Guidelines

This exam has 7 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, using the back of the page as necessary. 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 clean 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.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100</strong></td>
<td></td>
</tr>
</tbody>
</table>
1. Short answer

(a) If you ask your web browser to display the web page at http://www.cs.umd.edu/users/pugh/index.html, what will be the first line of the request sent by the web browser? Feel free to mark any places where it might be browser dependent.

(b) For each of the following, indicate which of the following design patterns is most likely to be relevant or used:

- Observer
- Decorator
- Adapter
- Visitor

  i. A graphical user interface
  ii. Reading a log file
  iii. Looking for coding errors in a class file

(c) Briefly, what does a continuous build system do and when? (One sentence should suffice).

(d) Describe two situations in which you would use branching in a version control system.
2. Bounded wildcards in generics

Assume you have a generic class `Foo<E>`. The `Foo` class defines a method `process` that takes a collection as an argument.

Assume that we also have

```java
class A { ... }
class B extends A { ... }
```

```java
Foo<A> f;
Collection c1;
Collection<Object> c2;
Collection<A> c3;
Collection<B> c4;
Collection<String> c5;
```

Below are some possible definitions for the argument of `process` in `Foo<E>`. For each one, report which `cn` variables could be passed to `f.process` with that definition.

(a) `void process(Collection c)`

(b) `void process(Collection<?> c)`

(c) `void process(Collection<? extends E> c)`

(d) `void process(Collection<? super E> c)`

(e) `void process(Collection<E> c)`

(f) `void process(Collection<Object> c)`
3. Decorator.

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 return the value most recently returned by `next`. Calling `curr()` is idempotent: calling it multiple times has no impact. Calling `curr()` before `next()` has been called results in a `IllegalStateException` being thrown.
4. Concurrency
This question requires you to implement a variant of the BoundedBuffer example we discussed in class. The variant is that rather than putting or taking a single element at a time, a sequence of objects are added or remove at once, as an atomic operation. In other words, if one thread tries to add ("a", "b") and another thread tries to add ("c", "d"), then is result must not be that the queue contains ("a", "c", "b", "d").

We are asking you to provide two implementations: the first one is a correct implementation. The second one should be correct except that the operations might not be performed atomically (e.g., if asked to put 5 elements into the queue, and the queue can only hold 3 more elements, 3 elements will be added immediately, and the put operation will wait/block until more room is available and the remaining 2 elements can be added to the queue.

To help you get started, we’ve provided you with a base implementation that isn’t thread safe and that doesn’t handle the situation where an operation can’t be performed immediately. You don’t have to base your implementations on this implementation, but you are welcome to do so:

```java
public class BoundedQueue {
    final LinkedList lst = new LinkedList();
    final int capacity;

    /** Create a buffer that can hold up to capacity elements */
    public BoundedQueue(int capacity) {
        this.capacity = capacity;
    }

    /** Remove N elements and return them, in order. If the queue doesn’t
     * contain N elements, the operation must block until N elements are available.
     * The elements must be taken as an atomic operation. If the thread
     * is interrupted while waiting, an InterruptedException exception is thrown
     * and the interrupted call makes no changes to the BoundedQueue.
     */
    public List takeN(int count) throws InterruptedException {
        if (count > lst.size())
            throw new UnsupportedOperationException("You must handle this case");
        LinkedList result = new LinkedList();
        for(int i = 0; i < count; i++) result.addLast(lst.removeFirst());
        return result;
    }

    /** Add a list of elements to the queue. If the queue contain contain all of
     * the added elements, the operation blocks until they can all be added in
     * a single atomic update that doesn’t exceed the capacity of the queue.
     * If the thread is interrupted while waiting, an InterruptedException exception is thrown
     * and the interrupted call makes no changes to the BoundedQueue.
     */
    public void putAll(List addThese) throws InterruptedException {
        if (lst.size() + addThese.size() > capacity)
            throw new UnsupportedOperationException("You must handle this case");
        for(Object e : addThese) lst.addLast(e);
    }
}
```
(a) Correct implementation

(b) Non-atomic implementation (but otherwise thread safe and correct)
5. Visitor Pattern

Assume we have the following classes/interfaces that are used to represent abstract syntax trees (ASTs).

```java
public interface Node {
}

public class IntegerConstant implements Node {
    public final int value;
    public IntegerConstant(int value) { this.value = value; }
}

public class PlusBinaryOp extends Node {
    public final Node lhs, rhs;
    public PlusBinaryOp(Node lhs, Node rhs) {
        this.lhs = lhs;
        this.rhs = rhs;
    }
}
```

Extend this code by

- defining a Visitor interface
- Adding methods to the Node classes to work with visitors
- Define a Evaluate class that implements the Visitor interface and computes the value of a node.

You should define things so that it would be easy to define new visitors that do other operations, such as produce a String representation of a AST. You don’t have to worry about doing anything to abstract out order in which the visitor visits nodes of the abstract syntax tree; it is OK to hard code that logic into each visitor.