Guidelines

This exam has 9 pages (including this one); make sure you have them all. Put your name on each page before starting the exam. Write your answers directly on the exam sheets. Bring your exam to the front when you are finished. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn’t need to do this at all, so be careful when making assumptions.

You may avail yourself of the punt rule. If you write down punt for any part of a question, you will earn 1/5 of the points for that question (rounded to nearest integer). Make it clear what you want to punt on.

Use good test-taking strategy: read through the whole exam first, and first answer the questions that are easiest for you and are worth the most points.

Ignore InterruptedException: In all questions, you are free to ignore InterruptedException. You don’t need to worry about it, catch it, or deal with it.
1. Short answer (15 points total)

(a) If you ask your web browser to display the web page at \texttt{http://www.cs.umd.edu/users/pugh/index.html}, what might be the first line of the HTTP request sent by the web browser to the web server?

(b) Briefly describe Cross site scripting. What can it do if you have Javascript and other active content (e.g., Flash and Java) turned off in your browser? What can it do if they are turned on?

(c) Discuss the interaction between cookies and cross site request forgery. If a web site doesn’t use cookies, how does that change the impact of cross site request forgery?
2. Thread jukebox (15 points)

Say we had a design where any of several threads could request that a particular song should be played, but the threads shouldn’t be blocked while until the requested song is played. We can also only play one song at a time; each song is played to completion before the next song starts. Sketch a design for implementing this. You can provide code below if you find that easier than describing a design. But a detailed design is sufficient (make sure you provide all the details needed to allow the code to be written).

Don’t worry about the exact name of any classes or methods. Tell me what the classes and methods do, and you can name them whatever you want.

Note: there is more than one good design (and probably several bad designs). Any good design is fine.

// Optional code template; a pure English description is perfectly fine

public class ThreadJukebox {
   /** Request a song be played; method doesn’t block until the song
    * is played. This method needs to be thread safe (allow
    * simultaneous calls by different threads)
    */
   public void playSong(Song song) {

3. HTML forms (20 points)

Looking at the HTML for the page [http://store.apple.com/ipad](http://store.apple.com/ipad), you see the following as part of the HTML.

```html
<li>
<form action="/buyNow" method="GET">
   iPad, 16GB, Wi-Fi, $499
   <input type="hidden" name="model" value="16gb-wifi" />
   <input type="hidden" name="price" value="499" />
   <input type="submit" value="Buy with 1-click" />
</form>
</li>

<li>
<form action="/buyNow" method="GET">
   iPad, 16GB, Wi-Fi + 3G, $629
   <input type="hidden" name="model" value="16gb-wifi-3g" />
   <input type="hidden" name="price" value="629" />
   <input type="submit" value="Buy with 1-click" />
</form>
</li>
```

- What URL would the browser load if you clicked the “Buy with 1-click” button for the iPad without 3G.

- What serious problem(s) do you see? Explain why they are problems.
4. Decorator (15 points)

The *Iterator* interface is given below:

```java
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Write/implement the following two *Iterator* decorators:

(a) **Non-removing iterator** This decorator is used to create a decorator that prevents *remove* from being invoked on the decorated *Iterator* by throwing an UnsupportedOperationException if a call to *remove* is made.
(b) **enhanced iterator** This decorator is used to support an additional capability: a method `E curr()` that returns the value most recently returned by `next()`. Calling `curr()` is idempotent: calling it multiple times in a row just returns the same value. Calling `curr()` before `next()` has been called results in a `IllegalStateException` being thrown.
5. Testing a Semaphore (15 points)

Consider a Semaphore initialized to hold two permits. We want to test the following situation. One thread acquires two permits, which is granted immediately. It then tries to acquire another permit, and this request blocks. Then, another thread releases a permit, and this allows the first thread to proceed.

Write a multithreaded test case, using the metronome timer and testing framework discussed in class and used in project 5, that tests this scenario.

As a reminder of how the metronome timer and testing framework works, here is the public test from project 5 that checks that two locks provide mutual exclusion.

```java
static class TwoLocksProvideMutualExclusion extends MultithreadedTestCase {
    final Lock lock0, lock1;

    public TwoLocksProvideMutualExclusion(Lock lock0, Lock lock1) {
        this.lock0 = lock0;
        this.lock1 = lock1;
    }

    public void thread0() {
        lock0.lock();
        assertEquals(0, getTick());
        waitForTick(2);
        lock0.unlock();
        assertEquals(2, getTick());
    }

    public void thread1() {
        waitForTick(1);
        lock1.lock(); // should block here
        assertEquals(2, getTick());
        waitForTick(3);
        lock1.unlock();
        assertEquals(3, getTick());
    }
}
```

For your case, you should only worry about testing the methods shown below. Note: for the unlike the standard Semaphore class, the methods don’t throw InterruptedException, and you don’t have to worry about interrupts.

```java
public class Semaphore {
    ...
    // the only methods you should invoke in your test cases.
    public void acquire(int permits) { ... }
    public void release(int permits) { ... }
}
```
static class CheckSemaphoreWithTwoPermits extends MultithreadedTestCase {
    final Semaphore semaphore = Semaphore(2);

    public void thread0() {

    }

    public void thread1() {

    }
}
6. A buggy semaphore (20 points)

We have a semaphore implementation, part of which is provided below. It seems to work in all of the test cases with one or two threads. But in fails in one test case with three threads. If you have a semaphore with no remaining permits, and two threads each try to acquire a permit, both requests are blocked, as they should be. But then, if another thread releases one permit, both of the blocked acquire requests become unblocked, but only one of them should. Based on this bug report, provide your best guess as to what the problem is, and what the code for acquire looks like.

(a) Description of likely problem

(b) Example of code that would have this problem (you are free to ignore the fact that wait is declared to throw InterruptedException).

```java
public class Semaphore {
    int permits;
    public Semaphore(int permits) { this.permits = permits; }
    public synchronized void release(int p) {
        this.notifyAll();
        this.permits += p;
    }
    public synchronized void acquire(int p) {
    }
}
```

(c) Correct implementation of acquire (consistent with the design above):

```java
public synchronized void acquire(int p) {
}
```