Solutions to Quiz 2 (November 19)

Problem 1 (Map/Filter/Reduce) (20 points).
Recall these three key higher-order functions for manipulating immutable lists in functional programming:

- map\(f\): apply a function \(f\) to all list elements to produce a new list.
- filter\(f\): keep only those list elements for which the function \(f\) returns true.
- reduce\(f\): combine the list elements into a single value using the two-argument function \(f\) (ignoring, for the purposes of this problem, some of the subtleties we discussed in class, with initial values, etc.).

In this problem, we expect you to write the function \(f\) using a lambda expression. For example, \(\lambda x, y. x + y\) represents a function that takes two inputs \(x\) and \(y\), and returns the value of the expression \(x + y\).

In this problem, you should write a pipeline of map/filter/reduce calls, using an arrow \(\rightarrow\) to separate pipeline stages, where the stages are run successively on the input list in left-to-right order. For instance, the following pipeline computes the sum of all positive numbers in a list:

\[
\text{filter}\left(\lambda n. n > 0\right) \rightarrow \text{reduce}(+) 
\]

Each problem below starts with a list of weather records, where a record \(r\) contains two fields: \(r.\text{Place}\), a string identifying the location where a reading was taken; and \(r.\text{Temp}\), an integer recording the temperature in degrees Fahrenheit.

Write a map/filter/reduce pipeline for each of the following computations on lists of weather records, using the notation introduced above.

(a) From a list of weather records, compute a list of all temperatures less than 0. The type of your answer should be a list of integers.

Solution.

\[
\text{filter}\left(\lambda x. x.\text{Temperature} < 0\right) \rightarrow \text{map}\left(\lambda x. x.\text{Temperature}\right) 
\]

(b) From a list of weather records, calculate the maximum recorded temperature in location “Boston”. The type of your answer should be an integer. (Hint: you may use \(\text{max}\), a function that takes in two integers and returns their maximum.)

Solution.

\[
\text{filter}\left(\lambda x. x.\text{Place} = \text{“Boston”}\right) \rightarrow \text{map}\left(\lambda x. x.\text{Temperature}\right) \rightarrow \text{reduce}(\text{max}) 
\]
Problem 2 (Deadlock) (20 points).
In the code below, three threads are trying to acquire 3 locks. This code can deadlock. Circle the method call where each thread would block in a three-way deadlock scenario. (There may be more than one possible three-way deadlock – just choose one.)

```
Thread 1      Thread 2      Thread 3
X.acquire()   X.acquire()   Y.acquire()
Y.acquire()   Z.acquire()   X.acquire()
Z.acquire()   Y.acquire()   Z.acquire()
Z.release()   X.release()   Y.release()
Y.release()   Z.release()   X.release()
X.release()   Y.release()   Z.release()
```

Solution. Only acquire() can block, not release(), so the only possible answers can only circle acquire() calls.

One solution is Thread1:X.acquire(), Thread2:Y.acquire(), Thread3:X.acquire().
Another solution is Thread1:Y.acquire(), Thread2:X.acquire(), Thread3:X.acquire().

Problem 3 (Locking Granularity) (20 points).
Listed below is the source code for a system that keeps track of airline ticket bookings. It contains the two classes BookingSystem and Ticket, and the CreditCard interface. You can assume that the public interface of any CreditCard object is thread-safe.

```
1 /** A booking system for airline tickets. */
2 public class BookingSystem {
3     private final Map<String, Set<Ticket>> ticketsByFlightCode =
4         new HashMap<String, Set<Ticket>>();
5     private int currentTicketNumber = 0;
6
7     public BookingSystem() {
8     }
9
10    /** Make a new ticket available for a flight. */
11    public void publishTicket(String flightCode, int price) {
12        Ticket ticket = new Ticket(flightCode, currentTicketNumber++, price);
13        Set<Ticket> tickets =
14            ticketsByFlightCode.get(ticket.getFlightCode());
15        if (tickets == null) {
16            tickets = new HashSet<Ticket>();
17            ticketsByFlightCode.put(ticket.getFlightCode(), tickets);
18        }
19        tickets.add(ticket);
20    }
21 }
```
/** Get tickets for a particular flight. */
* @param availableOnly If true, return only available tickets. */
public Set<Ticket> getTicketsForFlight(
    String flightCode, boolean availableOnly)
{
    Set<Ticket> tickets = ticketsByFlightCode.get(flightCode);
    if (tickets == null)
        return new HashSet<Ticket>();
    Set<Ticket> ret = new HashSet<Ticket>();
    for (Ticket ticket : tickets) {
        if (!availableOnly || !ticket.isBooked())
            ret.add(ticket);
    }
    return ret;
}

/**
 * If the ticket is still available, charge the credit card the price of the
 * ticket, and book the ticket if the credit card transaction was successful.
 * If the credit card is successfully charged, the ticket must successfully be
 * booked, and vice versa.
 * @return true if the ticket was successfully booked, false otherwise.
 */
public boolean bookTicket(Ticket ticket, CreditCard creditCard) {
    return ticket.book(creditCard);
}

/** Object representing an airline ticket in the booking system. */
public final class Ticket {
    private final String flightCode;
    private final int ticketNumber;
    private final int price;
    private boolean booked;

    Ticket(String flightCode, int ticketNumber, int price) {
        if (flightCode == null)
            throw new NullPointerException();
        if (price <= 0)
            throw new IllegalArgumentException();
        this.flightCode = flightCode;
        this.ticketNumber = ticketNumber;
        this.price = price;
        this.booked = false;
    }

    public boolean isBooked() {
        return booked;
    }

    public boolean book(CreditCard creditCard) {
        if (booked)
            return false;
        booked = creditCard.charge(getPrice());
    }
79    return booked;
80 }
81
82 public String getFlightCode() {
83    return flightCode;
84 }
85
86 public int getPrice() {
87    return price;
88 }
89 }
90
91 /** Object representing a customer’s credit card. You can assume this is thread-safe. */
92 public interface CreditCard {
93    /**
94     * Attempt to charge the credit card. Return true if successful, or false if
95     * the transaction was denied. This method may take up to a minute to run
96     * while the transaction is processed by the credit card company. There is no
97     * limit to the number of times this method can be called simultaneously,
98     * however.
99     *
100     * @param amount The amount to charge.
101     */
102     public boolean charge(int amount);
(a) In the code above, neither BookingSystem nor Ticket is thread-safe. **Assuming the Ticket class and CreditCard interface remain unchanged,** what is the minimal subset of methods in the BookingSystem class that must be declared synchronized in order to make BookingSystem’s public interface thread-safe? (Write “S” to the left of the fewest possible method names below, to indicate which methods should have the “synchronized” keyword specified in their declaration in this case):

```java
BookingSystem() // constructor
BookingSystem.publishTicket(...)
BookingSystem.getTicketsForFlight(...)
BookingSystem.bookTicket(...)
```

**Solution.** publishTicket(...) and getTicketsForFlight(...) must be synchronized because they access class members. bookTicket(...) must be synchronized because it calls a non-thread-safe method on Ticket. Constructors never need to be synchronized.

```java
S BookingSystem() // constructor
S BookingSystem.publishTicket(...)
S BookingSystem.getTicketsForFlight(...)
S BookingSystem.bookTicket(...)
```

(b) What is the minimal subset of methods in the Ticket class that must be declared synchronized in order to make Ticket’s public interface thread-safe? Please write “S” to the left of the fewest possible method names below, to indicate which methods should have the “synchronized” keyword specified in their declaration in this case:

```java
Ticket(...) // constructor
Ticket.isBooked()
Ticket.book(...)
Ticket.getFlightCode()
Ticket.getPrice()
```

**Solution.** Access to final fields need not be synchronized.

```java
S Ticket(...) // constructor
S Ticket.isBooked()
S Ticket.book(...)
Ticket.getFlightCode()
Ticket.getPrice()
```

(c) If we assume that the Ticket class has been modified to include the “synchronized” keyword in all methods required to make Ticket’s public interface thread-safe, what is now the minimal subset of methods in the BookingSystem class that must be declared synchronized in order to make BookingSystem’s public interface thread-safe? Please write “S” to the left of the fewest possible of the method names below, to indicate which methods should have the “synchronized” keyword specified in their declaration in this case:

```java
BookingSystem() // constructor
BookingSystem.publishTicket(...)
BookingSystem.getTicketsForFlight(...)
BookingSystem.bookTicket(...)
```
Solution. Since BookingSystem.bookTicket(...) does not actually access any of BookingSystem’s members, and since Ticket.book() can now be assumed to be thread-safe by itself, BookingSystem.bookTicket(...) does not need to be synchronized.

BookingSystem() // constructor
S BookingSystem.publishTicket(...)
S BookingSystem.getTicketsForFlight(...)
BookingSystem.bookTicket(...)

(d) In a situation where many customers are attempting to book different tickets at the same time, which thread-safety approach is likely to be both correct and yield the highest number of booked tickets per hour when clients are correctly calling BookingSystem.bookTicket(...)? Please circle one of the options below:

1. Make BookingSystem thread-safe according to the correct answer to part (a) above, and make Ticket thread-safe according to the correct answer to part (b) above.
2. Make Ticket thread-safe according to the correct answer to part (b) above, and make BookingSystem thread-safe according to the correct answer to part (c) above.
3. Leave Ticket not thread-safe, but make BookingSystem thread-safe according to the correct answer to part (a) above.
4. Make Ticket thread-safe according to the correct answer to part (b) above, but leave BookingSystem not thread-safe.
5. Leave both BookingSystem and Ticket not thread-safe.

Solution. Options (1), (2), and (3) work correctly; (4) and (5) obviously make no sense. (1) and (3) both lock up the entire booking system every time BookingSystem.bookTicket(...) is waiting for a credit card transaction to be processed. (2) avoids this by only locking the Ticket to be booked.

The correct answer is (2).

Problem 4 (Interleaving) (20 points).
Consider the code below.

```java
public class Main {
    public static void main(String[] args) {
        Thread hello = new Thread(new Hello());
        Thread kidding = new Thread(new Kidding());

        hello.start();
        System.out.print("A");
        kidding.start();
        System.out.print("B");
        kidding.join();
        System.out.print("C");
        hello.join();
    }
}
```
Assuming that there are no errors, no exceptions thrown, and that print() is threadsafe and atomic, circle all of the outputs that are possible at the end of execution of this code.

ABCHIJK
HIAJKB
ABJKHC
JKABHI
AHJKBC

Solution.  ABJHKIC
          JKABCHI
          AHJKBC
Recall that Collections.unmodifiableList() wraps any List object lst with an object that throws exceptions for mutator methods and delegates all other methods to the underlying lst object.

Louis Reasoner claims that this class is threadsafe because it’s immutable. He’s wrong. List all the threats to immutability that exist in this code.

**Solution.** The name field is public, so a client can change it directly.

Also, the prerequisites list can also be changed by the caller of the constructor, because the unmodifiable wrapper points to the original list object, and the caller could change that after the Course object is made.

However, callers of the getPrerequisites() method *can’t* mutate the list of prerequisites because of the unmodifiable wrapper. They can, however, mutate a Course on that list.

Finally, there is a technical reason in Java why this class wouldn’t be threadsafe even if these problems were fixed – the fields need to be declared final.