) {
        jobQueue.notifyAll();
    }
}
```

Let's test! First, we'll see if our Work Queue is working. Then we'll run some tests to see how well it performs compared to a single threaded program.

Runs: 1/1   ⊠ Errors: 0   ⊠ Failures: 0

The first test will add 1000 jobs to the queue that simply print a number. Try this yourself in your main method.

```java
public static void main(String[] args) {
    WorkQueue wq = new WorkQueue();
    for (int i = 0; i < 1000; i++) {
        int x = i;
        Job j = ()->{
            System.out.println("Printing " + x + " from thread #" +
                               Thread.currentThread().getId());
        };
        wq.addJob(j);
    }
    wq.shutdown();
}
```

of Amazing Programmers

The output should have all numbers from 0 to 999 printed and every thread should have a roughly even distribution of jobs performed. Jobs done by thread #1 were done by the main thread. Go main thread!!

```
Printing 968 from thread #1
Printing 972 from thread #1
Printing 963 from thread #17
Printing 974 from thread #17
Printing 975 from thread #17
Printing 976 from thread #17
Printing 977 from thread #17
Printing 961 from thread #14
Printing 979 from thread #14
Printing 980 from thread #14
Printing 981 from thread #14
Printing 982 from thread #14
Printing 983 from thread #14
Printing 984 from thread #14
Printing 985 from thread #14
Printing 962 from thread #15
Printing 956 from thread #16
Printing 987 from thread #16
Printing 988 from thread #16
Printing 936 from thread #13
Printing 991 from thread #13
Printing 992 from thread #13
Printing 993 from thread #13
Printing 994 from thread #13
Printing 995 from thread #13
Printing 996 from thread #13
Printing 997 from thread #13
Printing 998 from thread #13
Printing 999 from thread #13
Printing 934 from thread #11
Printing 990 from thread #16
Printing 989 from thread #15
Printing 986 from thread #14
Printing 978 from thread #17
Printing 973 from thread #1
Printing 971 from thread #12
```

You now have a very basic thread pool with a working job queue. Notice that the threads don't necessarily complete the jobs in the same order that they were added to the queue. Therefore, this queue is only good for jobs that don't rely on the output from other jobs in the queue at the same time. It should work just fine for things like pixel processing.

Input image
f(x,y)

operation

Output image
g(x,y)
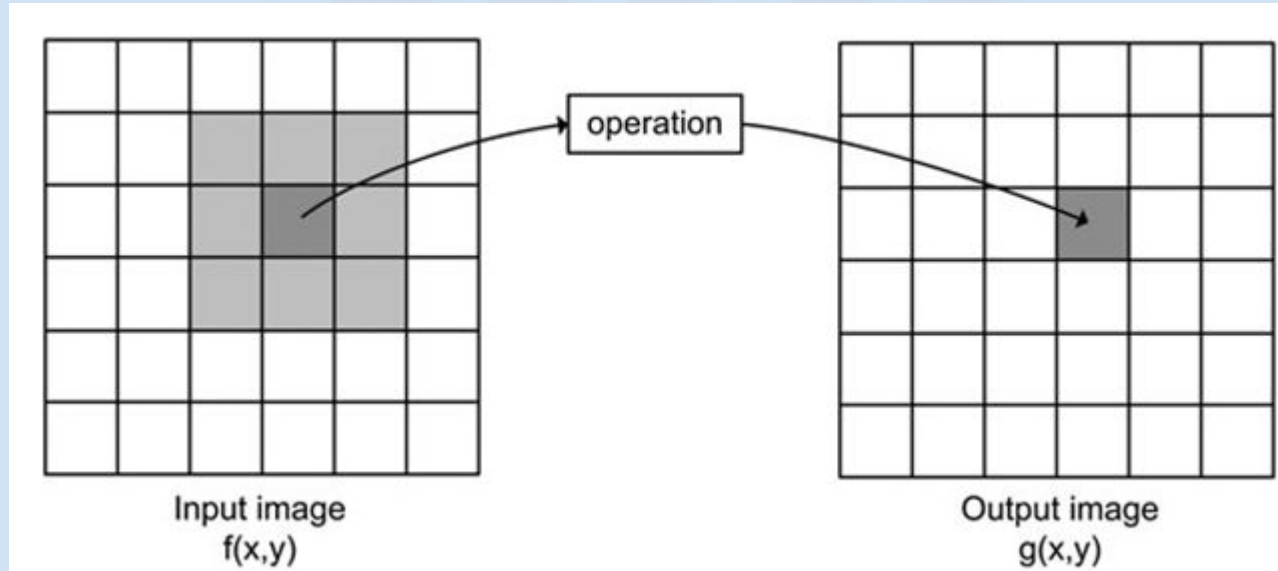
We will now test a couple of pixel processing algorithms to see if your thread pool can outperform a single threaded program. The first test will generate a giant 8192 x 8192 pixel array where every pixel's color is a representation of of its distance from the center of the image.
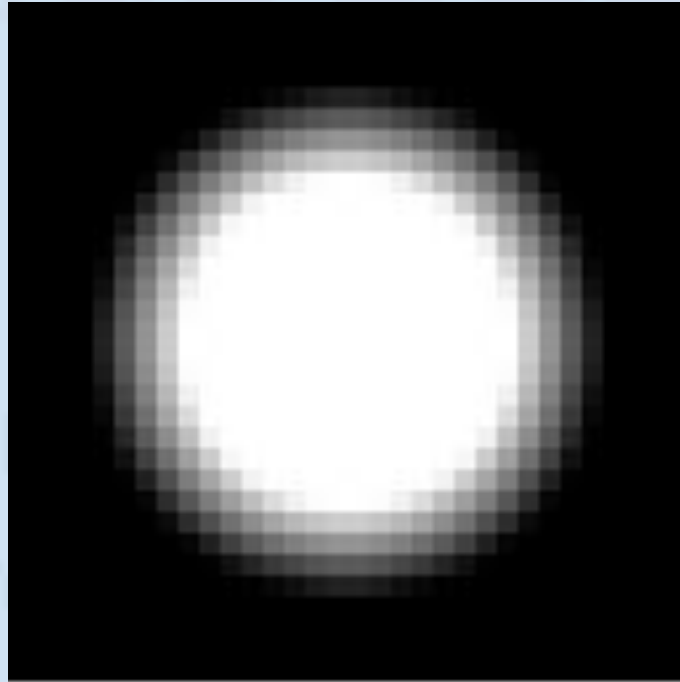
Run **DistanceFromCenterTest.java**. This program will run a speed and accuracy test. If it does not pass, check the output in the console to assess what the issue could be.


```
Distance From Center Test
Starting single threaded block
single thread time: 0.7767468 seconds
Starting multi-threaded block
milt-thread time: 0.0839094 seconds
Arrays are a match
The single-threaded block was 0.6928373999999999 seconds slower.
```
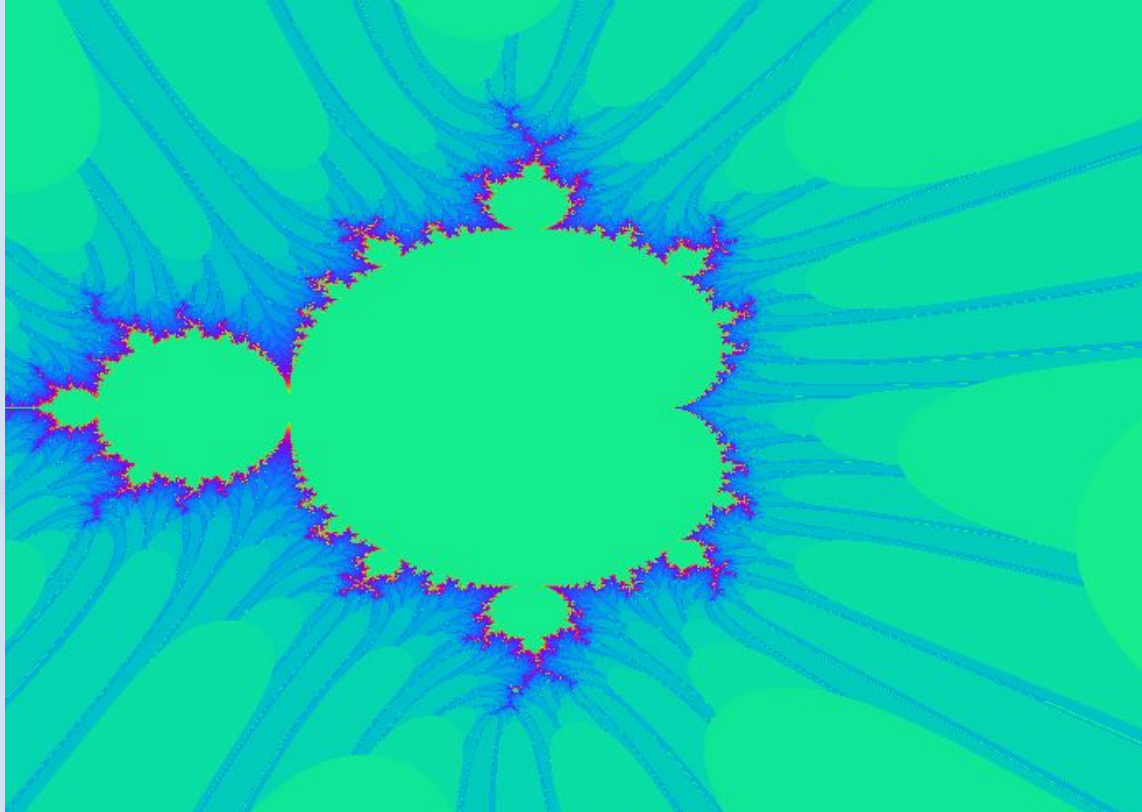
The next test will test the performance of your thread pool vs a single threaded program by producing a mathematically generated image called a Mandelbrot Set.

Run **Mandelbrot.java** and scroll around the Mandelbrot Set in both single threaded mode and multi threaded mode. You should notice that the multi threaded mode is consistently faster than the single threaded mode.

Press C for Controls
Mode: single thread
Detail Level: 64
Calculation Time: 0.1321329

Press C for Controls
Mode: multi thread
Detail Level: 64
Calculation Time: 0.0388665