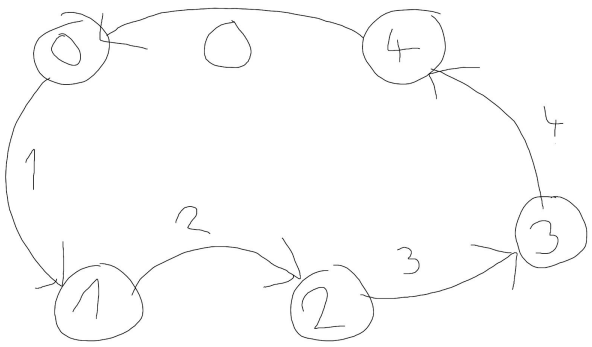
Dining Philosophers

Thread Action

Transition	Thread	Purpose	Thread Name
0-12	0	create fork/ phil. instances, launch threads	main
13	1	Attempts to lock fork 0, gets it	Thread-1
14	1	Attempts to lock fork 1	Thread-1
15	2	Attempts to lock fork 1, gets it	Thread-2
16	2	Attempts to lock fork 2	Thread-2
17	3	Attempts to lock fork 2, gets it	Thread-3
18	3	Attempts to lock fork 3	Thread-3
19	4	Attempts to lock fork 3, gets it	Thread-4
20	4	Attempts to lock fork 4	Thread-4
21	5	Attempts to lock fork 4, gets it	Thread-5
22	5	Attempts to lock fork 5	Thread-5

Dependency Graph of Forks/Locks



Patch

We introduce a new lock so acquiring forks becomes an atomic operation.

```
--- DiningPhil.java
+++ DiningPhil_fixed.java
@@ -19,40 +19,45 @@
 //

 public class DiningPhil {
-
        static class Fork {
        }
+       static class Lock {
+       }

        static class Philosopher extends Thread {

          Fork left;
          Fork right;
+         Lock lock;

-         public Philosopher(Fork left, Fork right) {
+         public Philosopher(Fork left, Fork right, Lock lock) {
                this.left = left;
                this.right = right;
+               this.lock = lock;
                start();
          }

          public void run() {
-               // think!
-               synchronized (left) {
-                 synchronized (right) {
-                       // eat!
-                 }
+
+               synchronized (lock) {
+                 synchronized (left) {
+                       synchronized (right) {
+                         // eat!
+                       }
+                         }
                }
          }
        }

        static final int N = 5;
-
        public static void main(String[] args) {
+         Lock lock = new Lock();
          Fork[] forks = new Fork[N];
          for (int i = 0; i < N; i++) {
                forks[i] = new Fork();
          }
          for (int i = 0; i < N; i++) {
-               new Philosopher(forks[i], forks[(i + 1) % N]);
+               new Philosopher(forks[i], forks[(i + 1) % N], lock);
          }
        }
   }
```

Bounded Queue

Part 2

Transition	Thread	Purpose	Thread Name
0-3	0	Main thread	main
4	1	?	Thread-1
5	1	Adds data to queue	Thread-1
6	1	Attempts to remove data from queue, waits	Thread-1
7	2	Attempts to add data to queue, waits	Thread-2

Part 3

Transition	Thread	Purpose	Thread Name
0-25	0	?	main
26-35	0	Initialize consumers and producers	main
36	1	Locks put	Thread-1
37-41	2	Locks remove	Thread-2
42	2	Waits for data in queue, releases remove	Thread-2
43	3	Attempts to get put	Thread-3
44	3	Fails to get put, Waits	Thread-3
45	4	Locks remove	Thread-4
46	4	?	Thread-4
47	4	Waits for data in queue, releases	Thread-4

		remove	
48	1	Executes put	Thread-1
49	1	Calls notify	Thread-1
50	1	Releases lock	Thread-1
51	3	Attempts to put, queue is full, has to wait	Thread-3

Problem arises since we call notify() instead of notify all. This leads to us having two waiting consumers and a full queue, making it impossible to ever wake them up

Part 4

The problem is with using notify() instead of using notifyAll(). In queueTest we only have two workers so using notify will always wake up the opposing thread. The reason the ProdCons could fail is that we have two producers and two consumers. If we have 2 consumers waiting and 1 producer waiting and the last producer finishes and wakes up the other producers we will fail. In queueTest both workers can both add and remove stuff from the queue.

Daytime

```
 import java.io.IOException;
 import java.io.OutputStreamWriter;
-import java.net.ServerSocket;
-import java.net.Socket;
+import env.java.net.ServerSocket;
+import env.java.net.Socket;
 import java.util.Date;

 public class DaytimeServer {
@@ -33,7 +33,9 @@ public class DaytimeServer {
     }
     finally {
       try {
-        connection.close();
+        if(connection != null){
+          connection.close();
+        }
        server.close();
       }
       catch(IOException e) {
```