Solutions to Quiz 2 (December 3, 2019)

This quiz uses the same abstract data type as Quiz 1, information board entries. The description of the abstract values is reproduced on the rest of this page, unchanged from Quiz 1.

The problems in this quiz refer to the code for mutable MutInfoEntry and immutable ImInfoEntry, starting on page 11. This code is different than the code in Quiz 1. You may detach the code pages.

Train stations, airports, and other transit hubs often have displays that show upcoming departures or arrivals along with other information: a track or gate number, delays, cancellations, etc.

For this quiz, an information board is made of several information board entries. Each entry has limited space: 16 characters to display a destination and 12 characters for a status. Both are restricted to upper-case letters, digits, colons, and spaces. For example, a board with three entries:

| WASHINGTON DC | 11:05 AM |
| LONDON HEATHROW | 11:55 AM |
| HONG KONG | DELAYED |

In order to show more information, the board cycles each entry through a looping sequence of up to four statuses. For example, if WASHINGTON DC and LONDON HEATHROW have 2-status loops, and HONG KONG has a 3-status loop, then every few seconds the board will update:

WASHINGTON DC ON TIME
LONDON HEATHROW ON TIME
HONG KONG NEW DEPRTURE

WASHINGTON DC 11:05 AM
LONDON HEATHROW 11:55 AM
HONG KONG 1:40 PM

WASHINGTON DC ON TIME
LONDON HEATHROW ON TIME
HONG KONG DELAYED

WASHINGTON DC 11:05 AM
LONDON HEATHROW 11:55 AM
HONG KONG NEW DEPRTURE

WASHINGTON DC ON TIME
LONDON HEATHROW ON TIME
HONG KONG 1:40 PM

... and so on.

**Problem 1** (Thread Safety) (26 points).
Suppose a train station’s information board system uses MutInfoEntry objects. The system is multi-threaded:
• one thread, the display thread, calls nextStatus() on all the MutInfoEntry objects every few seconds in order to display a cycling sequence of statuses to people in the station.
• other threads, the update threads, can call setStatuses() on any MutInfoEntry object when updated information about a train is received.

(a) Describe a race condition between the display thread and an update thread by showing an interleaving of operations that leads to a bad outcome, and state what the bad outcome is.

<table>
<thead>
<tr>
<th>display thread</th>
<th>update thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>statuses.clear()</td>
<td>statuses.remove(0)</td>
</tr>
</tbody>
</table>

bad outcome:

Solution.

One bad interleaving is statuses.clear() on the update thread, immediately followed by statuses.remove(0) on the display thread, leading to throwing an ArrayIndexOutOfBoundsException, which violates the postcondition of nextStatus().

Another bad interleaving is statuses.remove(0) on the display thread, immediately followed by statuses.clear() on the update thread, which eventually puts an out-of-date status back on the end of the just-updated statuses list, violating the postcondition of setStatuses(). If setStatuses() was called with a 4-element list, then this would also lead to breaking the rep invariant, since statuses now contains 5 statuses.

(b) Which objects involved in the rep of MutInfoEntry are in danger of having their rep invariants broken by concurrency? Circle either DANGER or SAFE, and explain why in at most one sentence.

ArrayList rep invariant

DANGER SAFE because:

MutInfoEntry rep invariant

DANGER SAFE because:

String rep invariant

DANGER SAFE because:

Solution.

ArrayList is SAFE because the only reference to it is inside a synchronized wrapper. The answer "SAFE because ArrayList is responsible for establishing and maintaining its own rep invariant" is not correct, because this question asked specifically about concurrency, and ArrayList does not guarantee to be thread-safe. The answer "DANGER because ArrayList is not threadsafe" is also not a sufficient answer, because it ignores the presence of the synchronized wrapper. The answer "DANGER" with an explanation that talks about dangers to the MutInfoEntry rep invariant, is also not correct, because the constraints on a MutInfoEntry rep aren't relevant to ArrayList's rep invariant.

MutInfoEntry is in DANGER because interleaved calls to setStatuses() (between two update threads) or nextStatus() and setStatuses() (between the display thread and an update thread) could end up putting more than 4 statuses on the list.

String is SAFE because it is immutable.
(c) Suppose that we decide to use the *monitor pattern*. State in one sentence what changes we would make to `MutInfoEntry`.

**Solution.**

Use the keyword *synchronized* on every public instance method of `MutInfoEntry`.

Other ways of using *synchronized* (e.g. only on the methods that touch *statuses*, or using the lock on the `statuses` object rather than *this*) may indeed solve the race condition in part (a), but they are not the monitor pattern.

---

**Problem 2** (Recursive Datatypes) *(26 points)*.

Suppose we want to implement `ImInfoEntry` (an *immutable* information board entry) as a recursive data type with two variants. The two variants are called A and B.

The snapshot diagram below shows how the datatype represents an information board entry \(t\) with destination “BOSTON” and two statuses “11:05 AM” and “ON TIME”, whose current status is “11:05 AM”.

(a) Write a datatype definition that corresponds to the snapshot diagram and implements `ImInfoEntry`.

\[ \text{ImInfoEntry} = \]

**Solution.**

\[ \text{A(stat:String, rest:ImInfoEntry)} + \text{B(stat:String, dest:String)} \]

(b) Fill in the blanks to implement `destination()`, `status()`, and `size()` for variants A and B:

```java
public class A implements ImInfoEntry {
    ...
    public String destination() { return ________________________________ ; }

    public String status() { return ________________________________ ; }

    public int size() { return ________________________________ ; }
}
```
To help implement the nextEntry operation, we add one more variant C. The result of \( u = t.\text{nextEntry}() \) is shown in the snapshot diagram below.
(c) Fill in the blanks to implement nextEntry() for all three variants.

```java
public class A implements ImInfoEntry {
    ...
    public ImInfoEntry nextEntry() {
        return new C(this, rest);
    }
}

public class B implements ImInfoEntry {
    ...
    public ImInfoEntry nextEntry() {
        return ________________________________ ;
    }
}

public class C implements ImInfoEntry {
    ...
    public ImInfoEntry nextEntry() {
        if (this.curr.size() == 1) { // curr has reached the end of the list
            return ________________________________ ;
        } else {
            return ________________________________ ;
        }
    }
}
```

Solution.
From the example and code shown for `A.nextEntry()`, the datatype definition is now:

```java
ImInfoEntry = A(stat:String, rest:ImInfoEntry)  
+ B(stat:String, dest:String) 
+ C(restart:ImInfoEntry, curr:ImInfoEntry)
```

The C variant represents an information entry whose current status is `curr.status()` (i.e., possibly advanced farther down the list), and whose future statuses follow the rest pointers from curr until the end of the list (a B variant), and then loop back to `restart`.

So the given code for `A.nextEntry()` constructs a C whose `restart` is the A object and whose current status is the successor of the A object:

```java
public class A implements ImInfoEntry {
    ...
    public ImInfoEntry nextEntry() {
        return new C(this, rest);
    }
}
```

Here is the simplest answer for `B.nextEntry()`:

```java
public class B implements ImInfoEntry {
    ...
    public ImInfoEntry nextEntry() {
        return this;
    }
}
```

...because B represents an entry with only 1 status, so it immediately loops back to itself. Another possible answer is `new C(this, this)`.

Here is the simplest answer for `B.nextEntry()`:

```java
public class C implements ImInfoEntry {
    ...
    public ImInfoEntry nextEntry() {
        if (this.curr.size() == 1) { // curr has reached the end of the list
            return this.restart;
        }
        else {
            return new C(this.restart, this.curr.nextEntry());
        }
    }
}
```

As before, `new C(this.restart, this.restart)` would also work for the first return statement.

Note that C is an immutable object, so it's necessary to create a fresh C here rather than, say, reassigning `this.curr = this.curr.nextEntry()`.

Note also that `this.curr.rest` is not correct, because `this.curr` might refer to any `ImInfoEntry` variant – A, B, or even C. `this.curr` doesn't necessarily point to a B object, and so the `rest` instance variable doesn't necessarily exist on that object.
Problem 3 (Grammars) (22 points).

(a) Which of these regular expressions accept (fully match) every legal status and destination string, and reject (fail to fully match) at least one illegal string? Circle YES or NO.

\[
[A-Z0-9: ]+
\]
matches every legal string? YES NO
rejects at least one illegal string? YES NO

Solution.

NO to matching every legal string – it does not match the empty string, which is a legal status.
YES to rejecting at least one illegal string – for example, it rejects the illegal string ",".

\[
([A-Z]*|[0-9]*|:*| *)+
\]
matches every legal string? YES NO
rejects at least one illegal string? YES NO

Solution.

YES to matching every legal string. The parenthesized regex can match any legal character, and can also match the empty string. The parenthesized regex must then match at least once, because of the + operator applied to it, but because the parenthesized regex can match the empty string, this means that the overall regex can also match the empty string.
YES to rejecting at least one illegal string – for example, it rejects the illegal string ",".

\[
[A-Z]*[0-9]*[:]*[ ]*
\]
matches every legal string? YES NO
rejects at least one illegal string? YES NO

Solution.

NO to matching every legal string – it does not match "11:05 AM", for example, because once the colon has matched [:]*, there is no way to match the remaining digits and letters in the string.
YES to rejecting at least one illegal string – for example, it rejects the illegal string ",".
matches every legal string? YES NO
rejects at least one illegal string? YES NO

Solution.

YES to matching every legal string, because \(.*\) can match the empty string at the start, and then \([A-Z\theta-9:]\)* can match the rest of a legal string.

NO to rejecting at least one illegal string, because this regex actually matches all possible strings – \(.*\) can match the entire string first, and then \([A-Z\theta-9:]\)* can be satisfied by matching the empty string at the end.

(b) Suppose an information board entry is represented as a string of text as in this example:

WASHINGTON|NEW DEPRTURE,TRACK 2,11:35AM

Complete the grammar below so that it can be used to parse an information board entry, with starting nonterminal infoentry. Your grammar must use the destination and status nonterminals shown, which you can assume have been defined with a correct answer from part (a).

For the purpose of this grammar, assume that statuses and destinations have no maximum length, and an information board entry has no maximum number of statuses.

\[
\begin{align*}
\text{destination} &::= \text{a correct regular expression from part (a)} \\
\text{status} &::= \text{a correct regular expression from part (a)} \\
\text{infoentry} &::= \\
\end{align*}
\]

Solution.

\[
\begin{align*}
\text{infoentry} &::= \text{destination} | \text{status} ( \text{',', status} )^* \\
\end{align*}
\]

Here is another solution that avoids using the repetition operator \(^*\) by introducing a new nonterminal:

\[
\begin{align*}
\text{infoentry} &::= \text{destination} | \text{statuses} \\
\text{statuses} &::= \text{status} \ | \ \text{status} ( \text{',', status} )^* \\
\end{align*}
\]

And here is a solution that enforces the maximum number of statuses (which was not required by the instructions):

\[
\begin{align*}
\text{infoentry} &::= \text{destination} | \text{statuses} \\
\text{statuses} &::= \text{status} \ | \ \text{status} , \text{status} \\
& \ | \ \text{status} , \text{status} , \text{status} \\
& \ | \ \text{status} , \text{status} , \text{status} , \text{status} \\
\end{align*}
\]

Problem 4 (Map/Filter and Callbacks) (26 points).

Suppose we add map and filter operations to ImInfoEntry, to transform the (cyclic) stream of status messages that an information board entry displays:
map: ImInfoEntry x (String -> String) -> ImInfoEntry
filter: ImInfoEntry x (String -> Boolean) -> ImInfoEntry

These operations affect only the statuses of an ImInfoEntry, not its destination.

(a) Of the four kinds of ADT operations, what kind(s) of operations is ImInfoEntry.map? Leave extra boxes blank:

Solution.
Producer.

(b) Use map to replace every English status message found in the translations map below with its corresponding French translation.

Map<String, String> translations = Map.of("ON TIME", "A LHEURE", "CANCELED", "SUPPRIME");

ImInfoEntry train1 = ImInfoEntry.parse("MONTREAL|ON TIME,11:05 AM");
// train1 has statuses "ON TIME", "11:05 AM"

ImInfoEntry train2 = train1.map(...MAP...);
// train2 has statuses "A LHEURE", "11:05 AM"

Write a Java lambda expression for ...MAP... in the code above:

Solution.
status -> map.getOrDefault(status, status)

Alternatively:
status -> {
    if (map.containsKey(status)) {
        return map.get(status);
    } else {
        return status;
    }
}

(c) Write a Java lambda expression that, if passed to filter (not map), would transform the stream of status messages in a way that cannot be a legal abstract value of the ImInfoEntry type.

Solution.

This filter function would fail to match all possible status messages, which is illegal because an information entry must have at least one status:

status -> false
Now suppose that a mutable information board entry `MutInfoEntry` also has a `map` operation:

\[
\text{map: MutInfoEntry} \times (\text{String} \to \text{String}) \to \text{void}
\]

`MutInfoEntry.map` transforms all statuses subsequently returned by the entry, as shown in this example:

1. `Function<String, String> toFrench = ...MAP...;` // a correct answer to part (b) above
2. `MutInfoEntry train = new MutInfoEntry("MONTREAL");`
3. `train.nextStatus(); // returns ""
4. `train.map(toFrench);
5. train.setstatuses(List.of("ON TIME","11:05 AM"));
6. train.nextStatus(); // returns "A LHEURE"
7. train.nextStatus(); // returns "11:05 AM"
8. train.setstatuses(List.of("CANCELED"));
9. train.nextStatus(); // returns "SUPPRIME"

(d) What kind(s) of operation is `MutInfoEntry.map`? Leave extra boxes blank:

**Solution.**

Mutator.

Note that "observer" is not a good answer here, because the `map` operation by itself returns no information to the client, and doesn’t even necessarily call the client’s transform function with any statuses (yet).

To implement `map`, the rep of `MutInfoEntry` now has a third field:

**private** `Function<String, String> f;`

and its abstraction function is (only relevant parts shown):

\[
\text{AF} (\text{destination}, \text{statuses}, f) = \text{the info board entry with current status } f(\text{statuses[0]}) \text{ and looping through future statuses } f(\text{statuses[1]}),..., f(\text{statuses[statuses.length-1]}), f(\text{status[0]}), \text{and so on... [rest of AF elided]}
\]

The `MutInfoEntry` methods are implemented to obey this AF and behave as shown in the code above.

(e) Write a Java lambda expression for the initial value of `f` for a new `MutInfoEntry` object.

**Solution.**

It should be the identity function, e.g.:

\[
\text{status} \to \text{status}
\]

(f) During which of the numbered lines in the example code above is the `toFrench` function called? List all line numbers that apply, or write NEVER if `toFrench` is never called. Note that this question is asking about `toFrench`.
Solution.

Lines 6, 7, 9. toFrench must be called by every nextStatus() call after it is installed.

Line 3 is not correct because toFrench is not been called yet.

Line 4 is not correct because toFrench is passed just as a reference to the function, not a call.

Lines 5, and 8 are not correct because the abstraction function declares that statuses in the rep is a list of untransformed statuses. The map function should not be called when the statuses are stored, only when they are displayed. Calling toFrench during these lines is possible (perhaps by a fast-failing checkRep() method) but not necessary.

(g) What should MutInfoEntry's rep invariant comment say about f? Note that this question is asking about f.

Solution.

f is a function that requires a valid status as input (the precondition) and returns a valid status (postcondition). Without this constraint, the nextStatus() operation is unable to satisfy its postcondition.

A statement that includes only the postcondition – e.g. "f returns only valid statuses" – is too strong, because it excludes many valid functions. For example, the identity function can return an invalid status if it is given an invalid status.

A statement that speaks only about the current state of the object – e.g. "f(statuses[i]) is a valid status for all i" – is not strong enough. There may be some valid status s *not currently* in statuses where f(s) is invalid. setStatuses() would have to accept s as one of the new statuses, but this would break the rep.

A statement that repeats the type signature of f, e.g. "f is a function from strings to strings", is not strong enough, and is unnecessary to include in the rep invariant comment because the type declaration already states it.

You may detach this page. Write your username at the top, and hand in all pages when you leave.

/**
 * An information board entry that shows a destination (e.g. "WASHINGTON DC")
 * and cycles through a list of 1 to 4 statuses (e.g. [ "11:05 AM", "ON TIME" ],
 * or [ "NOW BOARDING", "TRACK 3" ]).
 * A valid destination is up to 16 characters, consisting only of
 * upper-case letters A-Z, digits, colons, or spaces.
 * A valid status is up to 12 characters, consisting only of
 * upper-case letters A-Z, digits, colons, or spaces.
 */

public class MutInfoEntry {

    private final String destination;
}
private List<String> statuses = Collections.synchronizedList(new ArrayList<String>());

// Abstraction function:
// <elided>
// Rep invariant:
// - destination is a valid destination (defined above)
// - statuses has 1-4 elements, each of which is a valid status (defined above)

/** Create a new information board entry with the given destination and
 * a single empty status.
 * @param destination a valid destination (defined above) */
public MutInfoEntry(String destination) {
    this.destination = destination;
    statuses.add("");
}

/** @return the destination */
public String destination() { return destination; }

/** @return the next status to display, infinitely cycling through this
 * info board entry's statuses in order */
public String nextStatus() {
    final String status = statuses.remove(0);
    statuses.add(status); // put it back on end so that statuses cycle forever
    return status;
}

/** Set the statuses. The first status in the list will be displayed next.
 * @param statuses new statuses, a 1- to 4-item list of valid statuses */
public void setStatuses(List<String> statuses) {
    this.statuses.clear();
    this.statuses.addAll(statuses);
}

You may detach this page. Write your username at the top, and hand in all pages when you leave.

/**
 * An information board entry that shows a destination (e.g. "WASHINGTON DC")
 * and current status (e.g. "DELAYED") in a cycle of 1 to 4 statuses
 * (e.g. [ "DELAYED", "NEW DEPRTURE", "11:55 AM" ]).
 * A valid destination is up to 16 characters, consisting only of
 * upper-case letters A-Z, digits, colons, or spaces.
 * A valid status is up to 12 characters, consisting only of
 * upper-case letters A-Z, digits, colons, or spaces.
 */
public interface ImInfoEntry {

    /** @return the destination */
public String destination();

/** @return the currently-shown status */
public String status();

/** @return the entry with the same destination and statuses,
 * showing the next status in the cycle */
public ImInfoEntry nextEntry();

/** @return number of statuses in the cycle, from 1 to 4 */
public int size();

/** @param entry information board entry represented as a string according
 * to the grammar in Problem 3
 * @return corresponding information board entry value */
public static ImInfoEntry parse(String entry) { ... }