Quiz 4 (April 30, 2014)

Your name: _________________________________________________________

Your Athena username: _______________________________________________

Circle your recitation time: 10am 11am 12pm 1pm 2pm

You have 50 minutes to complete this quiz. It contains 8 pages (including this page) for a total of 100 points.

The quiz is closed-book and closed-notes, but you are allowed one two-sided page of notes. Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Before you begin, write your name on the top of every page.

Please write neatly. **No credit will be given if we cannot read what you write.**

For questions which require you to choose your answer(s) from a list, do so clearly and unambiguously by circling the number(s) or entire answer(s). Do not use check marks, underlines, or other annotations – they will not be graded.

*Good luck!*

<table>
<thead>
<tr>
<th>Points</th>
<th>Score</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple choice</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>map-filter-reduce</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>Sentence completion</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Recursive program recognition</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Concurrency arguments</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td><strong>100</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 1. (5 points, circle all that apply)
Which of the following must be true about Subclassing (B extends A)?
1. B and A must display encapsulation and be independently analyzable.
2. B must be a true subtype of A.
4. B must fulfill the same contract as A.
5. B must provide its own equals method.

Problem 2. (5 points, circle all that apply)
Java Swing package has the following properties
1. Swing methods on view objects should not be directly invoked by user's threads.
2. Swing package cannot be used in multi-threaded applications
3. While the main method is running, the GUI is running.
4. We can have multiple listeners on a publisher.
5. To solve the problem of UI unresponsiveness due to a blocked UI thread, we can call SwingUtilities.invokeLater().

Problem 3. (5 points, circle all that apply)
Using the Java keyword synchronized demonstrates which of the following ideas?
1. Thread safety
2. Message passing
3. Deadlock
4. Locking
5. Queues

Problem 4. (5 points, circle all that apply)
Programs that use concurrency are hard to debug because:
1. They can result in deadlock, which is impossible to detect
2. The client doesn't know how many threads are running
3. They are non-deterministic, which makes it difficult to reproduce a bug
4. The Swing event thread is always running
5. They use immutable datatypes
Problem 5. (5 points, circle all that apply)

If we are implementing message passing between two threads using a Queue, which of the following are acceptable ways to ensure thread-safety?

1. Using a thread-safe datatype, such as a BlockingQueue.
2. Using an immutable Queue.
3. Using synchronized(queue) in all threads that observe or modify the Queue.
4. Ensuring confinement of the Queue.
5. There is no way to ensure thread-safety when implementing message passing.

Problem 6. (5 points, circle all that apply)

Map/Filter/Reduce:

1. produces tree-like data structures.
2. Is used for operating over sequences of elements.
3. is an effective pattern for message-passing.
4. is an effective tool for separating concerns in user interface.
5. uses higher-order functions.
The python function below accepts a list of numbers and calculate the product of all the odd numbers.

```python
def productOfOdds(list):
    result = 1
    for i in list:
        if i % 2 == 1:
            result *= i
    return result
```

Suppose you want to rewrite the above Python code using the map, filter, and reduce Python functions

```python
def productOfOdds(list):
    return reduce(r_func, filter(f_func, map(m_func, list)))
```

**Problem 7.** Fill in the blanks for the function `m_func` used in map (6 points)

```python
def m_func(__________________):
    return __________________
```

**Problem 8.** Fill in the blanks for the function `f_func` used in filter (6 points)

```python
def f_func(__________________):
    return __________________
```

**Problem 9.** Fill in the blanks for the function `r_func` used in reduce (6 points)

```python
def r_func(__________________):
    return __________________
```
Problem 10. (32 points)

For each of the items in the first list, select the single best item from the second list to complete the sentence and write the letter corresponding to that sentence in the blank provided. Each sentence in the second list cannot be used more than once.

1. Abstract data types ...
2. Components in a view hierarchy ...
3. Consistent names for design patterns ...
4. Creator operations ...
5. Event listeners ...
6. Functional objects ...
7. Immutable types ...
8. Iterators ...

A. ... abstract away details of accessing elements in a sequence.
B. ... abstract away details of calling a function.
C. ... are never threadsafe.
D. ... are not supported in functional programming languages.
E. ... can be implemented using the composite pattern.
F. ... can be implemented using the factory method pattern.
G. ... cannot include mutator operations.
H. ... define a mapping from concrete values to abstract values.
I. ... help software developers communicate clearly.
J. ... must always be threadsafe.
K. ... must always be true subtypes.
L. ... represent a function by implementing an interface known to the caller.
M. ... send/receive events to/from user interface components.
N. ... subscribe/unsubscribe to/from event sources at runtime.
O. ... take in a function as input or produce a function as output.
Examine the following code, the definitions for the *ImList* interface and classes *Empty* and *Cons* are on page 8 of the test.

```java
public class NormRe {
    private static void normCal(final ImList<String> a, final ImList<String> b) {
        if (b.isEmpty())
            System.out.println(a);
        else {
            normCal(a, b.rest());
            normCal(a.cons(b.first()), b.rest());
        }
    }

    public static void main(String[] args) {
        final ImList<String> input = new Empty<String>().cons("M").cons("I").cons("T");
        normCal(new Empty<String>(), input);
    }
}
```

**Problem 11.** (5 points, circle the best description)

When this program gets executed, the results that gets printed is:
(note that the newline in the output is represented by a comma)

1. , MIT, MTI, IMT, ITM, TMI, TIM,
2. , M, I, MI, T, MT, IT, MIT,
3. , M, MI, MIT,
4. , M, I, T,
5. , MIT,

**Problem 12.** (5 points, circle the best description)

In *normCal*, the recursion

1. Will always terminate because lists *a* and *b* are of finite length
2. Will always terminate because each recursive call will get a list *b* of smaller length than the caller’s list
3. Will never terminate because some recursive call will get a list *a*, which is longer than the caller’s list
4. Will grow exponentially and never terminate as there are two recursive calls in each call to *normCal*
5. Will always terminate because the recursive call is never invoked
The code from the previous problem was modified to include concurrency as follows:

```java
public class ConRe {
    private static void conCal(final ImList<String> a, final ImList<String> b, final BlockingQueue<ImList<String>> res) {
        if (b.isEmpty())
            res.add(a);
        else {
            new Thread() {
                public void run() {
                    conCal(a, b.rest(), res);
                }
            }.start();
            conCal(a.cons(b.first()), b.rest(), res);
        }
    }

    public static void main(String[] args) {
        final BlockingQueue<ImList<String>> results = new LinkedBlockingQueue<ImList<String>>();
        final ImList<String> input = ...
        new Thread() {
            public void run() {
                try {
                    while (true) {
                        System.out.println(results.take().toString());
                    }
                } catch (InterruptedException e) {
                }
            }
        }.start();
        conCal(new Empty<String>(), input, results);
    }
}
```

**Problem 13.** (5 points, circle all that apply)
The above program, when executed,

1. Will calculate the answers concurrently and will print the results as they get calculated
2. Will print the answers in a different order in different invocations
3. Will print different answers in every invocation due to a race condition bug
4. Will print exactly the same answer in the exactly the same order every time as there are no concurrency bugs here
5. Will deadlock as the `results.take()` is executed before the code for `rest.add(a)` is run

Answer: 1, 2
Problem 14. (5 points, circle all that apply)
The thread safety argument of the above code will include:
1. Any thread safety violation will be caught by the exception handler in the code
2. While LinkedBlockingQueue object is mutated, it is thread safe as it is from a threadsafe collection
3. LinkedBlockingQueue object is thread safe as it is immutable because it is final
4. Arguments a and b within conCal are thread safe as ImList<E> is immutable
5. Arguments a and b within conCal are thread safe as they are confined to a single thread

```java
public interface ImList<E> {
    public ImList<E> cons (E e);
    public E first();
    public ImList<E> rest();
    public boolean isEmpty();
    public String toString();
}

public class Empty<E> implements ImList<E> {
    public Empty() { }
    public ImList<E> cons(E e) { return new Cons<E>(e, this); }
    public E first() { throw new UnsupportedOperationException(); }
    public ImList<E> rest() {
        throw new UnsupportedOperationException();
    }
    public boolean isEmpty() { return true; }
    public String toString() { return ""; }
}

public class Cons<E> implements ImList<E> {
    private E e;
    private ImList<E> rest;
    public Cons(E e, ImList<E> rest) {
        this.e = e;
        this.rest = rest;
    }
    public ImList<E> cons(E e) { return new Cons<E>(e, this); }
    public E first() { return e; }
    public ImList<E> rest() { return rest; }
    public boolean isEmpty() { return false; }
    public String toString() { return first().toString() + rest().toString(); }
}
```