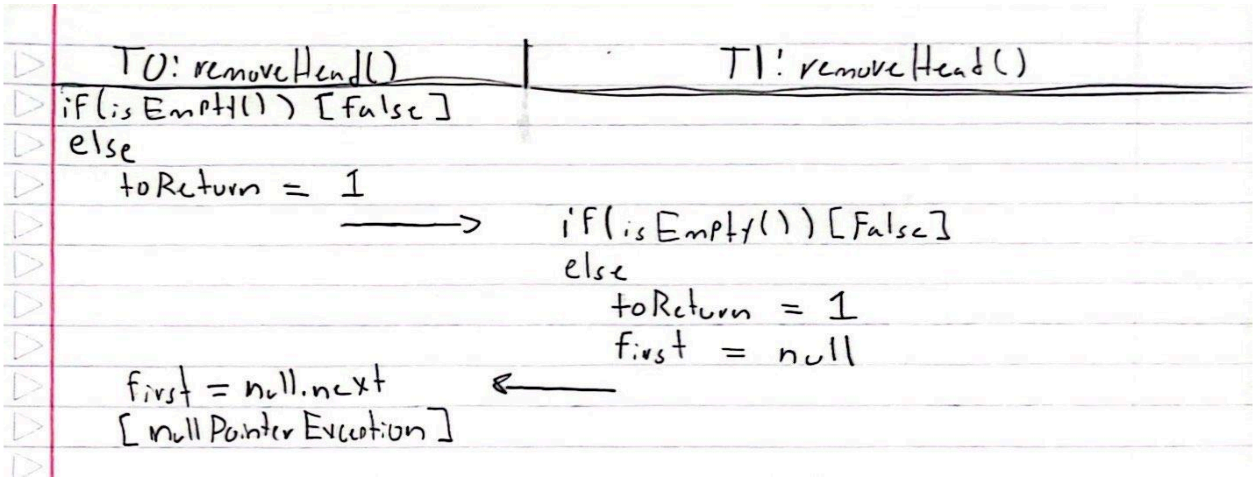
Part 1

Interleaving 1 Error Fixing:

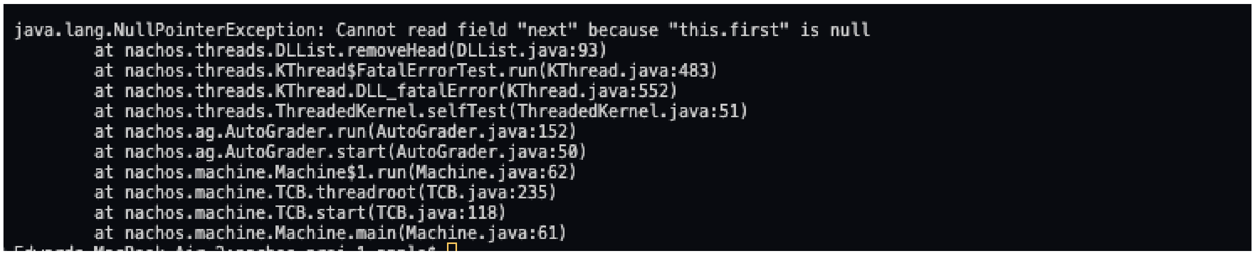


```
T0: removeHead()                     |      T1: removeHead()
if(isEmpty()) [False]
else
   toReturn = 1
                     ------>          if(isEmpty()) [False]
                                      else
                                         toReturn = 1
                                         first = null
   first = null.next       <------
   [null Pointer Exception]
```

The initial state of this interleaving prepends the DLList to include "1". Before monitor changes, this interleaving created a Fatal Error. To run this interleaving, go to the ThreadedKernal Class, and uncomment the line: `KThread.DLL_fatalError();`

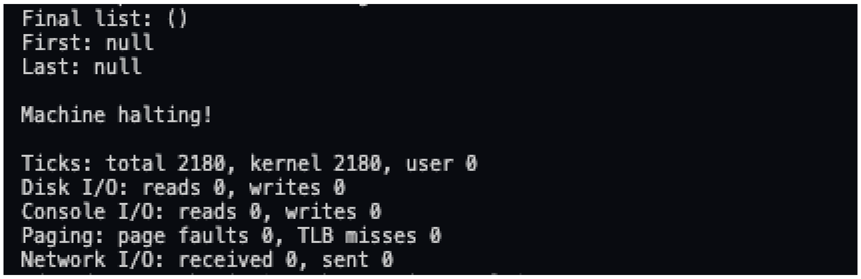
After running, you should see the following error:

java.lang.NullPointerException: Cannot read field "next" because "this.first" is null



```
java.lang.NullPointerException: Cannot read field "next" because "this.first" is null
        at nachos.threads.DLList.removeHead(DLList.java:93)
        at nachos.threads.KThread$FatalErrorTest.run(KThread.java:483)
        at nachos.threads.KThread.DLL_fatalError(KThread.java:552)
        at nachos.threads.ThreadedKernel.selfTest(ThreadedKernel.java:51)
        at nachos.ag.AutoGrader.run(AutoGrader.java:152)
        at nachos.ag.AutoGrader.start(AutoGrader.java:50)
        at nachos.machine.Machine$1.run(Machine.java:62)
        at nachos.machine.TCB.threadroot(TCB.java:235)
        at nachos.machine.TCB.start(TCB.java:118)
        at nachos.machine.Machine.main(Machine.java:61)
```

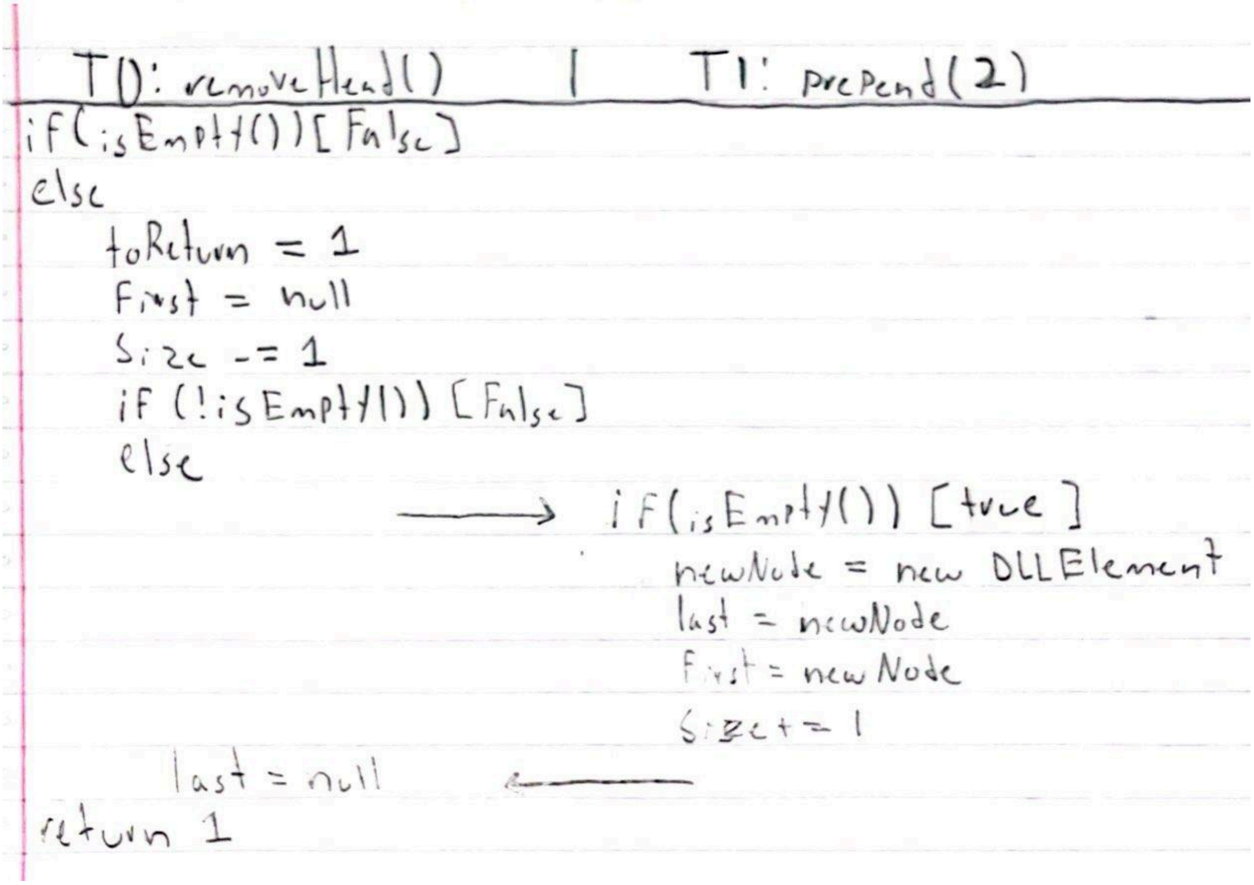
After the monitor changes, the following output is achieved:



```
Final list: ()
First: null
Last: null

Machine halting!

Ticks: total 2180, kernel 2180, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: page faults 0, TLB misses 0
Network I/O: received 0, sent 0
```

This output fixes the interleaving error because it matches the output expected when running removeHead() and removeHead() in sequential order.

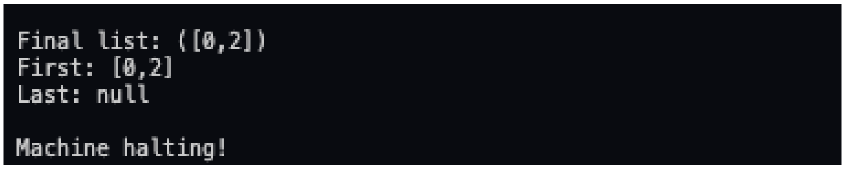
Interleaving 2 Error Fixing:



```
   T0: removeHead()          |      T1: prePend(2)
if(isEmpty())[False]
else
    toReturn = 1
    First = null
    Size -= 1
    if (!isEmpty()) [False]
    else
                     ──────→  if(isEmpty()) [true]
                                 newNode = new DLLElement
                                 last = newNode
                                 First = new Node
                                 Size += 1
        last = null      ←──────
return 1
```

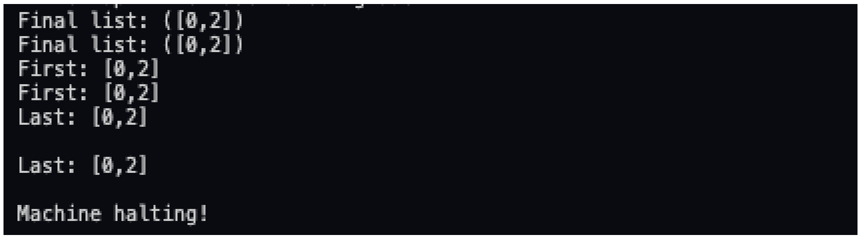
The initial state of this interleaving prepends the DLList to include "1". The prepend method is called with the integer value 2. Before monitor changes, this interleaving created a Non Fatal Error. To recreate this interleaving, go to the ThreadedKernal Class, and uncomment the line: *`KThread.DLL_nonFatalError();`*

After running, the final output you should see is:



```
Final list: ([0,2])
First: [0,2]
Last: null

Machine halting!
```

After the monitor changes, the final output you should see is:



```
Final list: ([0,2])
Final list: ([0,2])
First: [0,2]
First: [0,2]
Last: [0,2]

Last: [0,2]

Machine halting!
```

A final output of,
Final list: ([0,2])
First: [0,2]
Last: [0,2]
is valid because it matches the output of the sequential order of the threads running removeHead() then prepend(2).

Part 2

Mutual Exclusion Test:

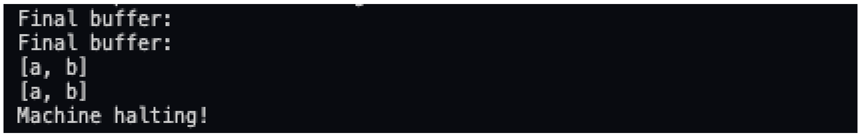
This test initializes an empty BoundedBuffer with a size of 2. The interleaving above shows the error that would occur without the monitor changes. The resulting output would have resulted in a buffer with the following contents: [b,]

This is an invalid output because it does not match either of the following sequential order outputs:

write('a') then write('b'):
Final buffer: [a, b]

write('b') then write('a'):
Final buffer: [b, a]

The output after the monitor changes is valid because it matches one of these sequential ordering:



```
Final buffer:
Final buffer:
[a, b]
[a, b]
Machine halting!
```

Underflow Test:

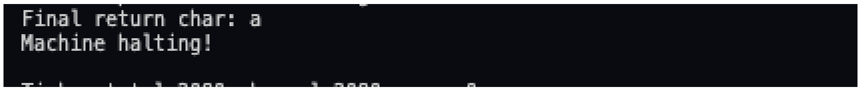
Thread 1:
read()

Thread 2:
write('a')

This test initializes an empty BoundedBuffer with a size of 2, and starts with Thread 1 running. The test above shows an error that would occur without the monitor changes. The resulting output would have been:
Final return char: null

This is an invalid output because it is an underflow where an empty buffer is being read from, resulting in a 'null' return character.

The output after the monitor changes is valid because it prevents the underflow by allowing a character to be written before the buffer is read from:



```
Final return char: a
Machine halting!
```

Overflow Test:

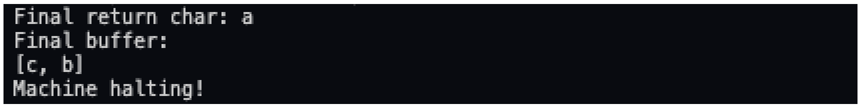
Thread 1:
write('a')
write('b')
write('c')

Thread 2:
read()

This test initializes an empty BoundedBuffer with a size of 2, and begins with Thread 1 running. The test above shows an error that would occur without the monitor changes. The resulting output would have been:
Final return char: b
Final buffer: [c,]

This is an invalid output because it is an overflow where a full buffer is being written to, resulting in character 'a' being lost.

The output after the monitor changes is valid because it prevents the underflow by allowing a character to be written before the buffer is read from:

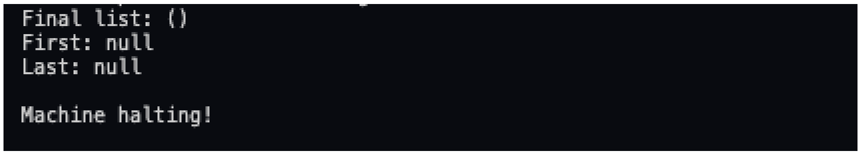


```
Final return char: a
Final buffer:
[c, b]
Machine halting!
```

Part 3

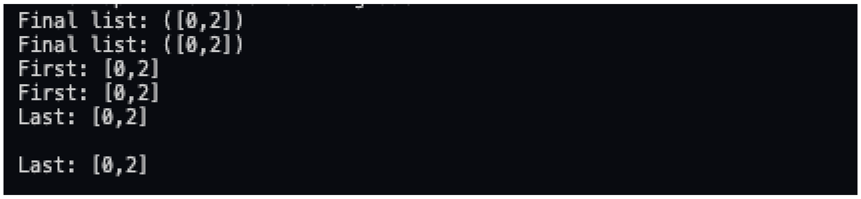
The following are the outputs of the part 1 and 2 tests, using the Condition2 instead of Condition. They confirm that Condition2 is working, because the output of these tests match the output of the tests with the Condition class.

Part 1 Interleaving 1:



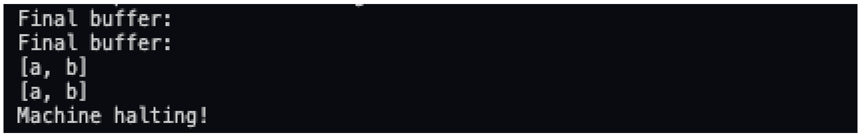
```
Final list: ()
First: null
Last: null

Machine halting!
```

Part 1 Interleaving 2:



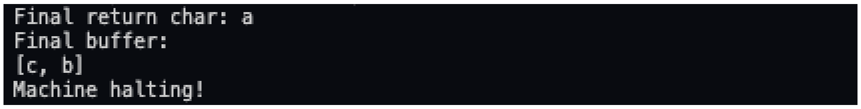
```
Final list: ([0,2])
Final list: ([0,2])
First: [0,2]
First: [0,2]
Last: [0,2]

Last: [0,2]
```

Part 2 Mutual Exclusion:



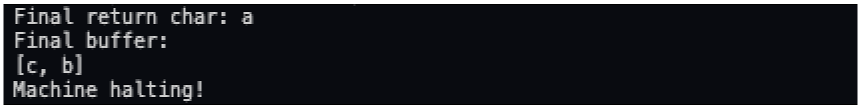
```
Final buffer:
Final buffer:
[a, b]
[a, b]
Machine halting!
```

Part 2 Underflow:



```
Final return char: a
Final buffer:
[c, b]
Machine halting!
```

Part 2 Overflow:



```
Final return char: a
Final buffer:
[c, b]
Machine halting!
```

Source Code:

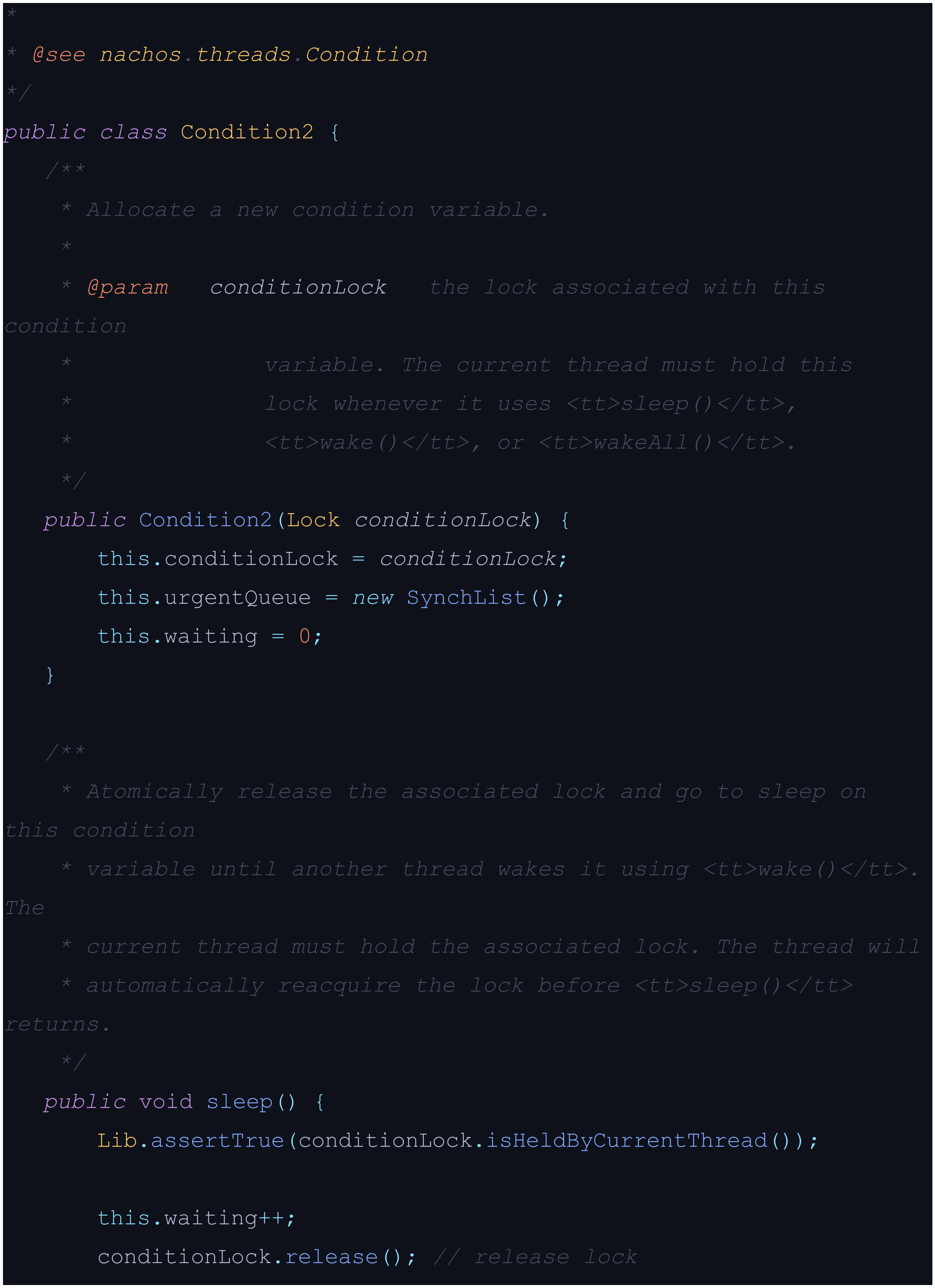
BoundedBuffer:

```java
package nachos.threads;
import java.util.Arrays;

public class BoundedBuffer {

   private Lock lock;       // class lock object
   private char[] buffer;   // buffer contents
   private int count;       // number of items in the buffer
   private int n;           // max buffer size
   private int nextIn, nextOut; // indexes denoting where the next char should be input and output
   private Condition2 emptySlot, fullSlot; // full and empty slot Condition2s

   // non-default constructor with a fixed size
   public BoundedBuffer(int maxsize) {
       lock = new Lock();
       buffer = new char[maxsize];
       count = nextIn = nextOut = 0;
       n = maxsize;
       emptySlot = new Condition2(lock);
       fullSlot = new Condition2(lock);
```

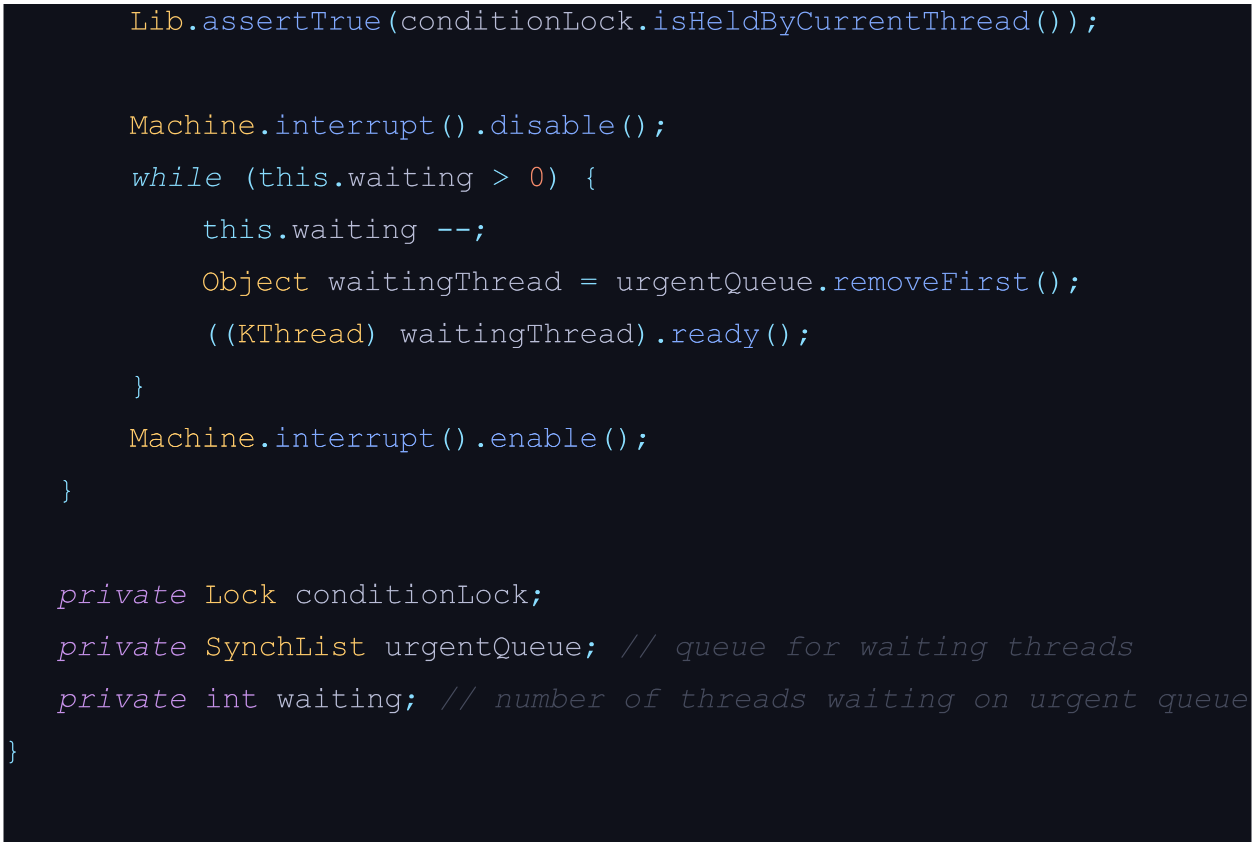
```
    }

    // Read a character from the buffer, blocking until there is a
char
    // in the buffer to satisfy the request. Return the char read.
    public char read() {
        lock.acquire();
        while (count <= 0) {
            fullSlot.sleep();
        }

        char c = buffer[nextOut];
        nextOut = (nextOut + 1) % n;
        count--;
        emptySlot.wake();
        lock.release();

        return c;

    }


    // Write the given character c into the buffer, blocking until
    // enough space is available to satisfy the request.
    public void write(char c) {
        lock.acquire();
        while (count == n) {
            emptySlot.sleep();
        }

        buffer[nextIn] = c;

        KThread.yieldIfShould(0);
        nextIn = (nextIn + 1) % n;
        count++;
        fullSlot.wake();

        lock.release();

    }

    // Prints the contents of the buffer; for debugging only
    public void print() {
        lock.acquire();
        System.out.println(Arrays.toString(buffer));
        lock.release();

    }

}
```

Condition2:

```
package nachos.threads;

import nachos.machine.*;

/**
* An implementation of condition variables that disables
interrupt()s for
* synchronization.
*
* <p>
* You must implement this.
```

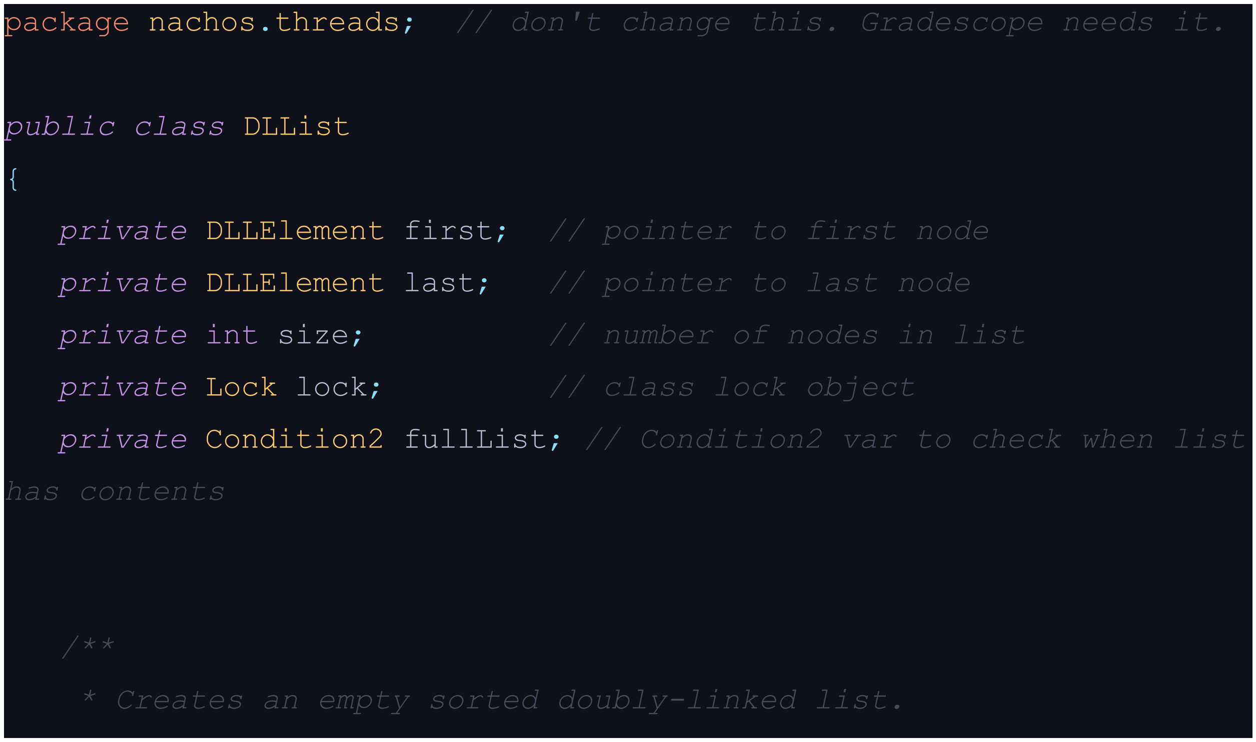


```
*
* @see nachos.threads.Condition
*/
public class Condition2 {
    /**
     * Allocate a new condition variable.
     *
     * @param   conditionLock   the lock associated with this
condition
     *              variable. The current thread must hold this
     *              lock whenever it uses <tt>sleep()</tt>,
     *              <tt>wake()</tt>, or <tt>wakeAll()</tt>.
     */
    public Condition2(Lock conditionLock) {
        this.conditionLock = conditionLock;
        this.urgentQueue = new SynchList();
        this.waiting = 0;
    }

    /**
     * Atomically release the associated lock and go to sleep on
this condition
     * variable until another thread wakes it using <tt>wake()</tt>.
The
     * current thread must hold the associated lock. The thread will
     * automatically reacquire the lock before <tt>sleep()</tt>
returns.
     */
    public void sleep() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());

        this.waiting++;
        conditionLock.release(); // release lock
        Machine.interrupt().disable(); // disable interrupts
        this.urgentQueue.add(KThread.currentThread()); // add current
thread to queue
        KThread.currentThread().sleep(); // block current thread
        Machine.interrupt().enable();// enable interrupts (when
thread gets woken up)

        conditionLock.acquire(); // reacquire the lock
    }

    /**
     * Wake up at most one thread sleeping on this condition
variable. The
     * current thread must hold the associated lock.
     */
    public void wake() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());

        Machine.interrupt().disable();
        if (this.waiting > 0) {
            this.waiting --;
            Object waitingThread = urgentQueue.removeFirst();
            ((KThread) waitingThread).ready();
        }
        Machine.interrupt().enable();
    }

    /**
     * Wake up all threads sleeping on this condition variable. The
current
     * thread must hold the associated lock.
     */
    public void wakeAll() {
```

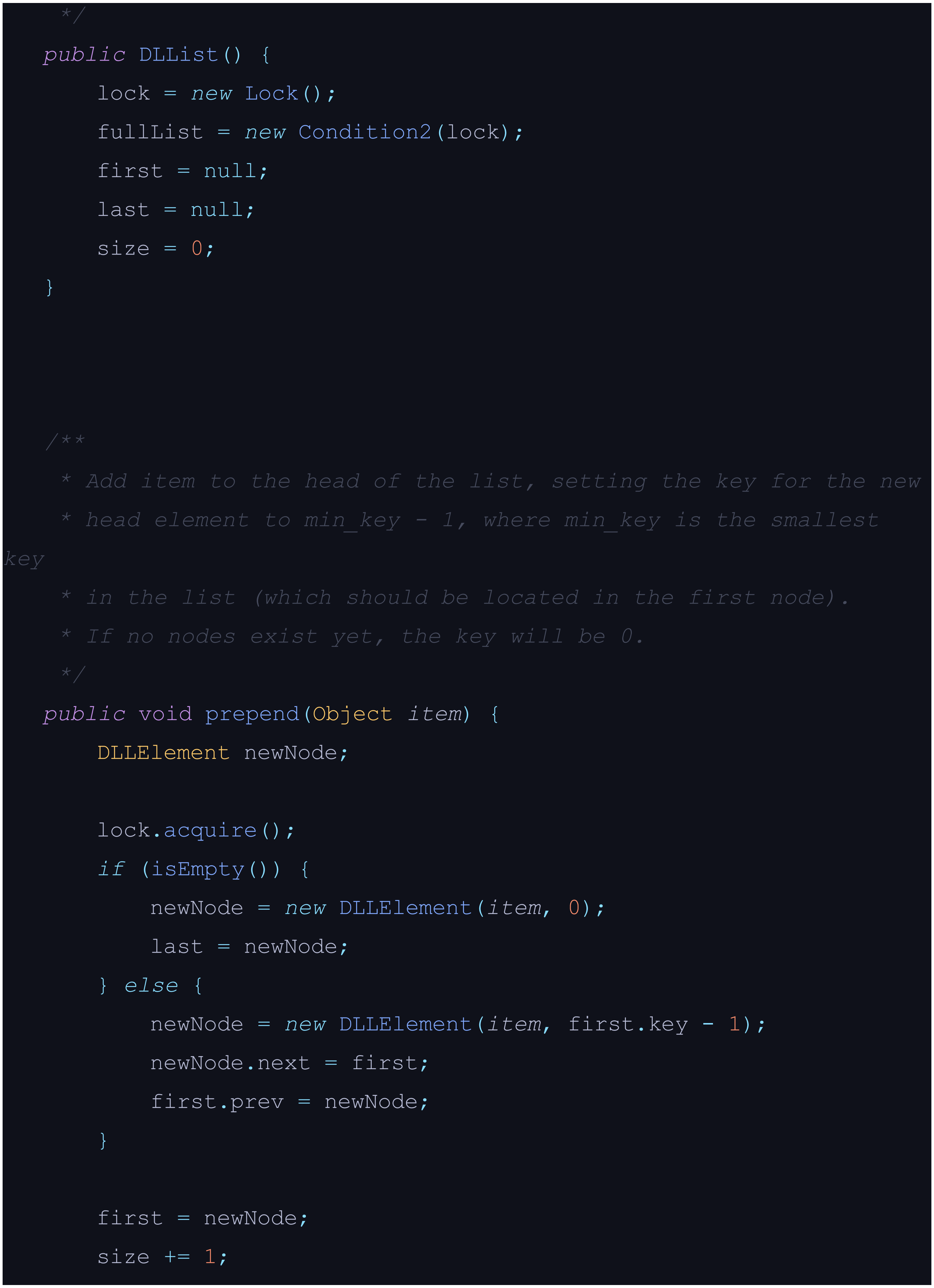


```
       Lib.assertTrue(conditionLock.isHeldByCurrentThread());

       Machine.interrupt().disable();
       while (this.waiting > 0) {
           this.waiting --;
           Object waitingThread = urgentQueue.removeFirst();
           ((KThread) waitingThread).ready();
       }
       Machine.interrupt().enable();
   }

   private Lock conditionLock;
   private SynchList urgentQueue; // queue for waiting threads
   private int waiting; // number of threads waiting on urgent queue
}
```

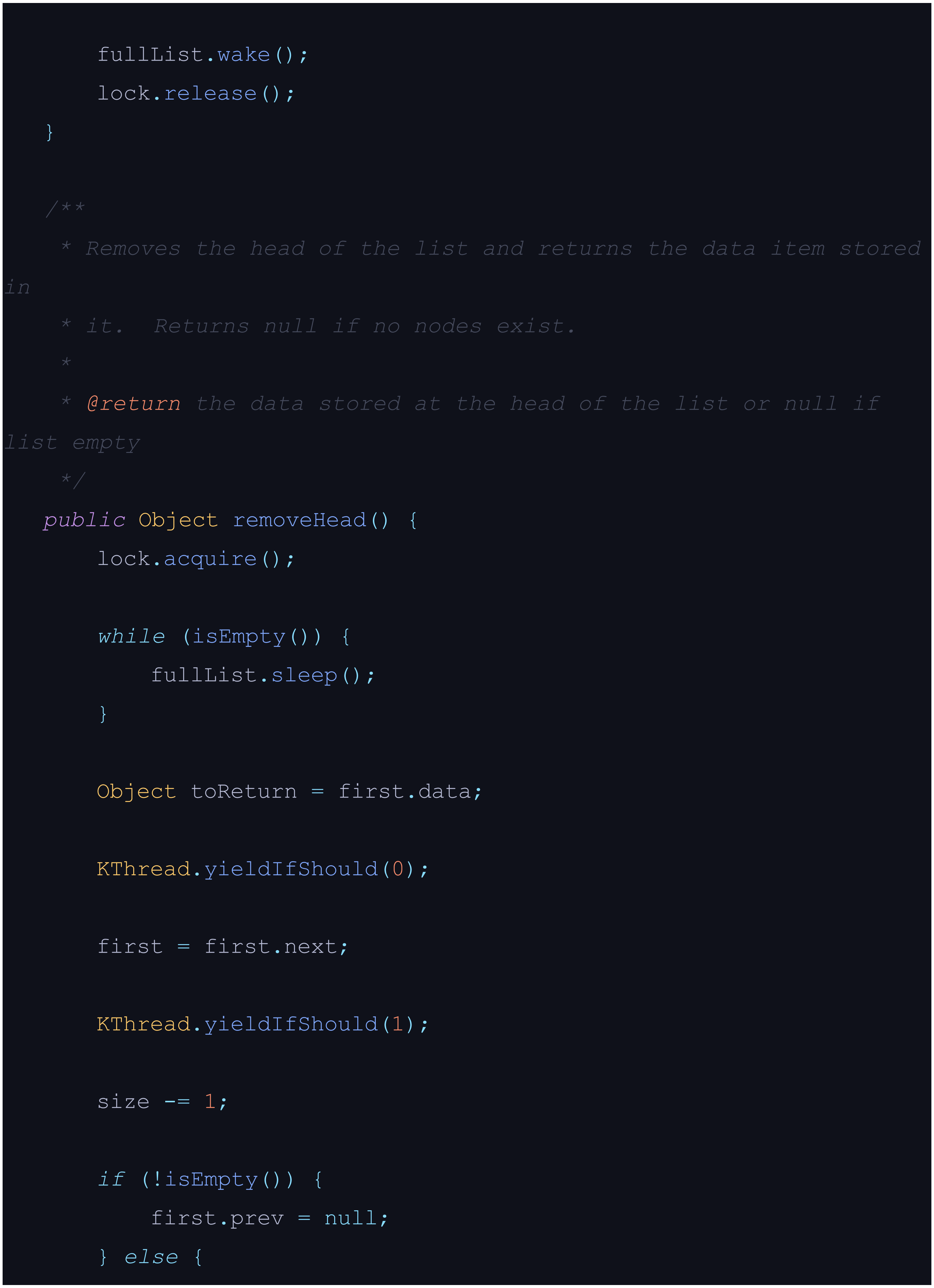
DLList:



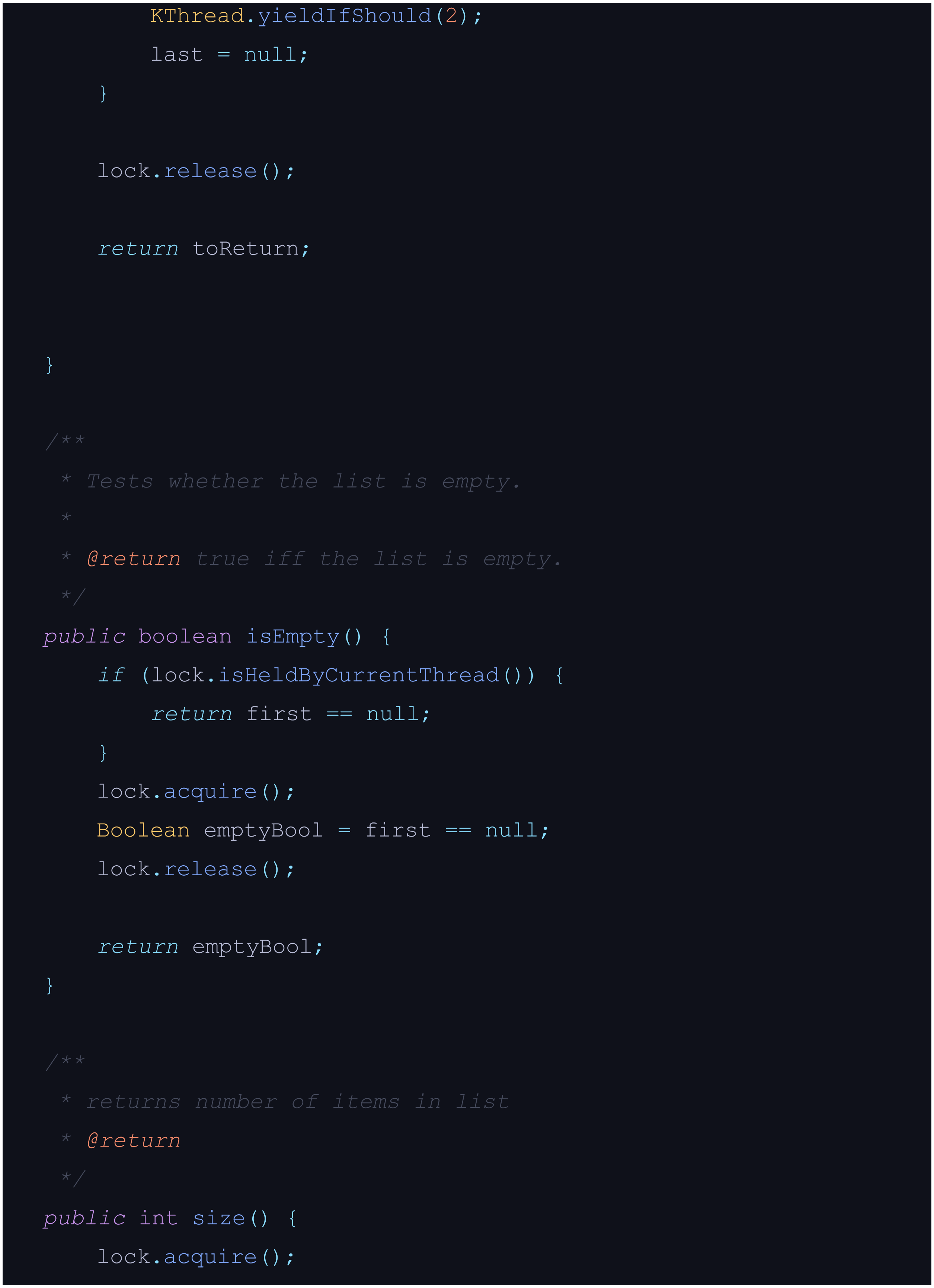
```
package nachos.threads;  // don't change this. Gradescope needs it.

public class DLList
{
   private DLLElement first;  // pointer to first node
   private DLLElement last;   // pointer to last node
   private int size;          // number of nodes in list
   private Lock lock;         // class lock object
   private Condition2 fullList; // Condition2 var to check when list
has contents


   /**
    * Creates an empty sorted doubly-linked list.
```



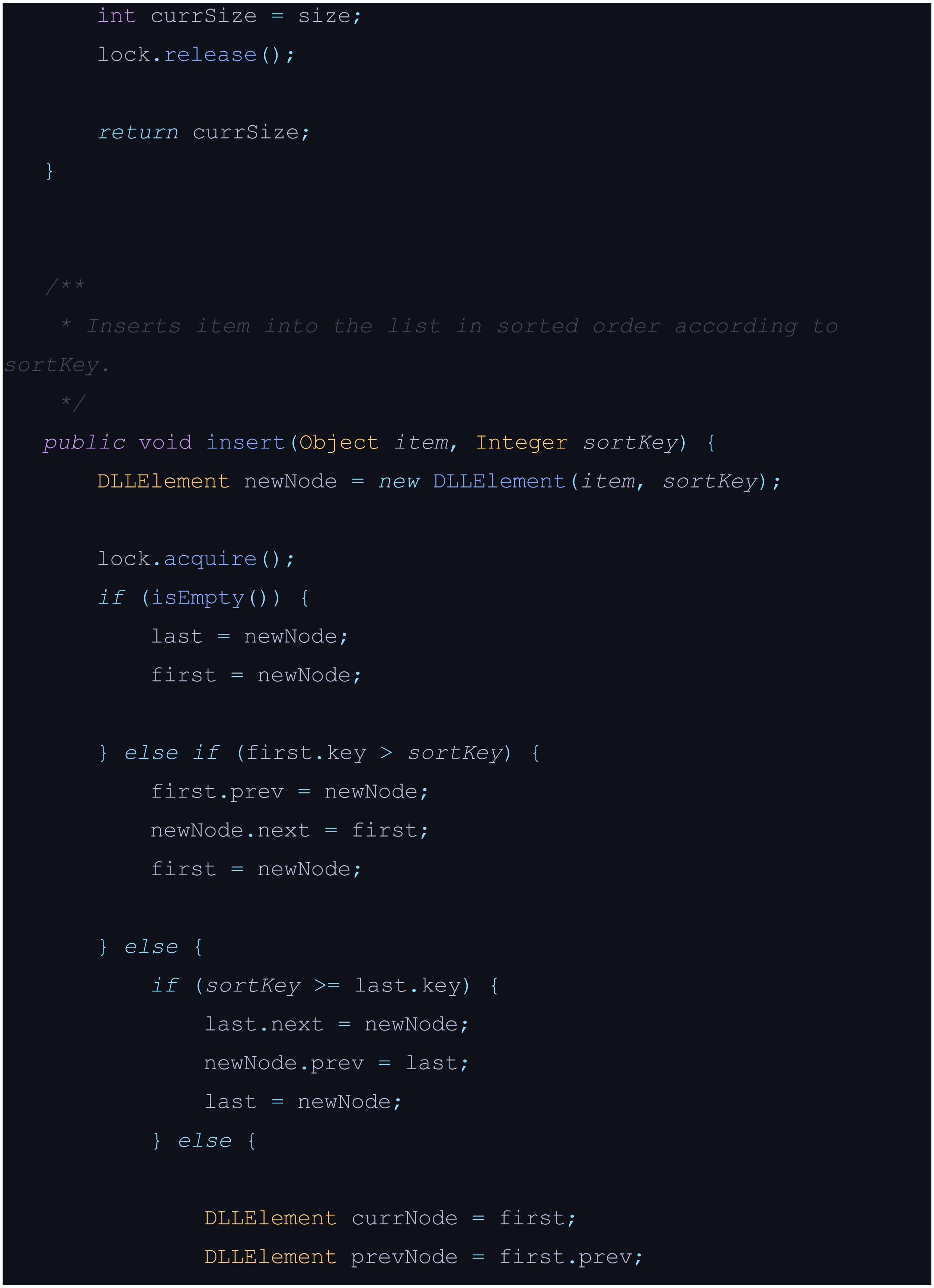
```
     */
    public DLList() {
        lock = new Lock();
        fullList = new Condition2(lock);
        first = null;
        last = null;
        size = 0;
    }



    /**
     * Add item to the head of the list, setting the key for the new
     * head element to min_key - 1, where min_key is the smallest
key
     * in the list (which should be located in the first node).
     * If no nodes exist yet, the key will be 0.
     */
    public void prepend(Object item) {
        DLLElement newNode;

        lock.acquire();
        if (isEmpty()) {
            newNode = new DLLElement(item, 0);
            last = newNode;
        } else {
            newNode = new DLLElement(item, first.key - 1);
            newNode.next = first;
            first.prev = newNode;
        }

        first = newNode;
        size += 1;
```



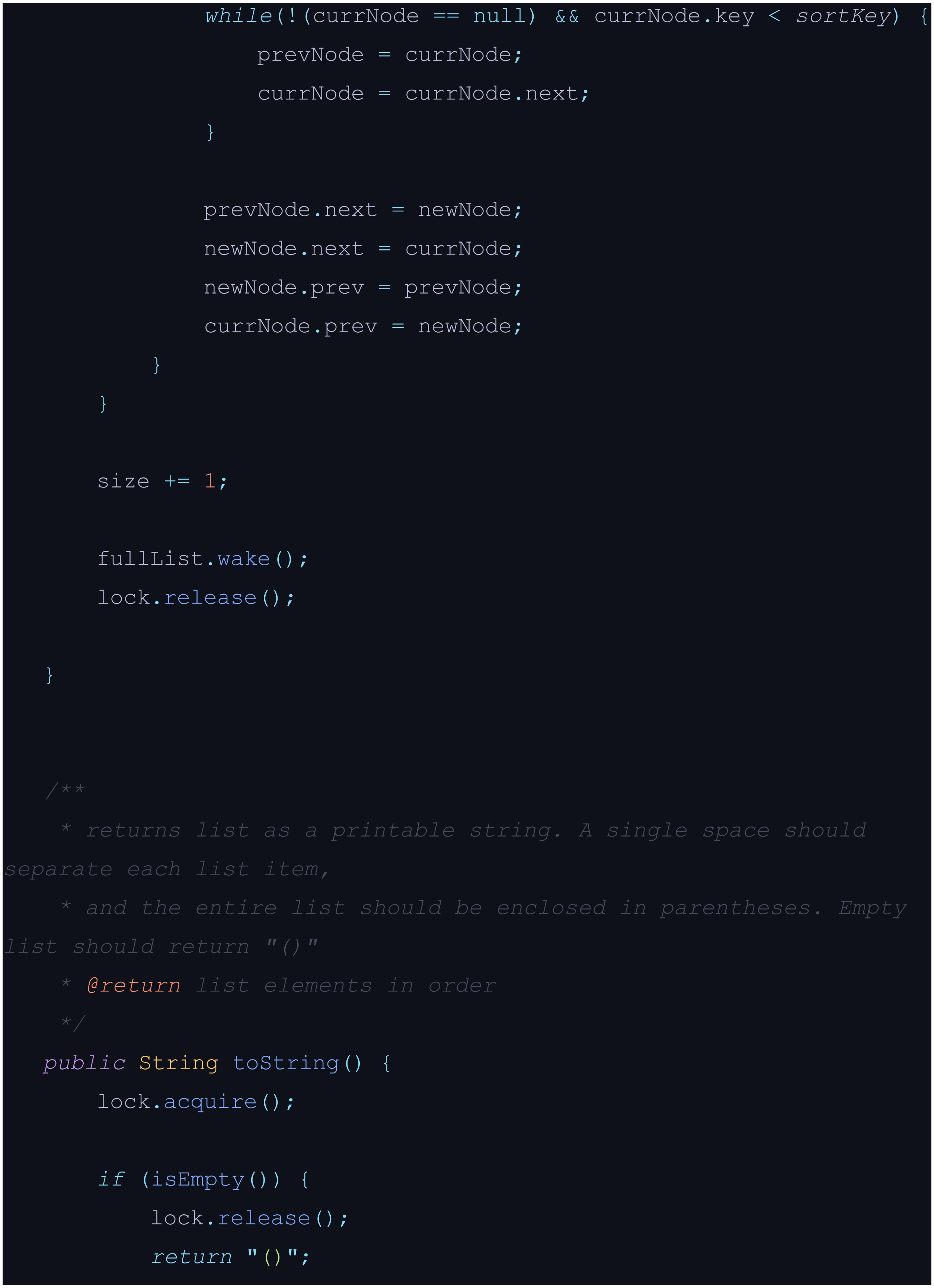
```
        fullList.wake();
        lock.release();
    }

    /**
     * Removes the head of the list and returns the data item stored
in
     * it.  Returns null if no nodes exist.
     *
     * @return the data stored at the head of the list or null if
list empty
     */
    public Object removeHead() {
        lock.acquire();

        while (isEmpty()) {
            fullList.sleep();
        }

        Object toReturn = first.data;

        KThread.yieldIfShould(0);

        first = first.next;

        KThread.yieldIfShould(1);

        size -= 1;

        if (!isEmpty()) {
            first.prev = null;
        } else {
```



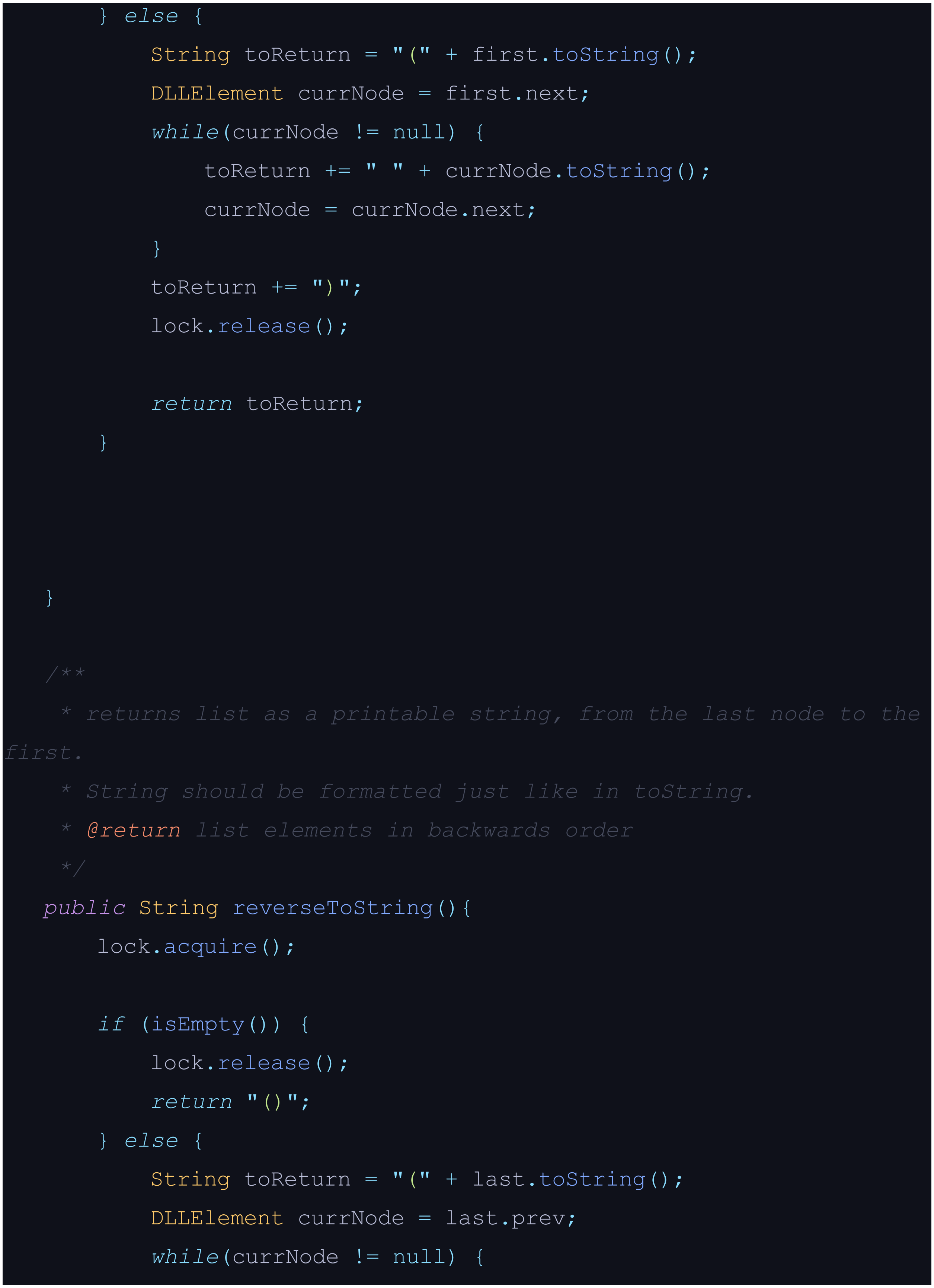
```
            KThread.yieldIfShould(2);
            last = null;
        }

        lock.release();

        return toReturn;


    }

    /**
     * Tests whether the list is empty.
     *
     * @return true iff the list is empty.
     */
    public boolean isEmpty() {
        if (lock.isHeldByCurrentThread()) {
            return first == null;
        }
        lock.acquire();
        Boolean emptyBool = first == null;
        lock.release();

        return emptyBool;
    }

    /**
     * returns number of items in list
     * @return
     */
    public int size() {
        lock.acquire();
```



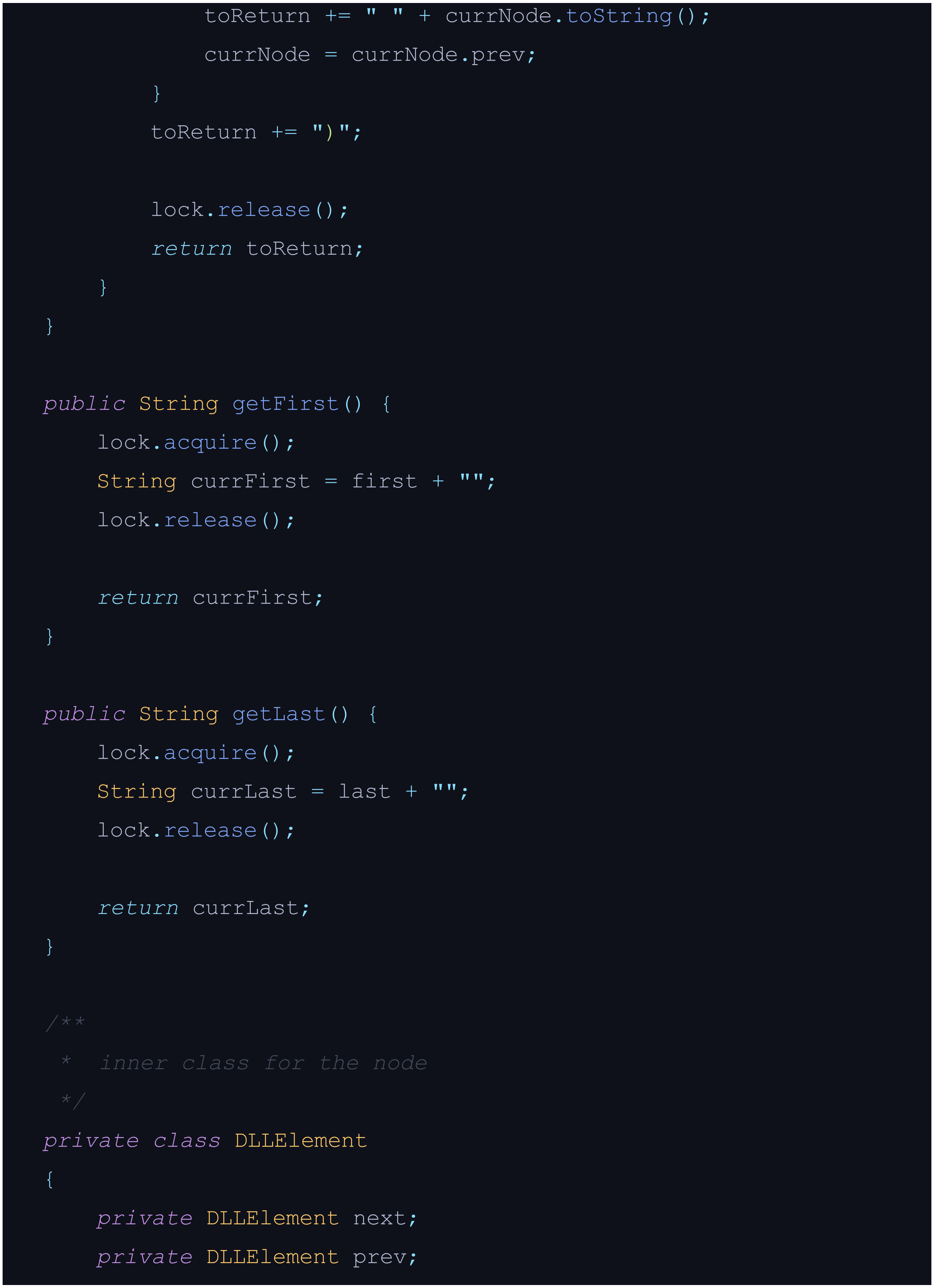
```
        int currSize = size;
        lock.release();

        return currSize;
    }


    /**
     * Inserts item into the list in sorted order according to
sortKey.
     */
    public void insert(Object item, Integer sortKey) {
        DLLElement newNode = new DLLElement(item, sortKey);

        lock.acquire();
        if (isEmpty()) {
            last = newNode;
            first = newNode;

        } else if (first.key > sortKey) {
            first.prev = newNode;
            newNode.next = first;
            first = newNode;

        } else {
            if (sortKey >= last.key) {
                last.next = newNode;
                newNode.prev = last;
                last = newNode;
            } else {

                DLLElement currNode = first;
                DLLElement prevNode = first.prev;
```



```
                while(!(currNode == null) && currNode.key < sortKey) {
                    prevNode = currNode;
                    currNode = currNode.next;
                }

                prevNode.next = newNode;
                newNode.next = currNode;
                newNode.prev = prevNode;
                currNode.prev = newNode;
            }
        }

        size += 1;

        fullList.wake();
        lock.release();

    }


    /**
     * returns list as a printable string. A single space should
separate each list item,
     * and the entire list should be enclosed in parentheses. Empty
list should return "()"
     * @return list elements in order
     */
    public String toString() {
        lock.acquire();

        if (isEmpty()) {
            lock.release();
            return "()";
```



```
        } else {
            String toReturn = "(" + first.toString();
            DLLElement currNode = first.next;
            while(currNode != null) {
                toReturn += " " + currNode.toString();
                currNode = currNode.next;
            }
            toReturn += ")";
            lock.release();

            return toReturn;
        }



    }

    /**
     * returns list as a printable string, from the last node to the
first.
     * String should be formatted just like in toString.
     * @return list elements in backwards order
     */
    public String reverseToString(){
        lock.acquire();

        if (isEmpty()) {
            lock.release();
            return "()";
        } else {
            String toReturn = "(" + last.toString();
            DLLElement currNode = last.prev;
            while(currNode != null) {
```



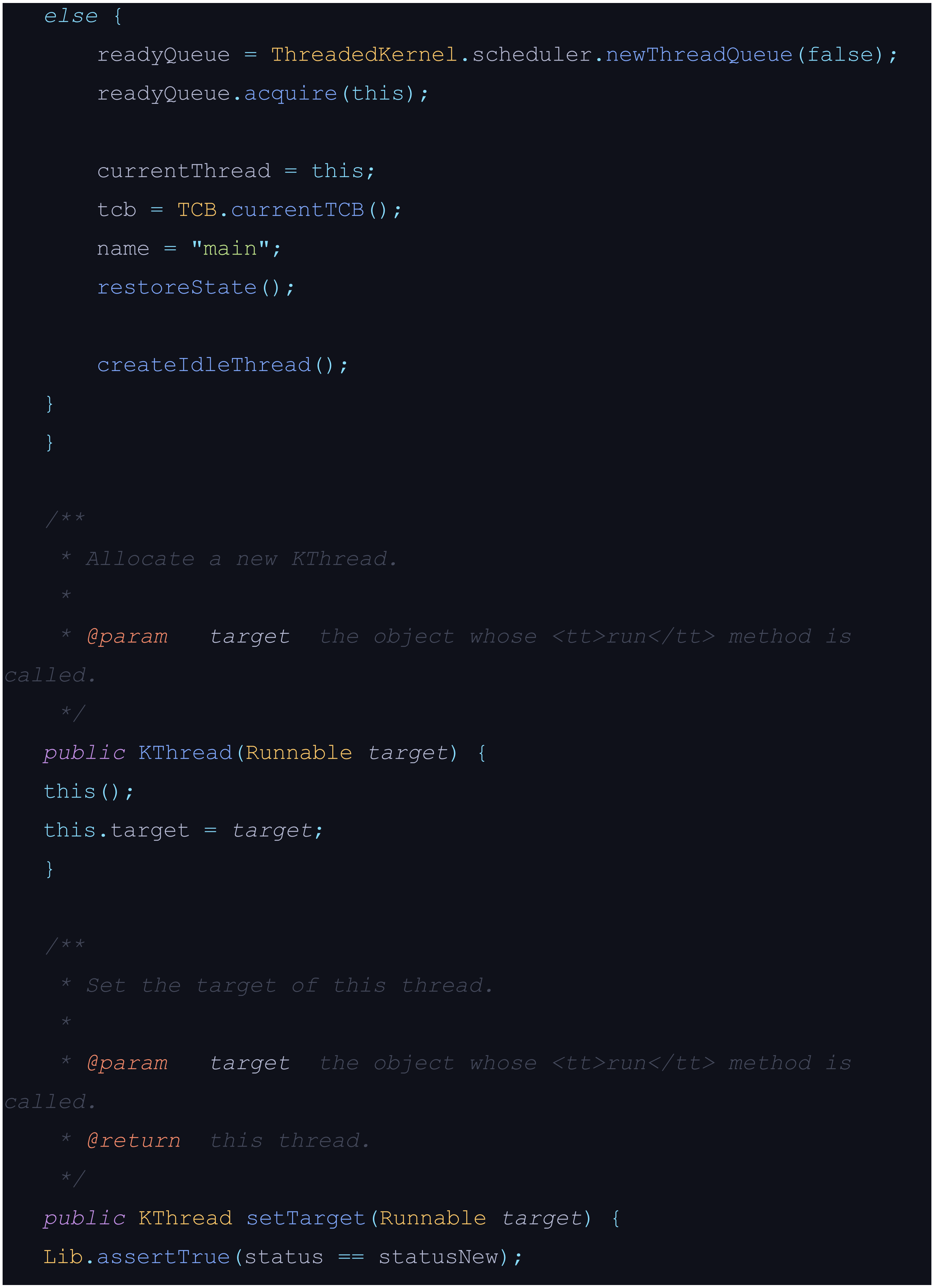
```
            toReturn += " " + currNode.toString();
            currNode = currNode.prev;
        }
        toReturn += ")";

        lock.release();
        return toReturn;
    }
}

public String getFirst() {
    lock.acquire();
    String currFirst = first + "";
    lock.release();

    return currFirst;
}

public String getLast() {
    lock.acquire();
    String currLast = last + "";
    lock.release();

    return currLast;
}

/**
 *  inner class for the node
 */
private class DLLElement
{
    private DLLElement next;
    private DLLElement prev;
```

```
        private int key;
        private Object data;

        /**
         * Node constructor
         * @param item data item to store
         * @param sortKey unique integer ID
         */
        public DLLElement(Object item, int sortKey)
        {
            key = sortKey;
            data = item;
            next = null;
            prev = null;
        }

        /**
         * returns node contents as a printable string
         * @return string of form [<key>,<data>] such as [3,"ham"]
         */
        public String toString(){
            return "[" + key + "," + data + "]";
        }
    }
}
```

KThread:

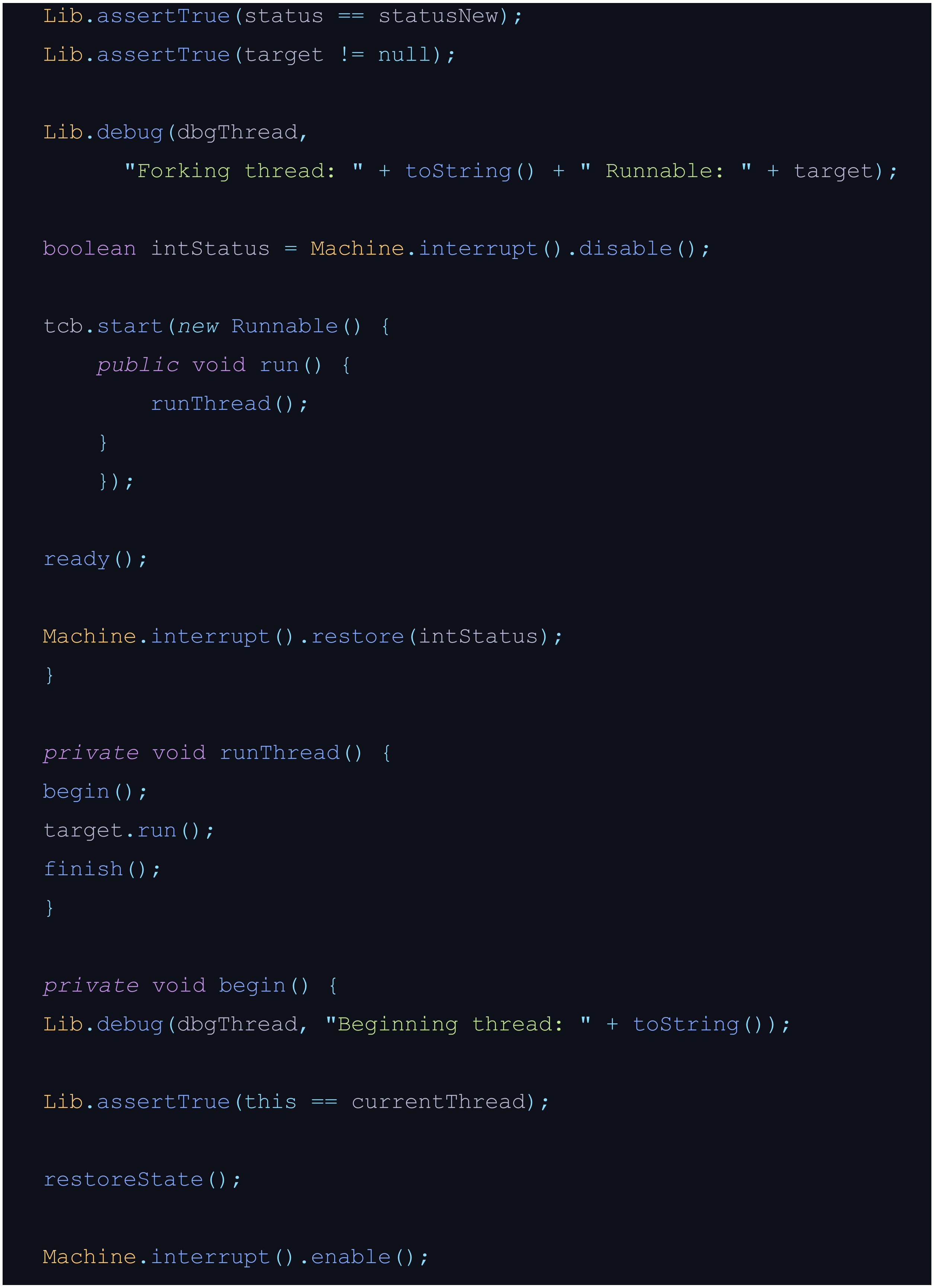
```
package nachos.threads;

import nachos.machine.*;
```

```java
/**
* A KThread is a thread that can be used to execute Nachos kernel
code. Nachos
* allows multiple threads to run concurrently.
*
* To create a new thread of execution, first declare a class that
implements
* the <tt>Runnable</tt> interface. That class then implements the
<tt>run</tt>
* method. An instance of the class can then be allocated, passed as
an
* argument when creating <tt>KThread</tt>, and forked. For example,
a thread
* that computes pi could be written as follows:
*
* <p><blockquote><pre>
* class PiRun implements Runnable {
*     public void run() {
*         // compute pi
*         ...
*     }
* }
* </pre></blockquote>
* <p>The following code would then create a thread and start it
running:
*
* <p><blockquote><pre>
* PiRun p = new PiRun();
* new KThread(p).fork();
* </pre></blockquote>
*/
public class KThread {
   // instance variables

   static int numTimesBefore = 0;
   // part 1: ascending numbers interleaving
   //static boolean[] oughtToYield =
{true,true,true,true,true,true,true,true,true,true,true,true};
   // part 1: a's and b's grouped by 2's
   static boolean[] oughtToYield =
{false,true,false,true,false,true,false,true,false,true,false,true};

   // interleaving yield variables
   static boolean[][] yieldData;
   static int[] yieldCount;

   /**
    * Get the current thread.
    *
    * @return  the current thread.
    */
   public static KThread currentThread() {
   Lib.assertTrue(currentThread != null);
   return currentThread;
   }

   /**
    * Allocate a new <tt>KThread</tt>. If this is the first
<tt>KThread</tt>,
    * create an idle thread as well.
    */
   public KThread() {
   if (currentThread != null) {
       tcb = new TCB();
   }
```



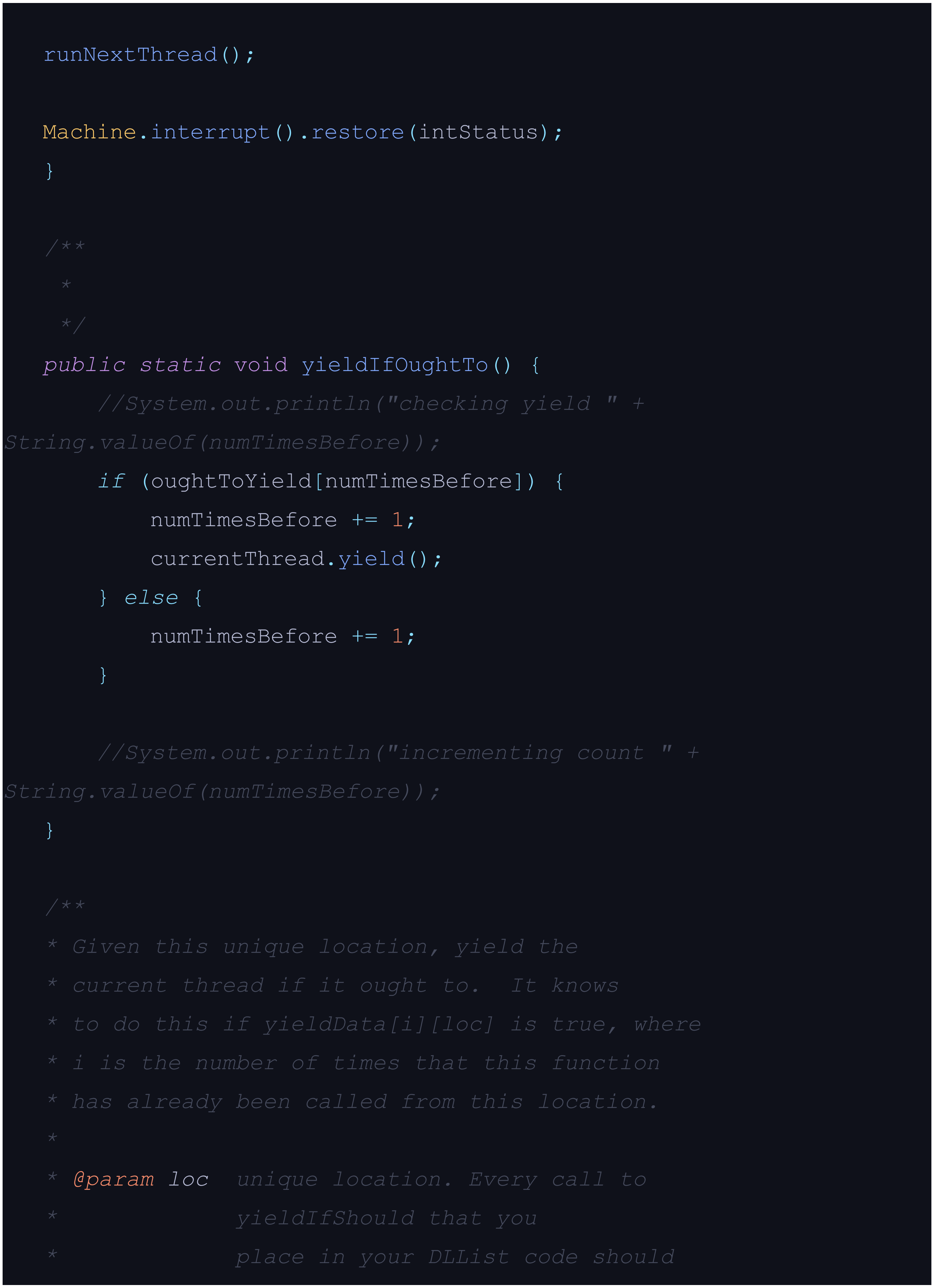
```
   else {
       readyQueue = ThreadedKernel.scheduler.newThreadQueue(false);
       readyQueue.acquire(this);

       currentThread = this;
       tcb = TCB.currentTCB();
       name = "main";
       restoreState();

       createIdleThread();
   }
   }

   /**
    * Allocate a new KThread.
    *
    * @param   target  the object whose <tt>run</tt> method is
called.
    */
   public KThread(Runnable target) {
   this();
   this.target = target;
   }

   /**
    * Set the target of this thread.
    *
    * @param   target  the object whose <tt>run</tt> method is
called.
    * @return  this thread.
    */
   public KThread setTarget(Runnable target) {
   Lib.assertTrue(status == statusNew);
   this.target = target;
   return this;
   }

   /**
    * Set the name of this thread. This name is used for debugging
purposes
    * only.
    *
    * @param   name    the name to give to this thread.
    * @return  this thread.
    */
   public KThread setName(String name) {
   this.name = name;
   return this;
   }

   /**
    * Get the name of this thread. This name is used for debugging
purposes
    * only.
    *
    * @return  the name given to this thread.
    */
   public String getName() {
   return name;
   }

   /**
    * Get the full name of this thread. This includes its name
along with its
    * numerical ID. This name is used for debugging purposes only.
```

```
    *
    * @return  the full name given to this thread.
    */
   public String toString() {
   return (name + " (#" + id + ")");
   }

   /**
    * Deterministically and consistently compare this thread to
another
    * thread.
    */
   public int compareTo(Object o) {
   KThread thread = (KThread) o;

   if (id < thread.id)
       return -1;
   else if (id > thread.id)
       return 1;
   else
       return 0;
   }

   /**
    * Causes this thread to begin execution. The result is that two
threads
    * are running concurrently: the current thread (which returns
from the
    * call to the <tt>fork</tt> method) and the other thread (which
executes
    * its target's <tt>run</tt> method).
    */
   public void fork() {
```



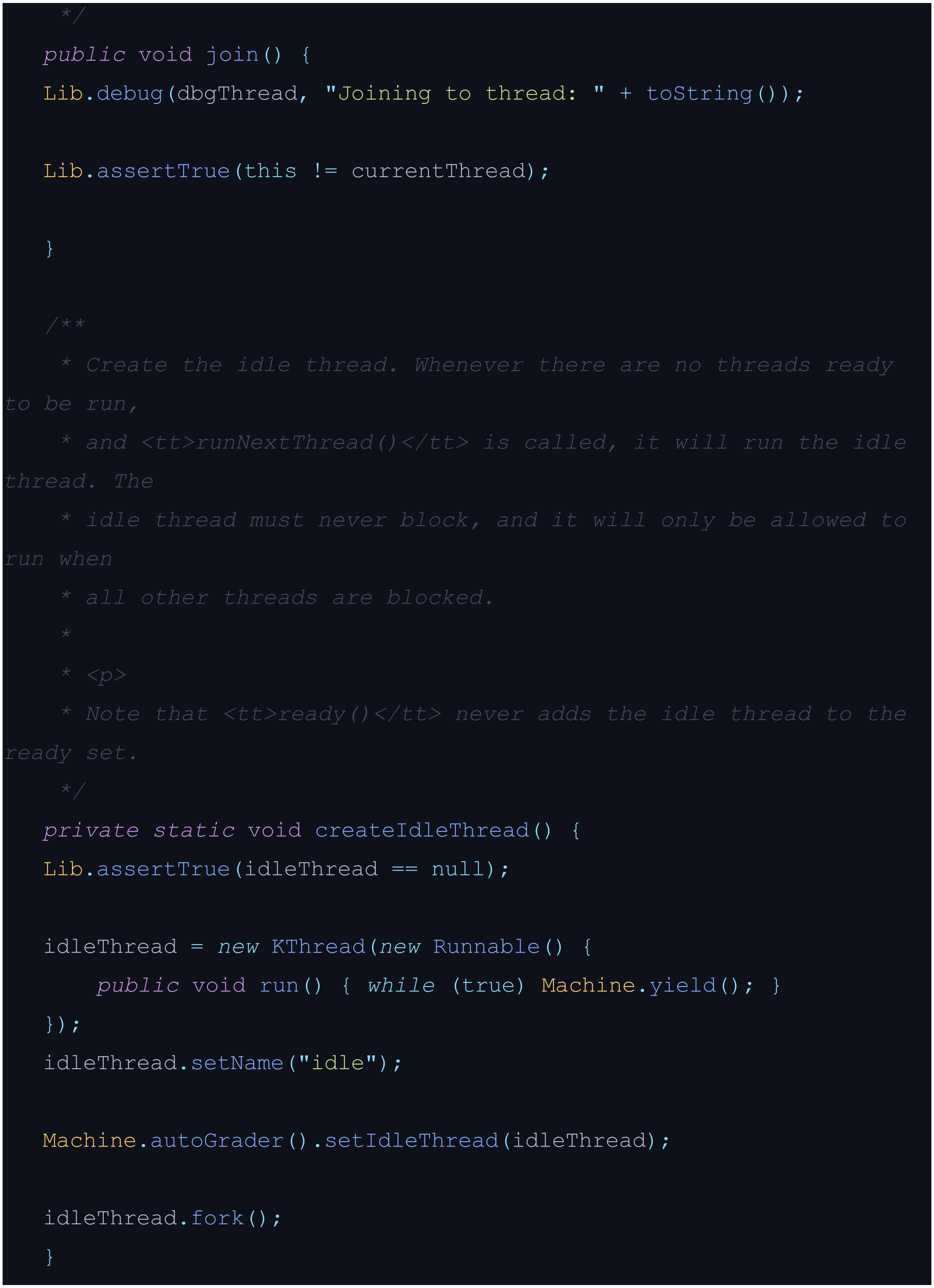
```
Lib.assertTrue(status == statusNew);
Lib.assertTrue(target != null);

Lib.debug(dbgThread,
      "Forking thread: " + toString() + " Runnable: " + target);

boolean intStatus = Machine.interrupt().disable();

tcb.start(new Runnable() {
    public void run() {
        runThread();
    }
    });

ready();

Machine.interrupt().restore(intStatus);
}

private void runThread() {
begin();
target.run();
finish();
}

private void begin() {
Lib.debug(dbgThread, "Beginning thread: " + toString());

Lib.assertTrue(this == currentThread);

restoreState();

Machine.interrupt().enable();
```

```
    }

    /**
     * Finish the current thread and schedule it to be destroyed
when it is
     * safe to do so. This method is automatically called when a
thread's
     * <tt>run</tt> method returns, but it may also be called
directly.
     *
     * The current thread cannot be immediately destroyed because
its stack and
     * other execution state are still in use. Instead, this thread
will be
     * destroyed automatically by the next thread to run, when it is
safe to
     * delete this thread.
     */
    public static void finish() {
    Lib.debug(dbgThread, "Finishing thread: " +
currentThread.toString());

    Machine.interrupt().disable();

    Machine.autoGrader().finishingCurrentThread();

    Lib.assertTrue(toBeDestroyed == null);
    toBeDestroyed = currentThread;


    currentThread.status = statusFinished;

    sleep();
    }

    /**
     * Relinquish the CPU if any other thread is ready to run. If
so, put the
     * current thread on the ready queue, so that it will eventually
be
     * rescheuled.
     *
     * <p>
     * Returns immediately if no other thread is ready to run.
Otherwise
     * returns when the current thread is chosen to run again by
     * <tt>readyQueue.nextThread()</tt>.
     *
     * <p>
     * Interrupts are disabled, so that the current thread can
atomically add
     * itself to the ready queue and switch to the next thread. On
return,
     * restores interrupts to the previous state, in case
<tt>yield()</tt> was
     * called with interrupts disabled.
     */
   public static void yield() {
   Lib.debug(dbgThread, "Yielding thread: " +
currentThread.toString());

   Lib.assertTrue(currentThread.status == statusRunning);

   boolean intStatus = Machine.interrupt().disable();

   currentThread.ready();
```

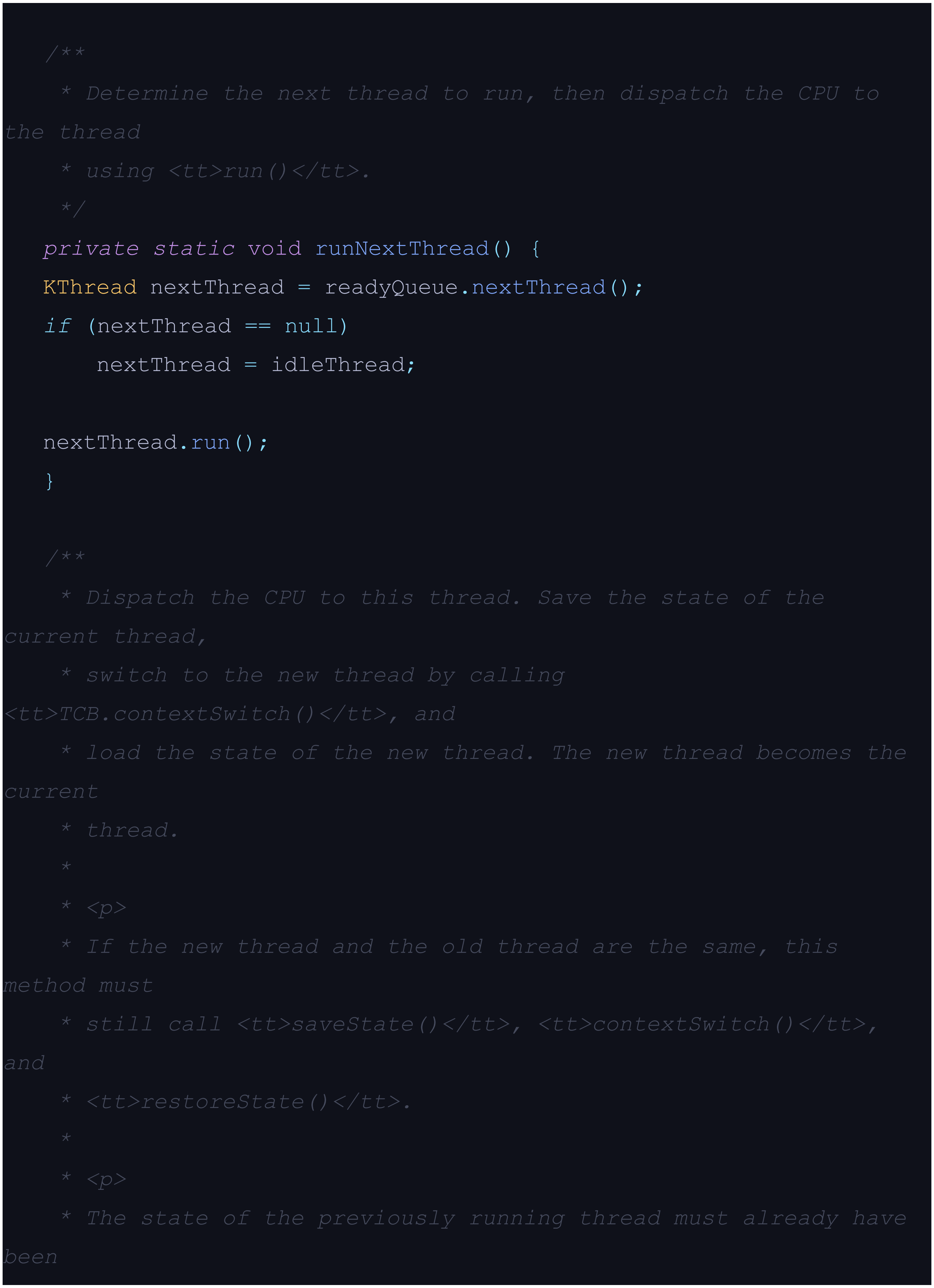


```
    runNextThread();

    Machine.interrupt().restore(intStatus);
    }

    /**
     *
     */
    public static void yieldIfOughtTo() {
        //System.out.println("checking yield " +
String.valueOf(numTimesBefore));
        if (oughtToYield[numTimesBefore]) {
            numTimesBefore += 1;
            currentThread.yield();
        } else {
            numTimesBefore += 1;
        }

        //System.out.println("incrementing count " +
String.valueOf(numTimesBefore));
    }

    /**
    * Given this unique location, yield the
    * current thread if it ought to.  It knows
    * to do this if yieldData[i][loc] is true, where
    * i is the number of times that this function
    * has already been called from this location.
    *
    * @param loc  unique location. Every call to
    *             yieldIfShould that you
    *             place in your DLList code should
```

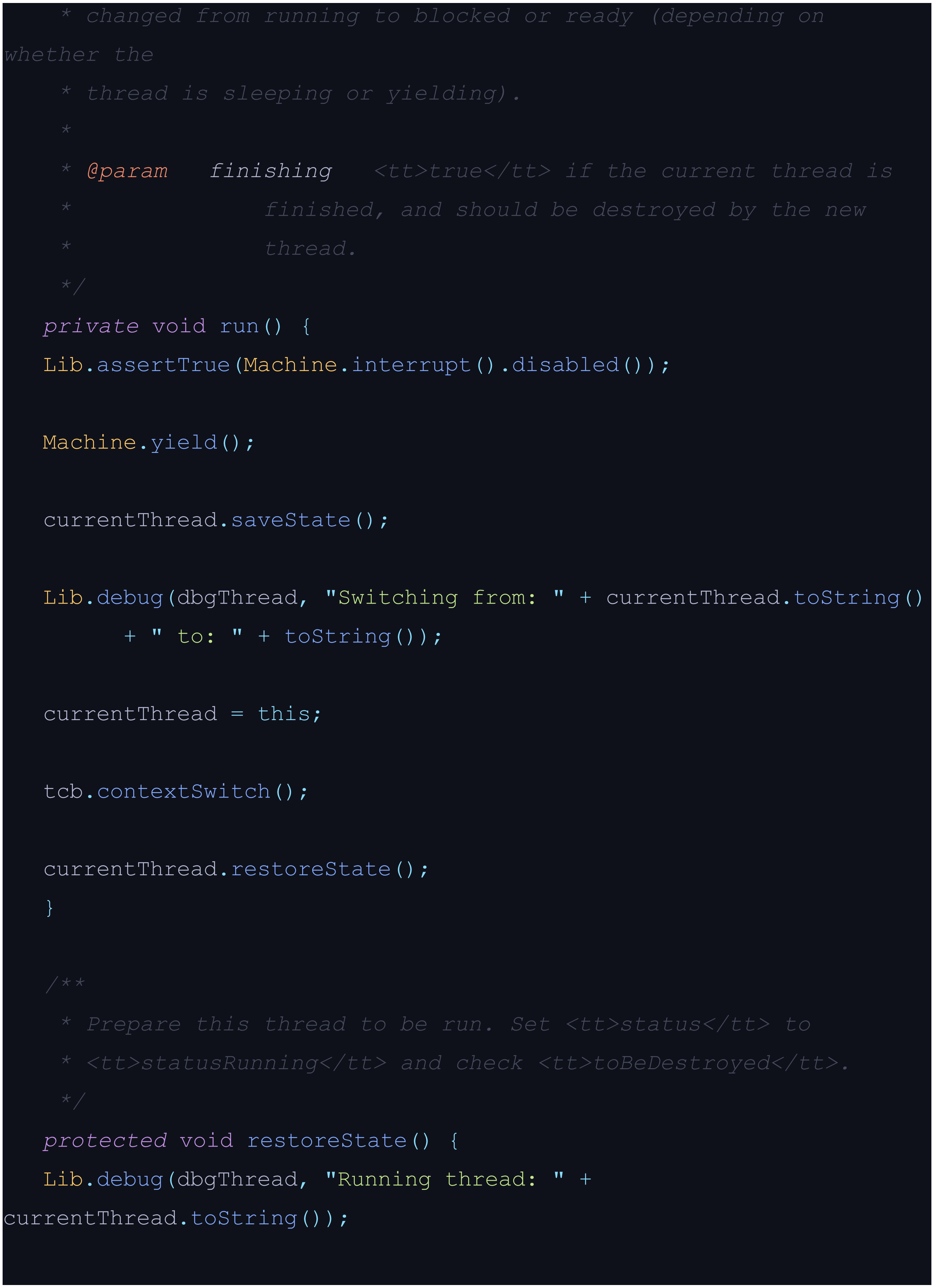
```
   *             have a different loc number.
   */
   public static void yieldIfShould(int loc) {
       if (KThread.yieldData[loc][KThread.yieldCount[loc]]) {
           KThread.yieldCount[loc] += 1;
           currentThread.yield();
       } else {
           KThread.yieldCount[loc] += 1;
       }
   }

   /**
    * Relinquish the CPU, because the current thread has either
finished or it
    * is blocked. This thread must be the current thread.
    *
    * <p>
    * If the current thread is blocked (on a synchronization
primitive, i.e.
    * a <tt>Semaphore</tt>, <tt>Lock</tt>, or <tt>Condition</tt>),
eventually
    * some thread will wake this thread up, putting it back on the
ready queue
    * so that it can be rescheduled. Otherwise, <tt>finish()</tt>
should have
    * scheduled this thread to be destroyed by the next thread to
run.
    */
   public static void sleep() {
   Lib.debug(dbgThread, "Sleeping thread: " +
currentThread.toString());

   Lib.assertTrue(Machine.interrupt().disabled());
if (currentThread.status != statusFinished)
    currentThread.status = statusBlocked;

runNextThread();
}

/**
 * Moves this thread to the ready state and adds this to the
scheduler's
 * ready queue.
 */
public void ready() {
Lib.debug(dbgThread, "Ready thread: " + toString());

Lib.assertTrue(Machine.interrupt().disabled());
Lib.assertTrue(status != statusReady);

status = statusReady;
if (this != idleThread)
    readyQueue.waitForAccess(this);

Machine.autoGrader().readyThread(this);
}

/**
 * Waits for this thread to finish. If this thread is already
finished,
 * return immediately. This method must only be called once; the
second
 * call is not guaranteed to return. This thread must not be the
current
 * thread.
```



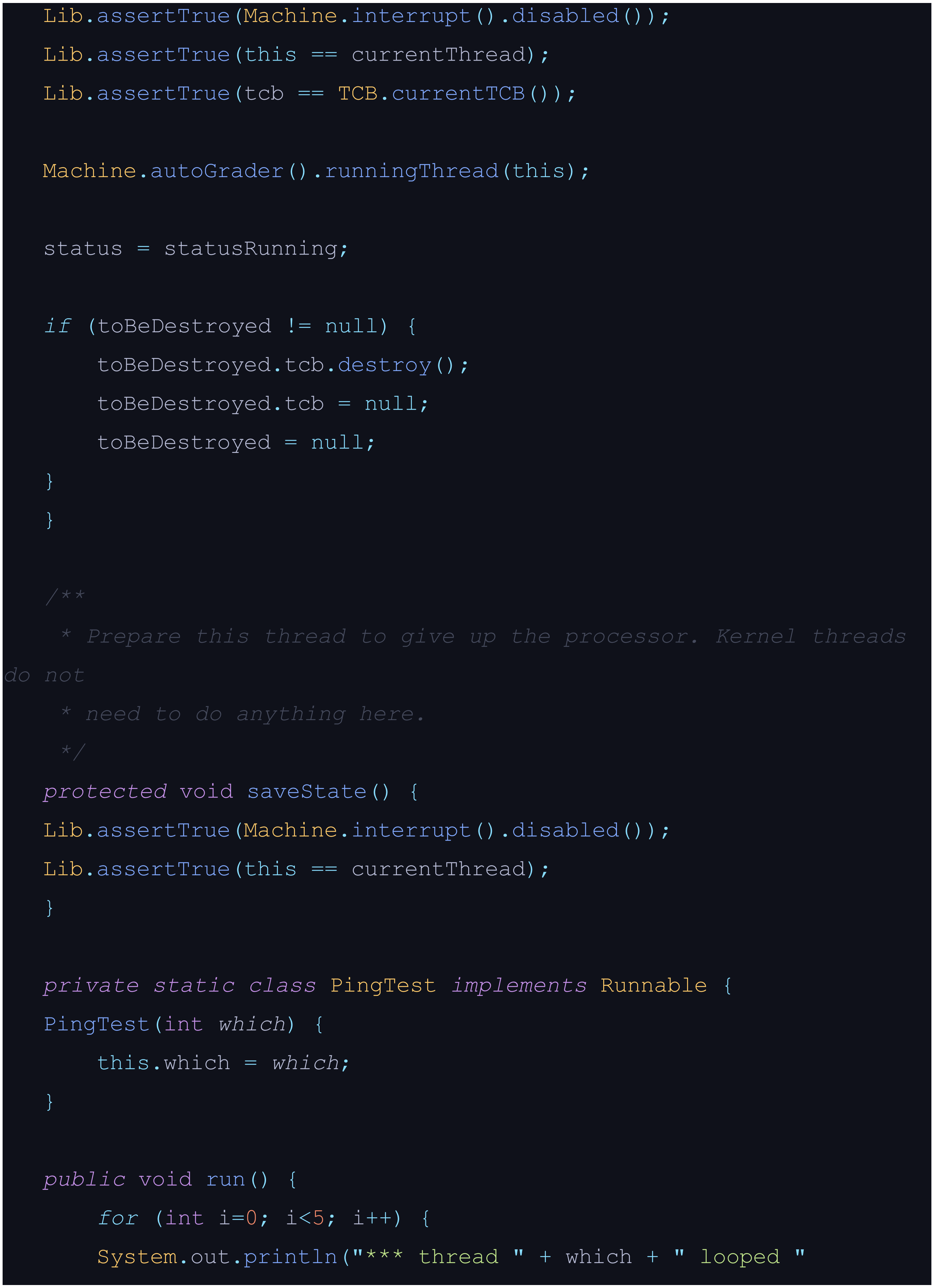
```
    */
   public void join() {
   Lib.debug(dbgThread, "Joining to thread: " + toString());

   Lib.assertTrue(this != currentThread);

   }

   /**
    * Create the idle thread. Whenever there are no threads ready
to be run,
    * and <tt>runNextThread()</tt> is called, it will run the idle
thread. The
    * idle thread must never block, and it will only be allowed to
run when
    * all other threads are blocked.
    *
    * <p>
    * Note that <tt>ready()</tt> never adds the idle thread to the
ready set.
    */
   private static void createIdleThread() {
   Lib.assertTrue(idleThread == null);

   idleThread = new KThread(new Runnable() {
       public void run() { while (true) Machine.yield(); }
   });
   idleThread.setName("idle");

   Machine.autoGrader().setIdleThread(idleThread);

   idleThread.fork();
   }
```



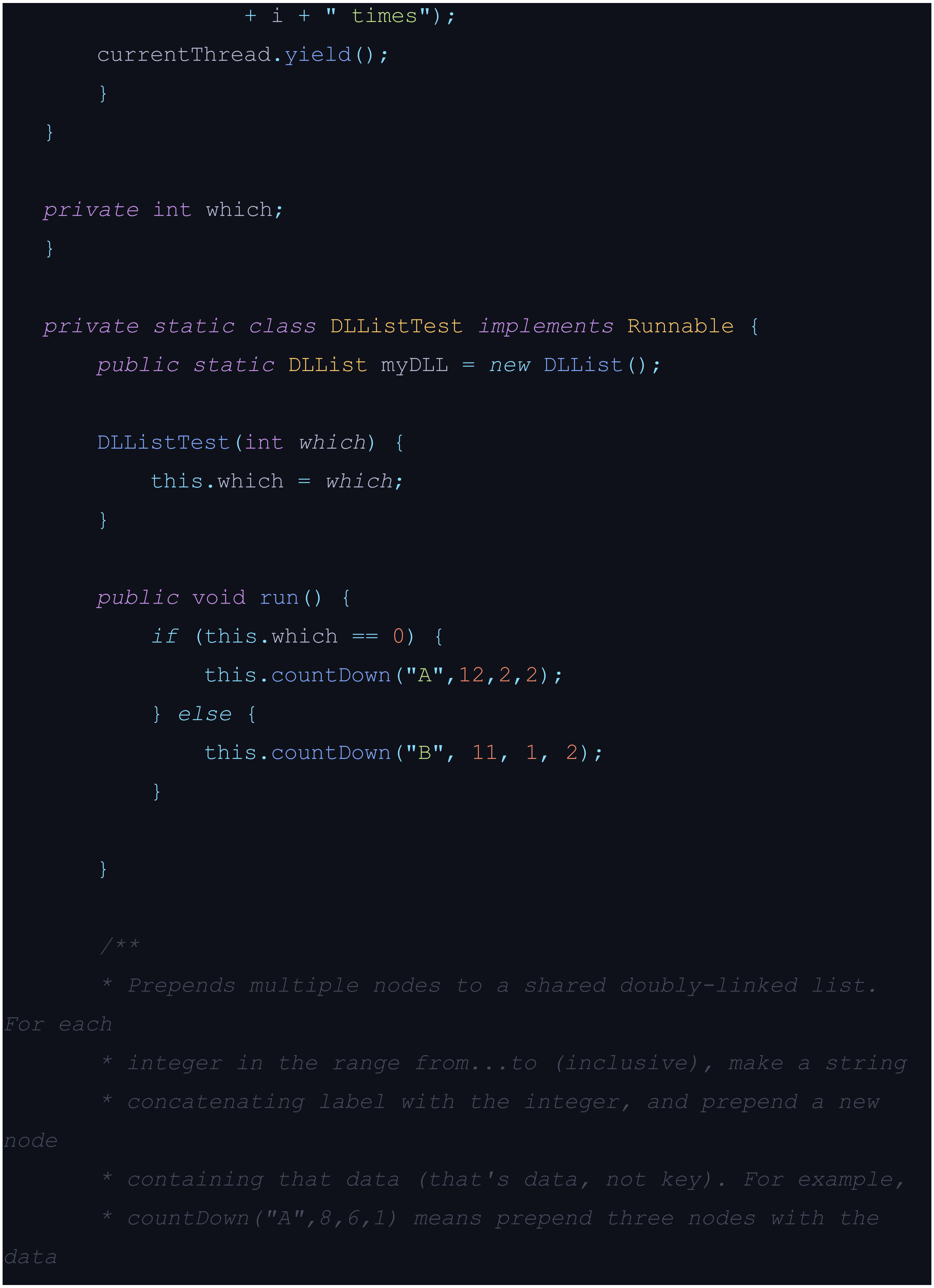
```
   /**
    * Determine the next thread to run, then dispatch the CPU to
the thread
    * using <tt>run()</tt>.
    */
   private static void runNextThread() {
   KThread nextThread = readyQueue.nextThread();
   if (nextThread == null)
       nextThread = idleThread;

   nextThread.run();
   }

   /**
    * Dispatch the CPU to this thread. Save the state of the
current thread,
    * switch to the new thread by calling
<tt>TCB.contextSwitch()</tt>, and
    * load the state of the new thread. The new thread becomes the
current
    * thread.
    *
    * <p>
    * If the new thread and the old thread are the same, this
method must
    * still call <tt>saveState()</tt>, <tt>contextSwitch()</tt>,
and
    * <tt>restoreState()</tt>.
    *
    * <p>
    * The state of the previously running thread must already have
been
```



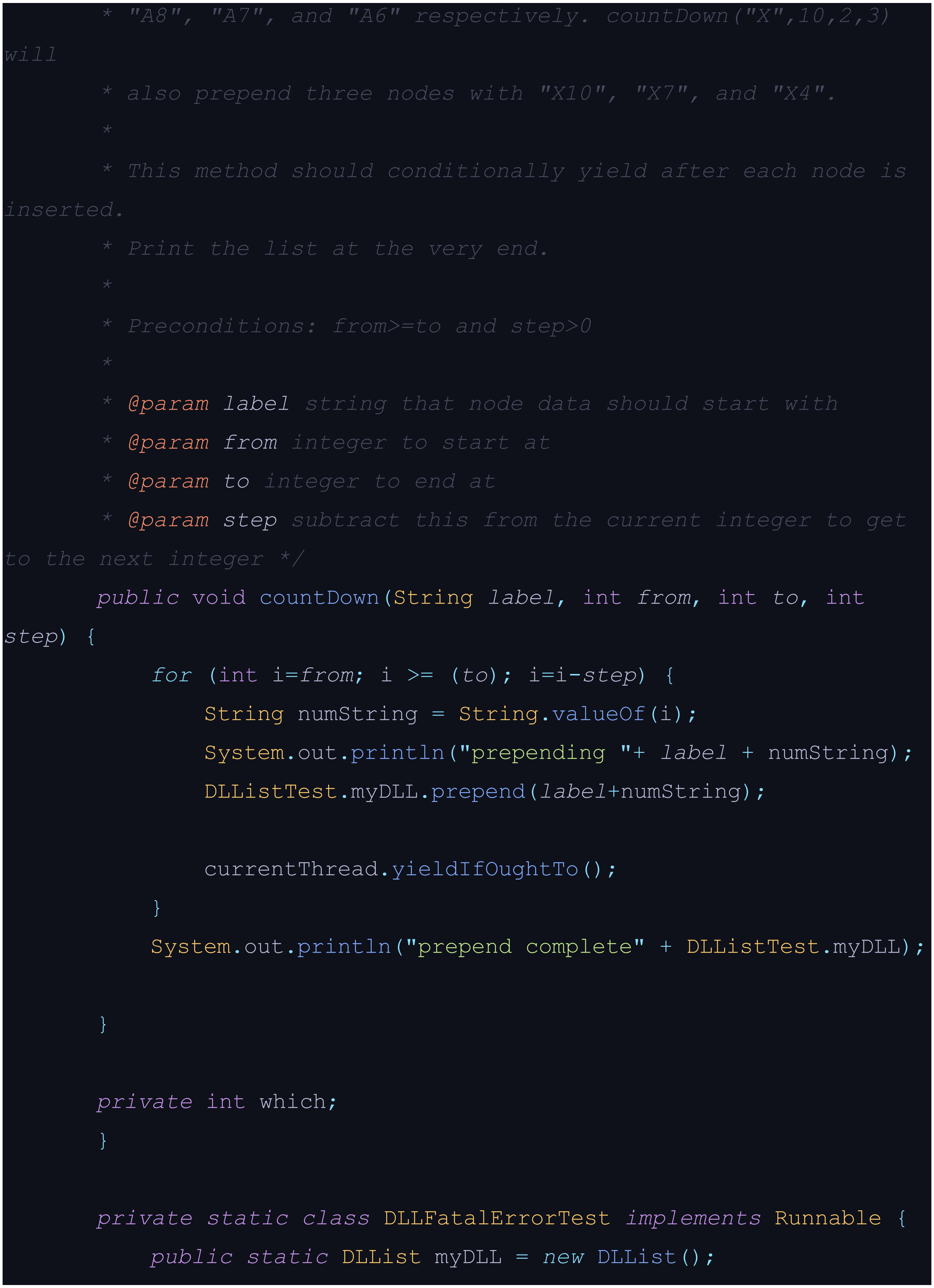
```
     * changed from running to blocked or ready (depending on
whether the
     * thread is sleeping or yielding).
     *
     * @param   finishing   <tt>true</tt> if the current thread is
     *                finished, and should be destroyed by the new
     *                thread.
     */
   private void run() {
   Lib.assertTrue(Machine.interrupt().disabled());

   Machine.yield();

   currentThread.saveState();

   Lib.debug(dbgThread, "Switching from: " + currentThread.toString()
         + " to: " + toString());

   currentThread = this;

   tcb.contextSwitch();

   currentThread.restoreState();
   }

   /**
     * Prepare this thread to be run. Set <tt>status</tt> to
     * <tt>statusRunning</tt> and check <tt>toBeDestroyed</tt>.
     */
   protected void restoreState() {
   Lib.debug(dbgThread, "Running thread: " +
currentThread.toString());
```



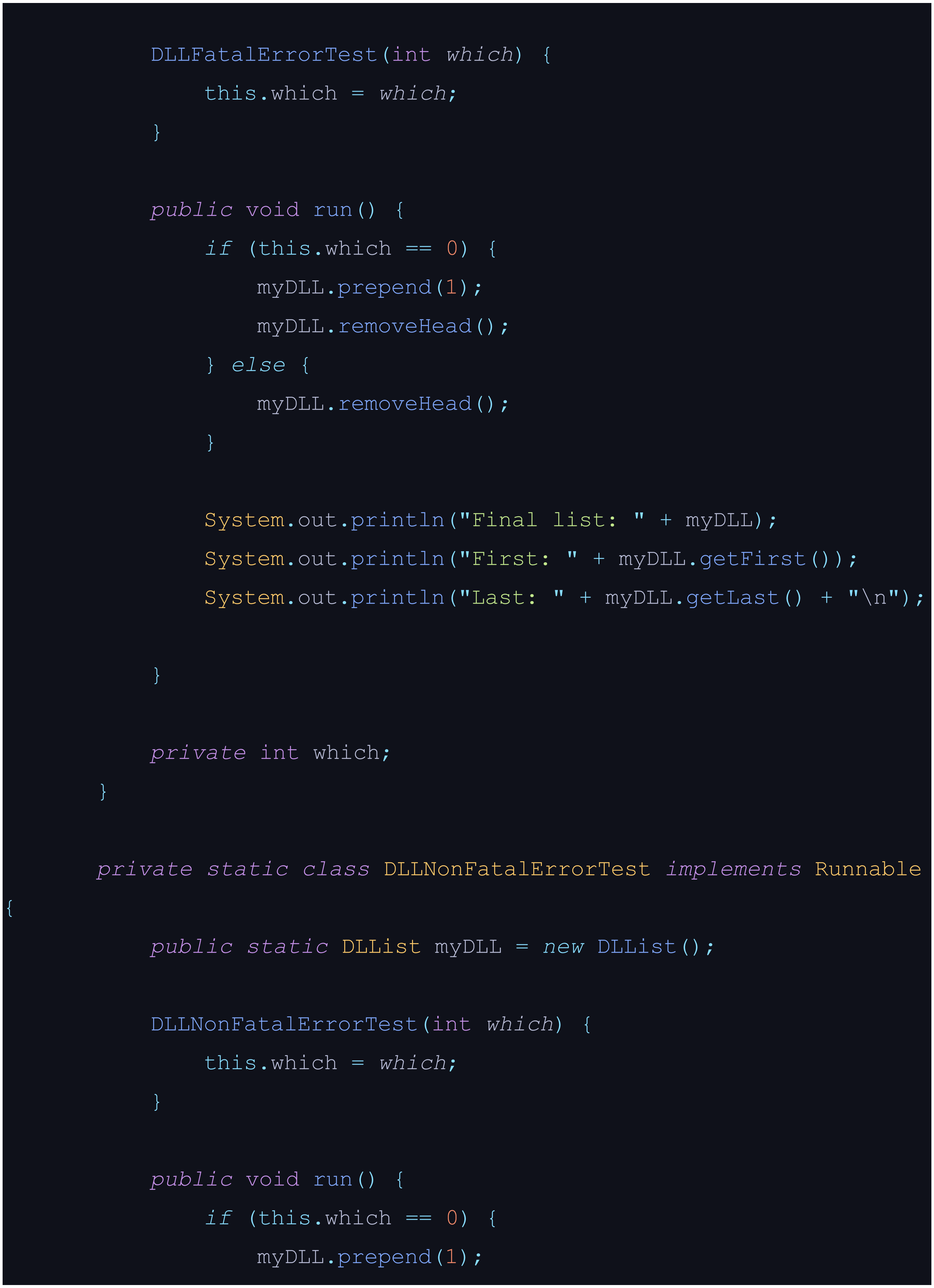
```
    Lib.assertTrue(Machine.interrupt().disabled());
    Lib.assertTrue(this == currentThread);
    Lib.assertTrue(tcb == TCB.currentTCB());

    Machine.autoGrader().runningThread(this);

    status = statusRunning;

    if (toBeDestroyed != null) {
        toBeDestroyed.tcb.destroy();
        toBeDestroyed.tcb = null;
        toBeDestroyed = null;
    }
    }

    /**
     * Prepare this thread to give up the processor. Kernel threads
do not
     * need to do anything here.
     */
    protected void saveState() {
    Lib.assertTrue(Machine.interrupt().disabled());
    Lib.assertTrue(this == currentThread);
    }

    private static class PingTest implements Runnable {
    PingTest(int which) {
        this.which = which;
    }

    public void run() {
        for (int i=0; i<5; i++) {
        System.out.println("*** thread " + which + " looped "
```



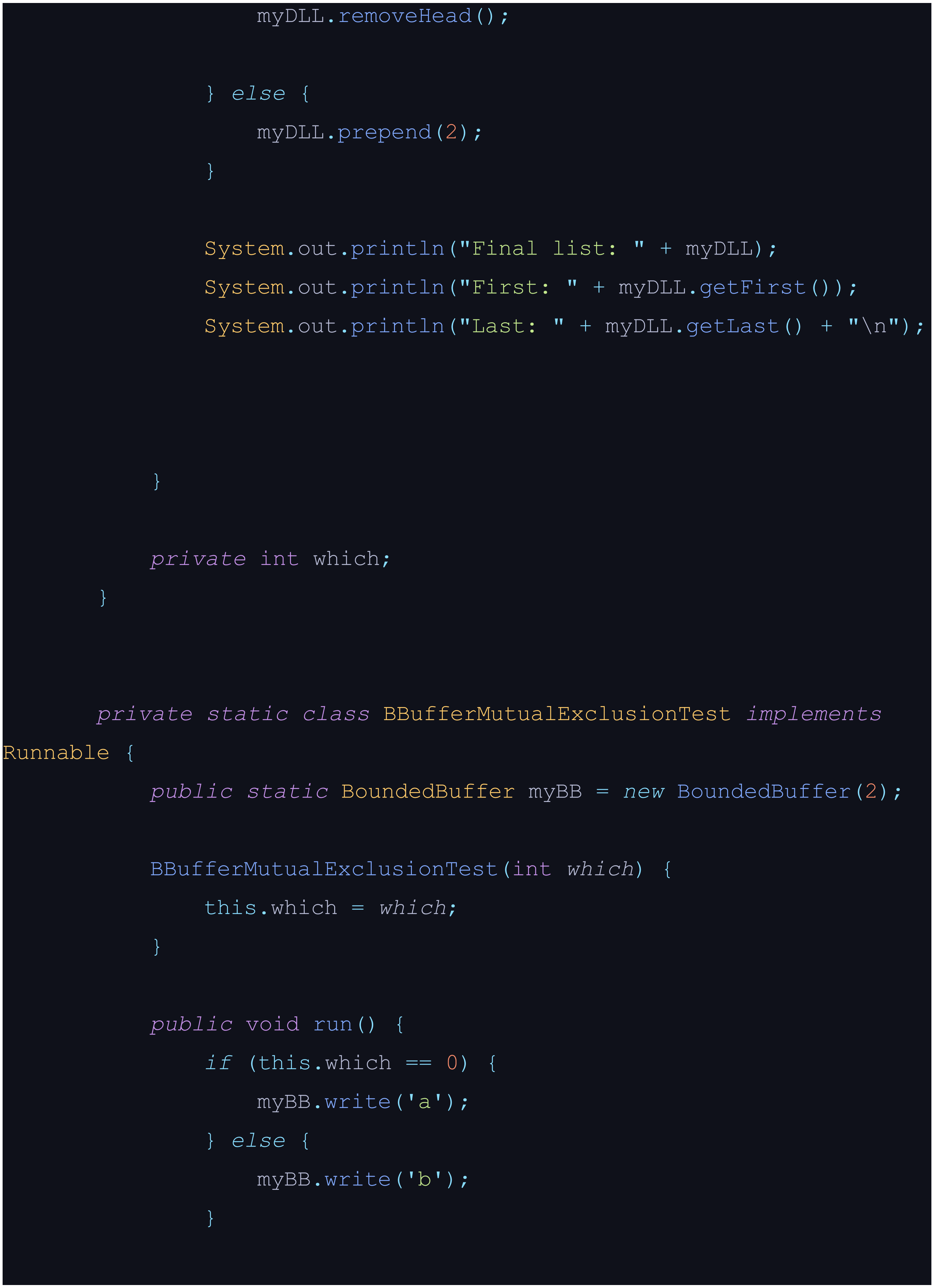
```
                  + i + " times");
       currentThread.yield();
       }
   }

   private int which;
   }

   private static class DLListTest implements Runnable {
       public static DLList myDLL = new DLList();

       DLListTest(int which) {
           this.which = which;
       }

       public void run() {
           if (this.which == 0) {
               this.countDown("A",12,2,2);
           } else {
               this.countDown("B", 11, 1, 2);
           }

       }

       /**
       * Prepends multiple nodes to a shared doubly-linked list.
For each
       * integer in the range from...to (inclusive), make a string
       * concatenating label with the integer, and prepend a new
node
       * containing that data (that's data, not key). For example,
       * countDown("A",8,6,1) means prepend three nodes with the
data
```



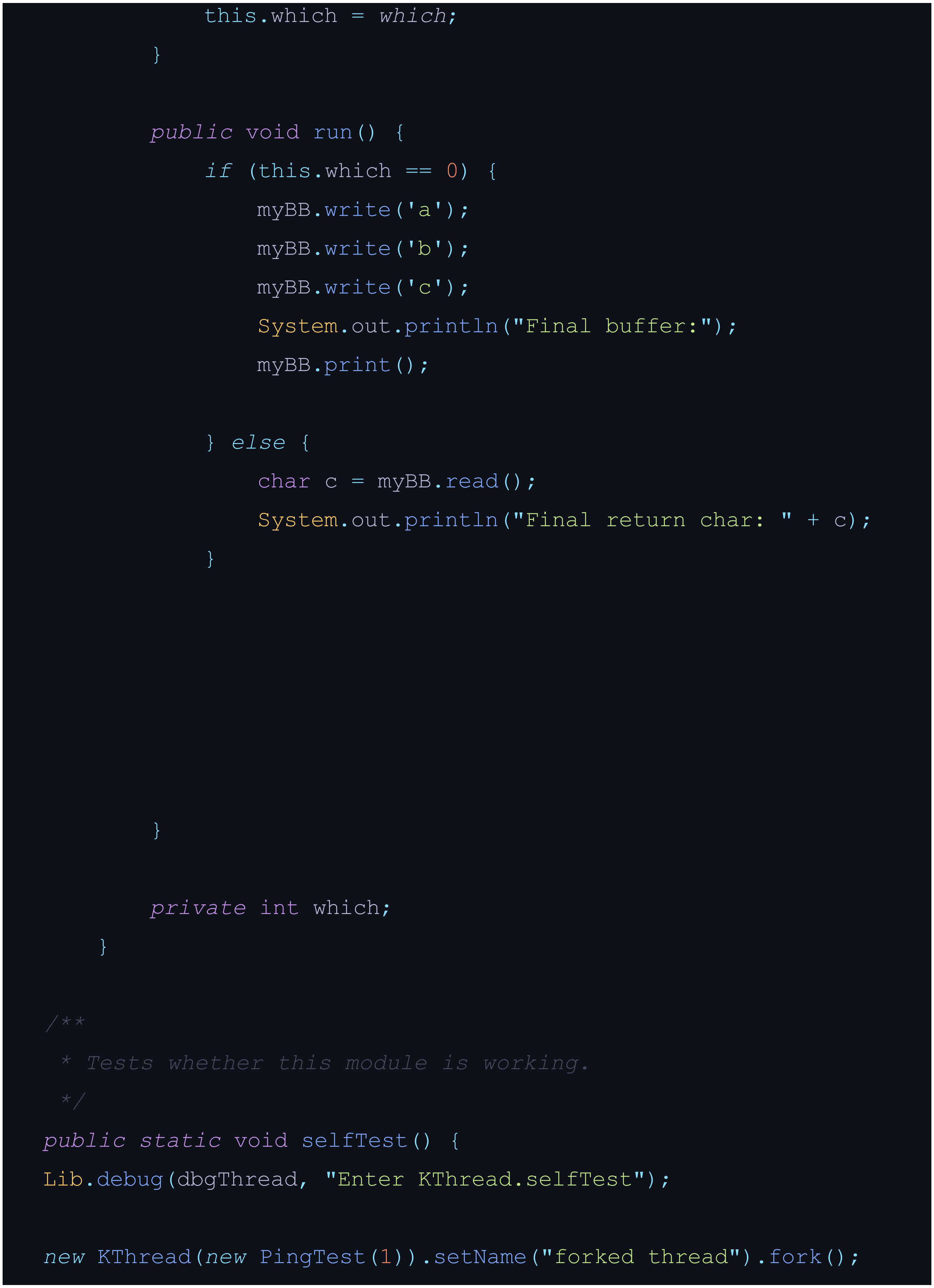
```
        * "A8", "A7", and "A6" respectively. countDown("X",10,2,3)
will
        * also prepend three nodes with "X10", "X7", and "X4".
        *
        * This method should conditionally yield after each node is
inserted.
        * Print the list at the very end.
        *
        * Preconditions: from>=to and step>0
        *
        * @param label string that node data should start with
        * @param from integer to start at
        * @param to integer to end at
        * @param step subtract this from the current integer to get
to the next integer */
        public void countDown(String label, int from, int to, int
step) {
            for (int i=from; i >= (to); i=i-step) {
                String numString = String.valueOf(i);
                System.out.println("prepending "+ label + numString);
                DLListTest.myDLL.prepend(label+numString);

                currentThread.yieldIfOughtTo();
            }
            System.out.println("prepend complete" + DLListTest.myDLL);

        }

        private int which;
        }

        private static class DLLFatalErrorTest implements Runnable {
            public static DLList myDLL = new DLList();
```



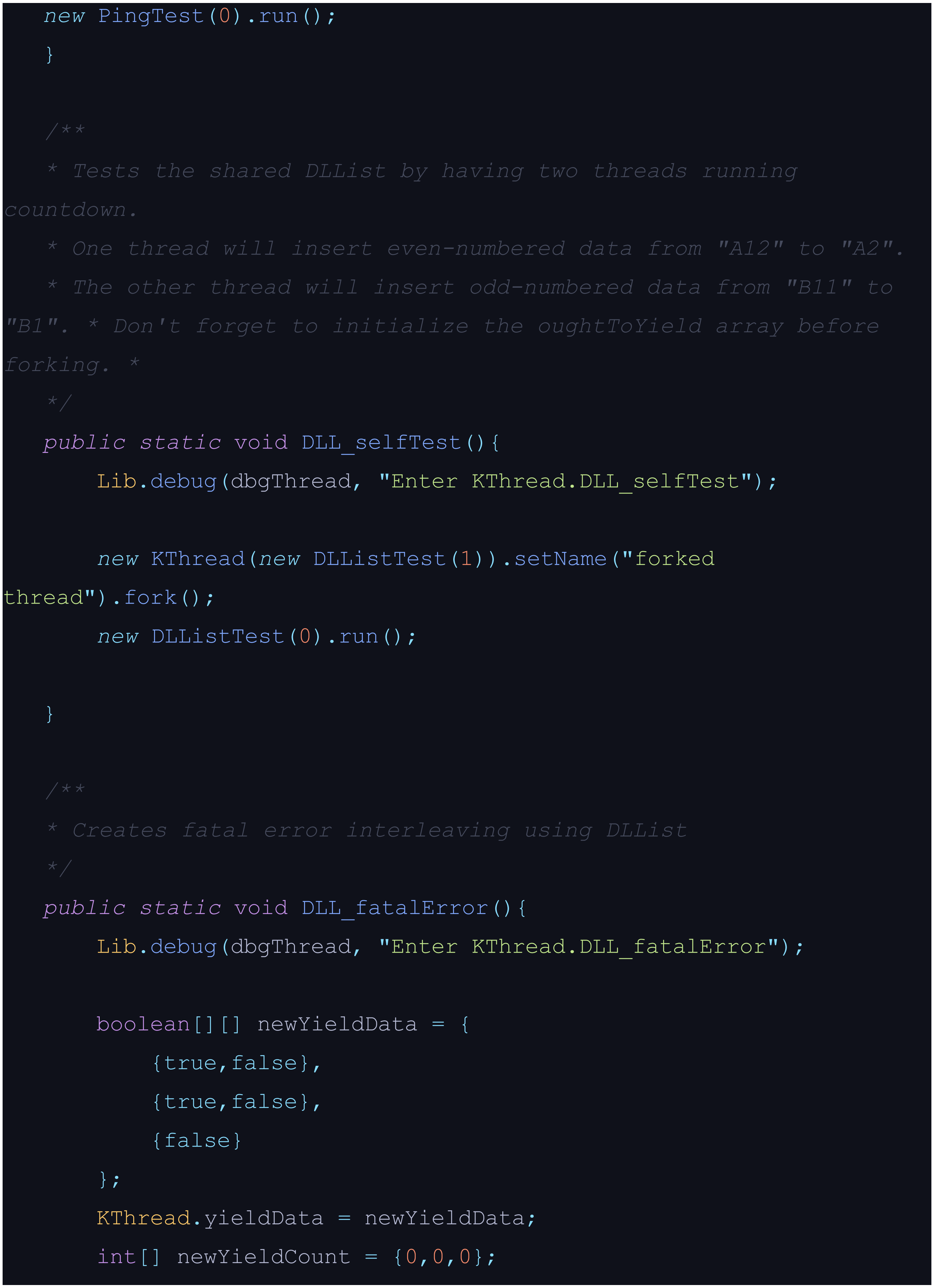
```
        DLLFatalErrorTest(int which) {
            this.which = which;
        }

        public void run() {
            if (this.which == 0) {
                myDLL.prepend(1);
                myDLL.removeHead();
            } else {
                myDLL.removeHead();
            }

            System.out.println("Final list: " + myDLL);
            System.out.println("First: " + myDLL.getFirst());
            System.out.println("Last: " + myDLL.getLast() + "\n");

        }

        private int which;
    }

    private static class DLLNonFatalErrorTest implements Runnable
{
        public static DLList myDLL = new DLList();

        DLLNonFatalErrorTest(int which) {
            this.which = which;
        }

        public void run() {
            if (this.which == 0) {
                myDLL.prepend(1);
```



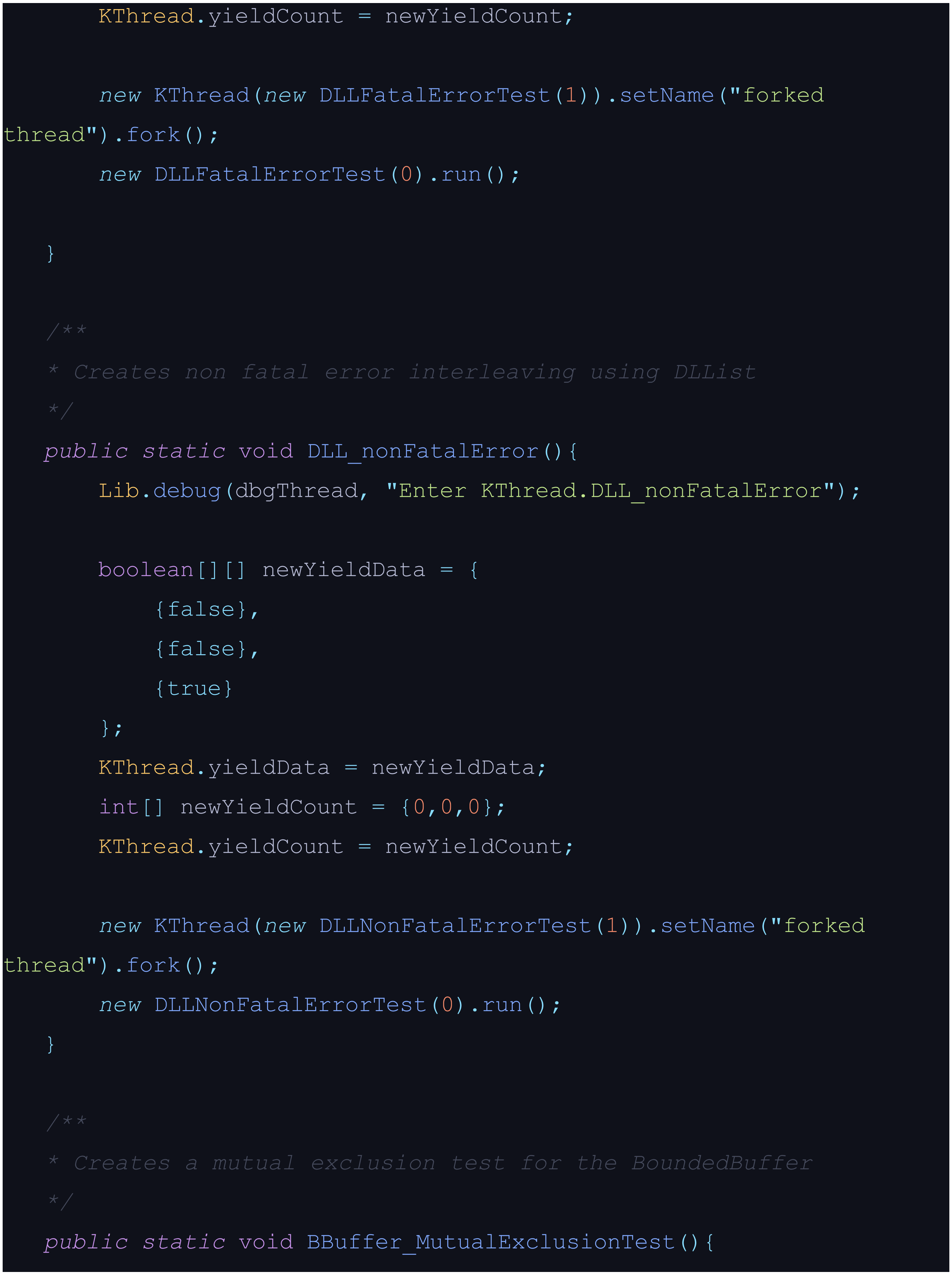
```
                    myDLL.removeHead();

                } else {
                    myDLL.prepend(2);
                }

                System.out.println("Final list: " + myDLL);
                System.out.println("First: " + myDLL.getFirst());
                System.out.println("Last: " + myDLL.getLast() + "\n");



            }

            private int which;
        }


        private static class BBufferMutualExclusionTest implements
Runnable {
            public static BoundedBuffer myBB = new BoundedBuffer(2);

            BBufferMutualExclusionTest(int which) {
                this.which = which;
            }

            public void run() {
                if (this.which == 0) {
                    myBB.write('a');
                } else {
                    myBB.write('b');
                }
            System.out.println("Final buffer:");
            myBB.print();

        }

        private int which;
    }

    private static class BBufferUnderflowTest implements Runnable
{
        public static BoundedBuffer myBB = new BoundedBuffer(2);

        BBufferUnderflowTest(int which) {
            this.which = which;
        }

        public void run() {
            if (this.which == 0) {
                char c = myBB.read();
                System.out.println("Final return char: " + c);
            } else {
                myBB.write('a');
            }
         }

        private int which;
    }

    private static class BBufferOverflowTest implements Runnable
{
        public static BoundedBuffer myBB = new BoundedBuffer(2);

        BBufferOverflowTest(int which) {
```



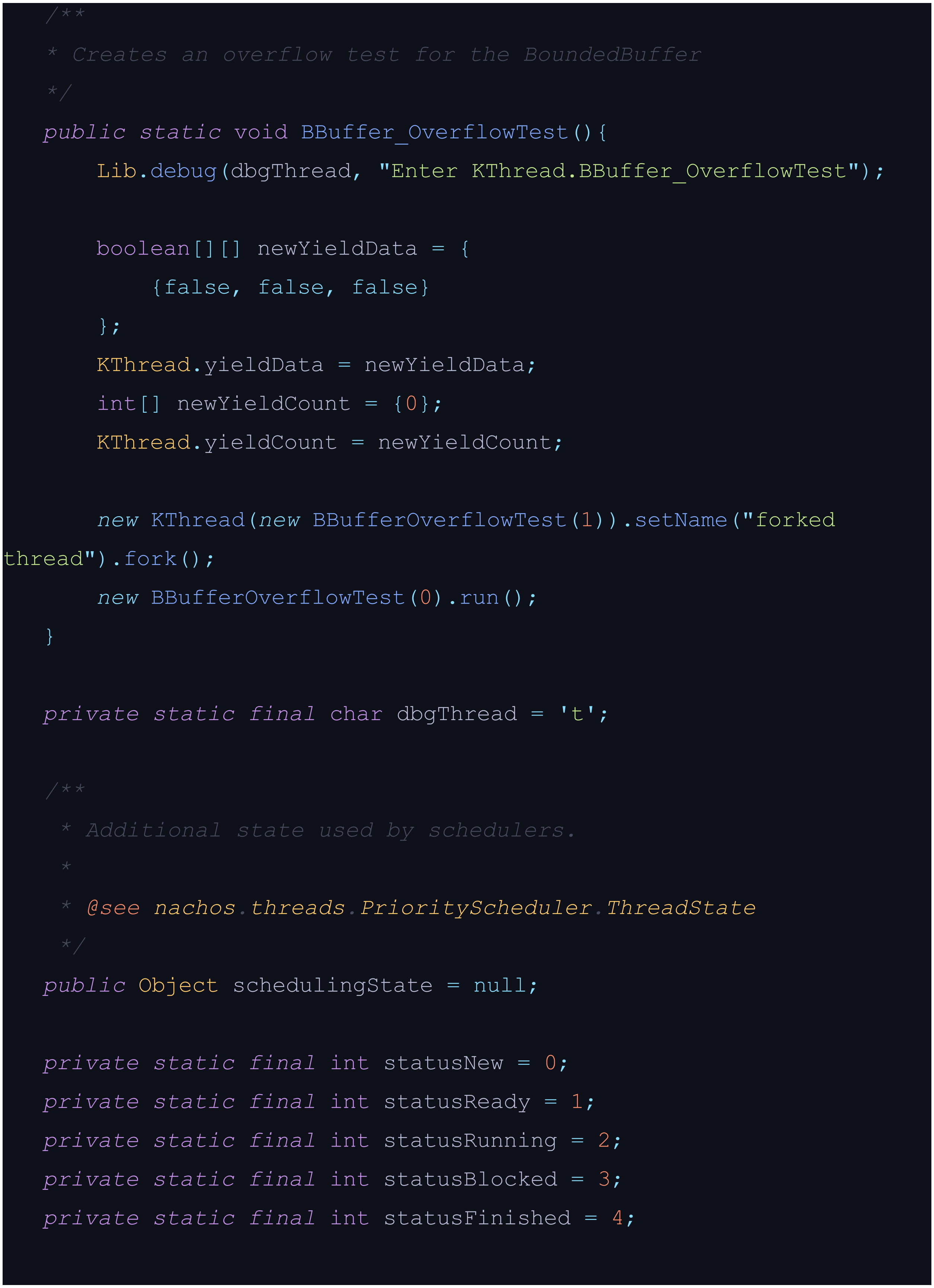
```
            this.which = which;
        }

        public void run() {
            if (this.which == 0) {
                myBB.write('a');
                myBB.write('b');
                myBB.write('c');
                System.out.println("Final buffer:");
                myBB.print();

            } else {
                char c = myBB.read();
                System.out.println("Final return char: " + c);
            }






        }

        private int which;
    }

/**
 * Tests whether this module is working.
 */
public static void selfTest() {
Lib.debug(dbgThread, "Enter KThread.selfTest");

new KThread(new PingTest(1)).setName("forked thread").fork();
```



```
    new PingTest(0).run();
    }

    /**
    * Tests the shared DLList by having two threads running
countdown.
    * One thread will insert even-numbered data from "A12" to "A2".
    * The other thread will insert odd-numbered data from "B11" to
"B1". * Don't forget to initialize the oughtToYield array before
forking. *
    */
    public static void DLL_selfTest(){
        Lib.debug(dbgThread, "Enter KThread.DLL_selfTest");

        new KThread(new DLListTest(1)).setName("forked
thread").fork();
        new DLListTest(0).run();

    }

    /**
    * Creates fatal error interleaving using DLList
    */
    public static void DLL_fatalError(){
        Lib.debug(dbgThread, "Enter KThread.DLL_fatalError");

        boolean[][] newYieldData = {
            {true,false},
            {true,false},
            {false}
        };
        KThread.yieldData = newYieldData;
        int[] newYieldCount = {0,0,0};
```



```
        KThread.yieldCount = newYieldCount;

        new KThread(new DLLFatalErrorTest(1)).setName("forked
thread").fork();
        new DLLFatalErrorTest(0).run();

    }

    /**
    * Creates non fatal error interleaving using DLList
    */
    public static void DLL_nonFatalError(){
        Lib.debug(dbgThread, "Enter KThread.DLL_nonFatalError");

        boolean[][] newYieldData = {
            {false},
            {false},
            {true}
        };
        KThread.yieldData = newYieldData;
        int[] newYieldCount = {0,0,0};
        KThread.yieldCount = newYieldCount;

        new KThread(new DLLNonFatalErrorTest(1)).setName("forked
thread").fork();
        new DLLNonFatalErrorTest(0).run();
    }

    /**
    * Creates a mutual exclusion test for the BoundedBuffer
    */
    public static void BBuffer_MutualExclusionTest(){
        Lib.debug(dbgThread, "Enter
KThread.BBuffer_MutualExclusionTest");

        boolean[][] newYieldData = {
            {true, false}
        };
        KThread.yieldData = newYieldData;
        int[] newYieldCount = {0};
        KThread.yieldCount = newYieldCount;

        new KThread(new BBufferMutualExclusionTest(1)).setName("forked
thread").fork();
        new BBufferMutualExclusionTest(0).run();
    }

    /**
    * Creates an underflow test for the BoundedBuffer
    */
    public static void BBuffer_UnderflowTest(){
        Lib.debug(dbgThread, "Enter KThread.BBuffer_UnderflowTest");

        boolean[][] newYieldData = {
            {false}
        };
        KThread.yieldData = newYieldData;
        int[] newYieldCount = {0};
        KThread.yieldCount = newYieldCount;

        new KThread(new BBufferUnderflowTest(1)).setName("forked
thread").fork();
        new BBufferUnderflowTest(0).run();
    }
```

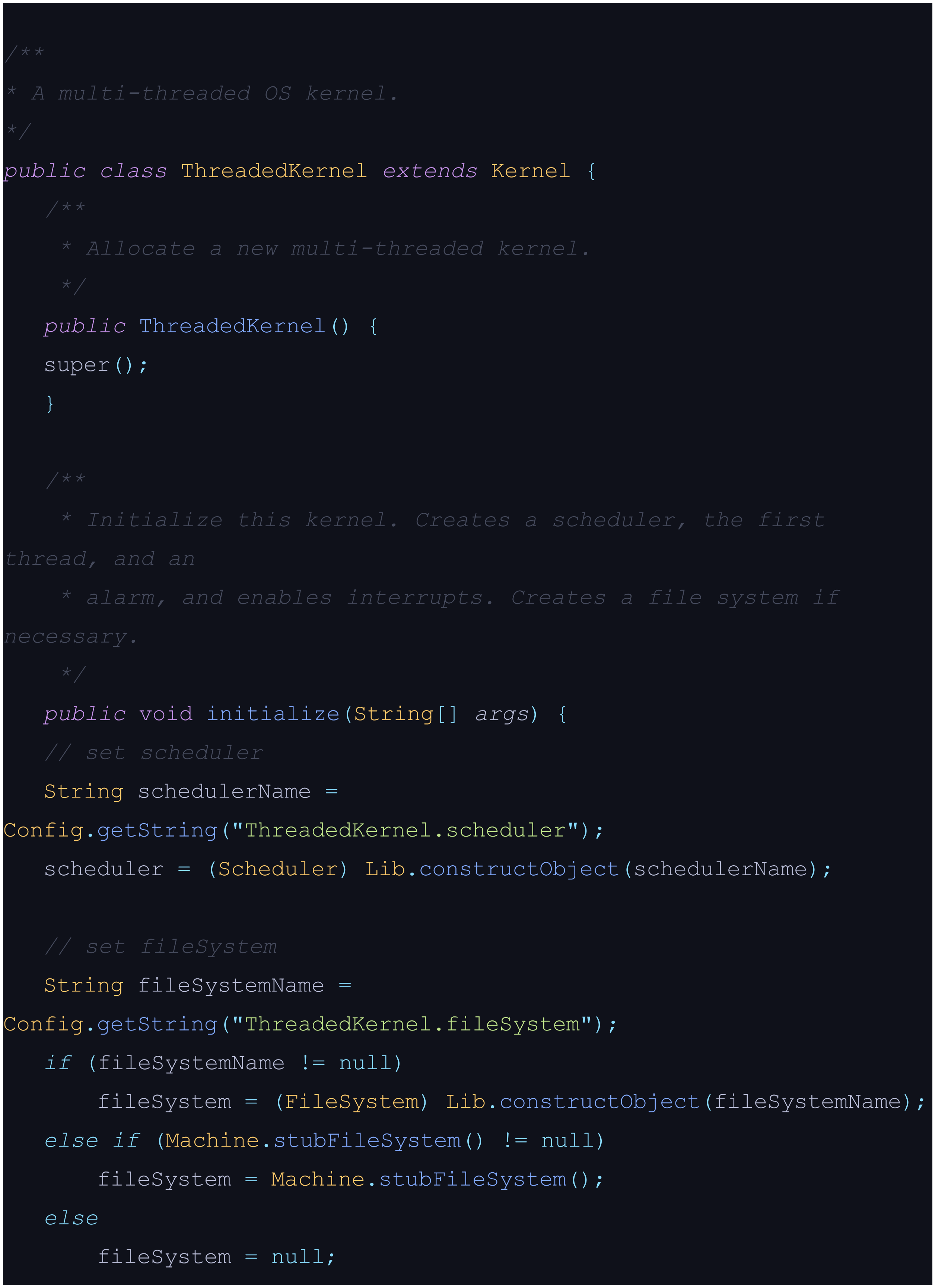


```
   /**
   * Creates an overflow test for the BoundedBuffer
   */
   public static void BBuffer_OverflowTest(){
       Lib.debug(dbgThread, "Enter KThread.BBuffer_OverflowTest");

       boolean[][] newYieldData = {
           {false, false, false}
       };
       KThread.yieldData = newYieldData;
       int[] newYieldCount = {0};
       KThread.yieldCount = newYieldCount;

       new KThread(new BBufferOverflowTest(1)).setName("forked
thread").fork();
       new BBufferOverflowTest(0).run();
   }

   private static final char dbgThread = 't';

   /**
    * Additional state used by schedulers.
    *
    * @see nachos.threads.PriorityScheduler.ThreadState
    */
   public Object schedulingState = null;

   private static final int statusNew = 0;
   private static final int statusReady = 1;
   private static final int statusRunning = 2;
   private static final int statusBlocked = 3;
   private static final int statusFinished = 4;
```

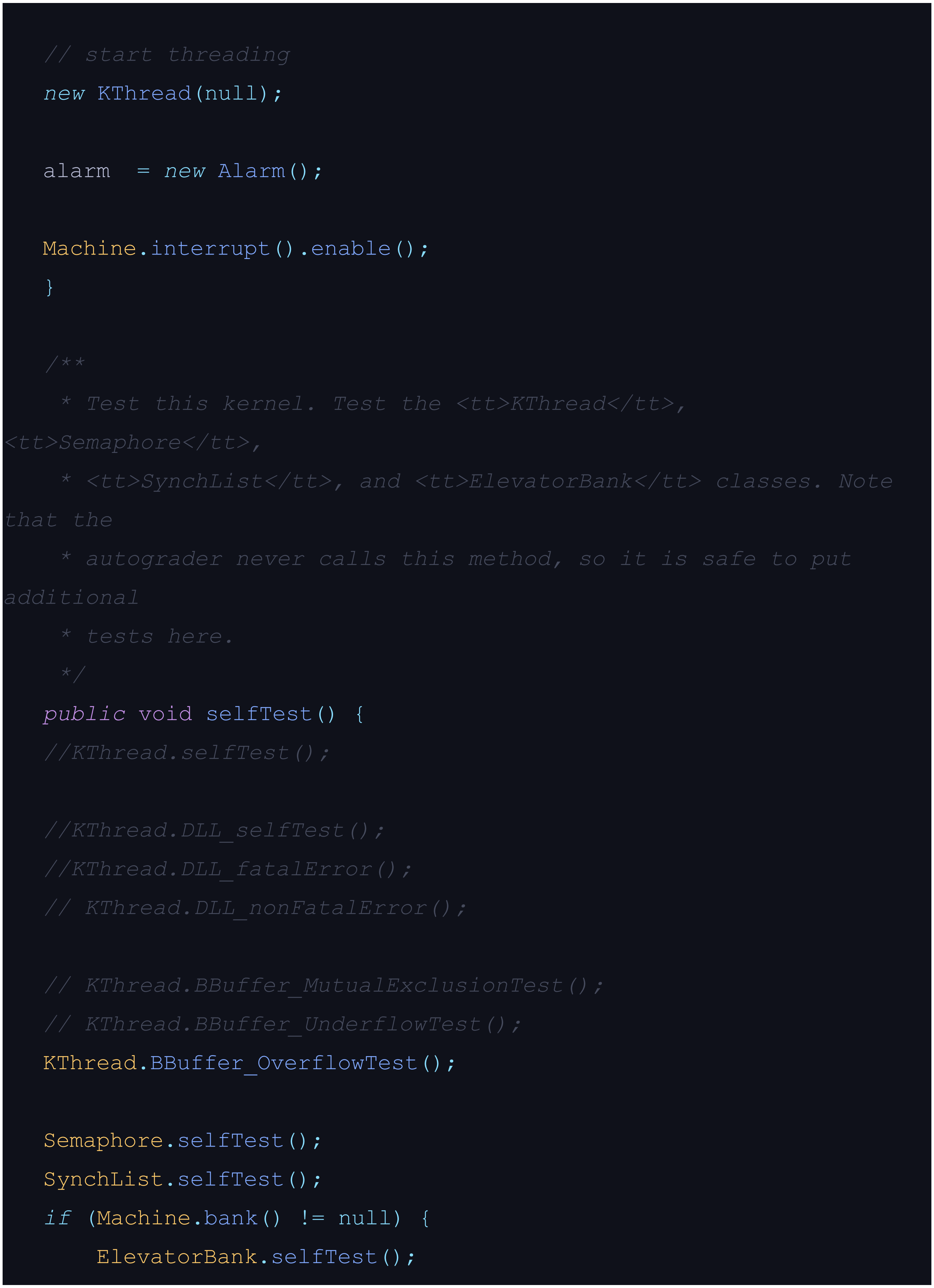
```
    /**
     * The status of this thread. A thread can either be new (not
yet forked),
     * ready (on the ready queue but not running), running, or
blocked (not
     * on the ready queue and not running).
     */
    private int status = statusNew;
    private String name = "(unnamed thread)";
    private Runnable target;
    private TCB tcb;

    /**
     * Unique identifer for this thread. Used to deterministically
compare
     * threads.
     */
    private int id = numCreated++;
    /** Number of times the KThread constructor was called. */
    private static int numCreated = 0;

    private static ThreadQueue readyQueue = null;
    private static KThread currentThread = null;
    private static KThread toBeDestroyed = null;
    private static KThread idleThread = null;
}
```

ThreadedKernel:

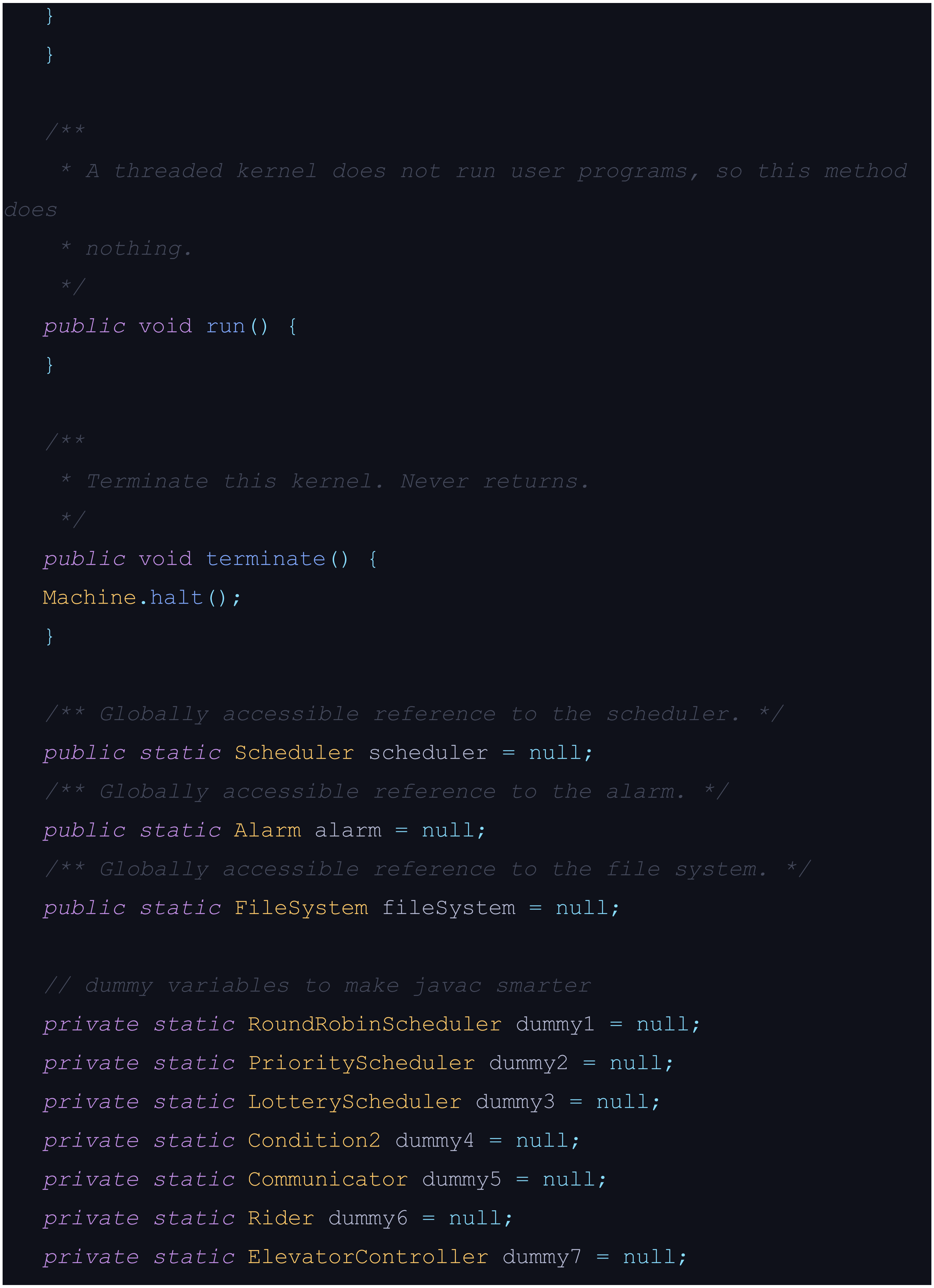
```
package nachos.threads;

import nachos.machine.*;
```



```
/**
* A multi-threaded OS kernel.
*/
public class ThreadedKernel extends Kernel {
   /**
    * Allocate a new multi-threaded kernel.
    */
   public ThreadedKernel() {
   super();
   }

   /**
    * Initialize this kernel. Creates a scheduler, the first
thread, and an
    * alarm, and enables interrupts. Creates a file system if
necessary.
    */
   public void initialize(String[] args) {
   // set scheduler
   String schedulerName =
Config.getString("ThreadedKernel.scheduler");
   scheduler = (Scheduler) Lib.constructObject(schedulerName);

   // set fileSystem
   String fileSystemName =
Config.getString("ThreadedKernel.fileSystem");
   if (fileSystemName != null)
       fileSystem = (FileSystem) Lib.constructObject(fileSystemName);
   else if (Machine.stubFileSystem() != null)
       fileSystem = Machine.stubFileSystem();
   else
       fileSystem = null;
```



```
   // start threading
   new KThread(null);

   alarm  = new Alarm();

   Machine.interrupt().enable();
   }

   /**
    * Test this kernel. Test the <tt>KThread</tt>,
<tt>Semaphore</tt>,
    * <tt>SynchList</tt>, and <tt>ElevatorBank</tt> classes. Note
that the
    * autograder never calls this method, so it is safe to put
additional
    * tests here.
    */
   public void selfTest() {
   //KThread.selfTest();

   //KThread.DLL_selfTest();
   //KThread.DLL_fatalError();
   // KThread.DLL_nonFatalError();

   // KThread.BBuffer_MutualExclusionTest();
   // KThread.BBuffer_UnderflowTest();
   KThread.BBuffer_OverflowTest();

   Semaphore.selfTest();
   SynchList.selfTest();
   if (Machine.bank() != null) {
       ElevatorBank.selfTest();
```



```
   }
   }

   /**
    * A threaded kernel does not run user programs, so this method
does
    * nothing.
    */
   public void run() {
   }

   /**
    * Terminate this kernel. Never returns.
    */
   public void terminate() {
   Machine.halt();
   }

   /** Globally accessible reference to the scheduler. */
   public static Scheduler scheduler = null;
   /** Globally accessible reference to the alarm. */
   public static Alarm alarm = null;
   /** Globally accessible reference to the file system. */
   public static FileSystem fileSystem = null;

   // dummy variables to make javac smarter
   private static RoundRobinScheduler dummy1 = null;
   private static PriorityScheduler dummy2 = null;
   private static LotteryScheduler dummy3 = null;
   private static Condition2 dummy4 = null;
   private static Communicator dummy5 = null;
   private static Rider dummy6 = null;
   private static ElevatorController dummy7 = null;
```

```
}
```