You have 50 minutes to complete this quiz.
The quiz contains 9 pages (including this one).

Before you begin, write your Athena username on the top left of every page.

Answer the questions in the spaces provided on the question sheets.

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

For questions which require you to circle your answer(s), do so clearly and unambiguously. Circle A, the letter or B, the entire answer. Do not use ✓ check marks, underlines, or other annotations – they will not be graded.

Name: 

Athena username: 

<table>
<thead>
<tr>
<th>Page</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
**Short Multiple Choice**

Recall our definitions of *map, filter, and reduce*, which have the following signatures:

- $\text{map} : (T \rightarrow U) \times \text{Seq}(T) \rightarrow \text{Seq}(U)$
- $\text{filter} : (T \rightarrow \text{boolean}) \times \text{Seq}(T) \rightarrow \text{Seq}(T)$
- $\text{reduce} : (U \times T \rightarrow U) \times \text{Seq}(T) \times U \rightarrow U$

Circle True or False:

1. *map* is a higher-order function because, when we use it, we do not need to write explicit control flow.
   A. True   B. False (2pts)

2. We can make use of *map* and *reduce* in situations where types $T$ and $U$ are actually the same type.
   A. True   B. False (2pts)

3. We cannot implement map/filter/reduce in Java because there are no first-class functions.
   A. True   B. False (2pts)

4. The following Python definition for *filter* is correct:
   $\text{def filter}(\text{pred}, \text{seq})$:
   $$\text{return map}(\lambda e: e \text{ if pred(e) else None, seq})$$
   A. True   B. False (2pts)

For the following questions, circle the best name for each pattern.

5. Traversing a container or collection and accessing each element.
   A. Encapsulation   B. Subclassing   C. Iteration   D. Exceptions   E. Generics
   F. Factory   G. Visitor   H. Publish-Subscribe   I. Model-View-Controller   J. Interpreter (3pts)

6. Parametrizing a class in terms of other classes.
   A. Encapsulation   B. Subclassing   C. Iteration   D. Exceptions   E. Generics
   F. Factory   G. Visitor   H. Publish-Subscribe   I. Model-View-Controller   J. Interpreter (3pts)

7. Building classes that build instances of other classes, to make it easier to build complex instances.
   A. Encapsulation   B. Subclassing   C. Iteration   D. Exceptions   E. Generics
   F. Factory   G. Visitor   H. Publish-Subscribe   I. Model-View-Controller   J. Interpreter (3pts)

8. Creating a new class with more functionality than the original, while ensuring it can be used everywhere the original could.
   A. Encapsulation   B. Subclassing   C. Iteration   D. Exceptions   E. Generics
   F. Factory   G. Visitor   H. Publish-Subscribe   I. Model-View-Controller   J. Interpreter (3pts)

9. Separating how data are shown to the user from how they are processed and stored.
   A. Encapsulation   B. Subclassing   C. Iteration   D. Exceptions   E. Generics
   F. Factory   G. Visitor   H. Publish-Subscribe   I. Model-View-Controller   J. Interpreter (3pts)
Calculations

We’d like to implement an interface Calc that represents double-precision mathematical calculations, and an interface Inverse that represents invertible calculations. Given the example usage below, complete each interface definition by:

1. Picking the correct specification; and
2. Writing the method signature.

Example usage:

```java
public class Root implements Calc, Inverse<Exp> {
    private final double degree;
    public Root(double degree) { this.degree = degree; }
    public double calc(double n) { return Math.pow(n, 1/degree); }
    public Exp inverse() { return new Exp(degree); }
}

public class Exp implements Calc, Inverse<Root> {
    private final double exponent;
    public Exp(double exponent) { this.exponent = exponent; }
    public double calc(double n) { return Math.pow(n, exponent); }
    public Root inverse() { return new Root(exponent); }
}
```

Complete the definitions by circling one specification and writing the method signature:

```java
public interface Calc {

1.   A. /** Given input "param", return a new Calc initialized with param. */ (4pts)
    B. /** Given input "num", return the value of this applied to num. */
    C. /** Given input "calc", return the value of calc applied to this. */
    D. /** Given inputs "calc" and "num", return the value of calc applied to num. */

2.   ______________________________; (7pts)
}

public interface Inverse<T extends Calc> extends Calc {

3.   A. /** Given input "T", return inverse T^(-1). */ (4pts)
    B. /** Given input type T, return an instance of T representing the inverse of this. */
    C. /** Return an instance of T representing the inverse of this. */
    D. /** Return a type T representing the inverse of this type. */

4.   ______________________________; (7pts)
}
```
SocialKarma & Concurrency

Jim Bitdiddle dreams of being the next Zuck, so the ambitious high-schooler writes the following web app to manage the social currency of his peers:

```java
public class Person {
    private int socialKarma;
    private List<Person> buddies;

    public Person() {
        socialKarma = 1000;
        buddies = new ArrayList<Person>();
    }

    public void newBuddies(Person friend) {
        buddies.add(friend);
    }

    public void neverNeverEver(Person frenemy) {
        buddies.remove(frenemy);
    }

    public void pay(Person friend, int points) {
        assert (points > 0);
        if (socialKarma - points > 0) {
            socialKarma -= points;
            friend.socialKarma += points;
        }
    }
}
```
Jim’s friends Elisa and Tim are so impressed, they use the system to send the following payments to Jim:

- Elisa: 100, 500
- Tim: 50

1. If these were the only payments done and each person’s payments are done from a single thread (i.e. there is one thread per person), the resulting socialKarma of Jim could be (select all that apply): (5pts)
   A. 1000  B. 1650  C. 1600  D. 1550

2. Thinking that there may be a concurrency issue, Jim rewrites the `Person.pay()` method with synchronized helper methods: (5pts)
   ```java
   private synchronized int getK() { return socialKarma; }
   private synchronized void addK(int k) { socialKarma += k; }
   
   public void pay(Person friend, int points) {
       assert (points > 0);
       if (getK() - points > 0) {
           addK(-points);
           friend.addK(points);
       }
   }
   ```

   If the above payments were redone (starting from a new instantiation of the whole program), Jim’s socialKarma could be (select all that apply): 
   A. 1000  B. 1650  C. 1600  D. 1550

3. Jim’s older brother Ben, a freshman at MIT, is disgusted by the naïveté of his little brother and rewrites the code into the following synchronized method: (5pts)
   ```java
   public synchronized void pay(Person friend, int points) {
       assert (points > 0);
       if (socialKarma - points > 0) {
           socialKarma -= points;
           friend.socialKarma += points;
       }
   }
   ```

   Again, if the system was restarted and the above payments were done, the Jim’s socialKarma could be (select all that apply): 
   A. 1000  B. 1650  C. 1600  D. 1550
4. Jim’s friends decide that having an ultra-cool friend improves the socialKarma of his/her buddies. So they decide to keep a graph of buddies using a list of best buddies for each person. When someone pays, they decide to only keep half the socialKarma points to him/herself and allocate the rest evenly to their best buddies.

```java
public void pay(Person friend, int points) {
    assert (points > 0);
    synchronized (friend) {
        if (socialKarma - points > 0) {
            socialKarma -= points / 2;
            friend.socialKarma += points / 2;
            for (Person buddy : friend.buddies) {
                pay(buddy, points / (2 * friend.buddies.size()));
            }
        }
    }
}
```

Given an arbitrary graph of buddies and arbitrary payments between them, this program may (select all that apply):

A. Deadlock
B. Enter infinite recursion
C. Have races, resulting in incorrect socialKarma points
D. Trigger the assert even when original points paid is > 0
E. None of the above, the program always works correctly

5. Jim is concerned about the concurrent access to the buddies list in his implementation in the previous question (that is, with the recursive `pay` method in #4). Jim should fix the problem by (select one):

A. Doing nothing
B. Using a concurrent list, changing the allocation of buddies to
   `buddies = Collections.synchronizedList(new ArrayList<Person>());`
C. Using `synchronized` on the `newBuddies` and `neverNeverEver` methods
D. Putting all computation in the `newBuddies` and `neverNeverEver` methods inside a `synchronized (buddies) { ... }`
Queued Stock Trading

Leif Noad just started a new job working for a stock trading company. He decides to implement a new system for trading stocks:

```java
public class Trade {
    public final int numShares;
    public final String stockName;
    public Trade(int numShares, String stockName) {
        this.numShares = numShares;
        this.stockName = stockName;
    }
}

public class TradeServer {
    private final Queue<Trade> tradesQueue;
    private final Set<TradeLogger> loggers;

    public TradeServer() {
        tradesQueue = new LinkedList<Trade> ();
        loggers = new HashSet<TradeLogger> ();
    }

    public Queue<Trade> getQueue() {
        return tradesQueue;
    }

    public void addLogger(TradeLogger logger) {
        loggers.add(logger);
    }

    public void addTrade(int numShares, String stockName) {
        Trade t = new Trade(numShares, stockName);
        tradesQueue.offer(t);
        for (TradeLogger logger : loggers) {
            logger.logTrade(this, t);
        }
    }
}

public class TradeWorker implements Runnable {
    private final Queue<Trade> tradesQueue;

    public TradeWorker(Queue<Trade> tradesQueue) {
        this.tradesQueue = tradesQueue;
    }

    public void run() {
        while (true) {
            Trade t = tradesQueue.poll();
            TradeProcessor.handleTrade(t.numShares, t.stockName);
        }
    }
}

public class TradeProcessor {
    public static void handleTrade(int numShares, String stockName) {
        /* ... process the trade ... takes a while ... */
    }

    public static void handleException(Exception e) {
        /* ... handle the exception ... */
    }
}

public interface TradeLogger {
    public void logTrade(TradeServer s, Trade t);
}
```
1. What design pattern is Leif implementing here? (3pts)
   A. Visitor  B. Producer/Consumer  C. Factories  D. Interpreter  E. None of these

2. Anna Graham notices the implementation of the server is broken. (8pts)
   Supposing several TradeWorker threads are running with a single TradeServer, what possible behaviors can result from the bug(s) in TradeServer? (Circle all that apply.)
   A. Some trades do not get processed in order
   B. A trade can get processed multiple times
   C. The program crashes with an exception
   D. Some trades get processed in the correct order

3. Anna decides to fix the problem(s) with the server by using a BlockingQueue throughout the server implementation, including the workers. She makes the appropriate changes to the TradeServer constructor and addTrade() method, and proceeds to work on TradeWorker.run(). (8pts)
   ```java
   public void run() {
      while (true) {
         try {
            ???
         } catch (Exception e) {
            TradeProcessor.handleException(e);
            throw new Error(e);
         }
      }
   }
   ```
   Circle all the code segments below that complete TradeWorker.run() correctly, including those with unnecessary synchronization or waiting.
   REMINDER:
   poll() returns null if the queue is empty — take() blocks until an item is available
   offer(..) adds to the queue if possible — peek() does not remove the peeked item
   A. Trade t = tradesQueue.poll();
      TradeProcessor.handleTrade(t.numShares, t.stockName);
   B. Trade t;
      synchronized (tradesQueue) {
         t = tradesQueue.take();
      }
      TradeProcessor.handleTrade(t.numShares, t.stockName);
   C. Trade t = tradesQueue.take();
      TradeProcessor.handleTrade(t.numShares, t.stockName);
   D. if (tradesQueue.peek() != null) {
      Trade t = tradesQueue.poll();
      TradeProcessor.handleTrade(t.numShares, t.stockName);
Leif designed his server with the flexibility to support arbitrary TradeLoggers.

4. **public class SuccessLogger implements** TradeLogger {  
    **public void logTrade**(TradeServer s, Trade t) {  
        System.out.println("OK, finished processing " + t + " with no errors");  
    }  
    
    }  

The output from a SuccessLogger (circle one):
  A. Is always accurate.
  B. May incorrectly print ‘OK’ for some trades that do *not* finish processing.
  C. May incorrectly never print ‘OK’ for some trades that *do* finish processing.
  D. Is never accurate.

5. **public class DaringLogger implements** TradeLogger {  
    **public void logTrade**(TradeServer s, Trade t) {  
        s.addTrade(1, "AAPL");  
    }  
    
    }  

Assuming we have at least one working TradeWorker, if we add *two* DaringLoggers to the TradeServer and run

    addTrade(1, "MSFT")

to trade one share of MSFT, how many shares of AAPL will we trade? (circle one)
  A. 1
  B. 2
  C. 4
  D. The value of tradesQueue.remainingCapacity()
  E. Unbounded

Did you write your Athena username on every page of the quiz?  

END