MIT
6.031: Software Construction
Max Goldman and Prof. Rob Miller
revised Wednesday 4th December, 2019, 16:22

Quiz 2 (December 3, 2019)

Your name: _____

Your Kerberos username: _____

You have 50 minutes to complete this quiz. It contains 10 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 Kerberos username 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 letter(s) or entire answer(s). Do not use check marks, underlines, or other annotations – they will not be graded.

Good luck!

Problem	Points
1: Thread Safety	26
2: Recursive Datatypes	26
3: Grammars	22
4: Map/Filter and Callbacks	26
Total	100

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 9. **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.

display thread	update thread

bad outcome:

(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:

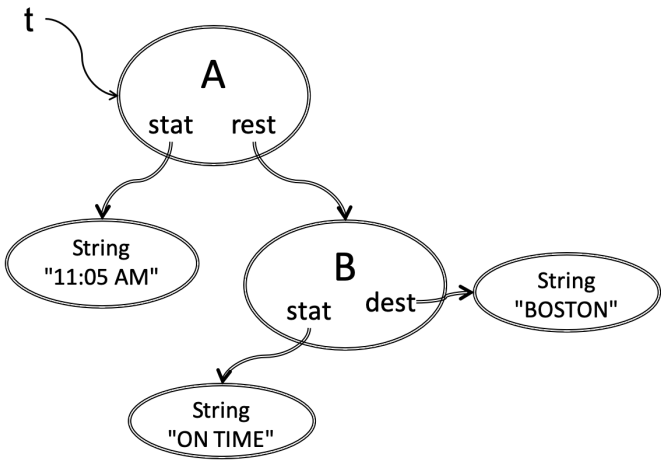
`String` rep invariant

DANGER SAFE because:

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

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".



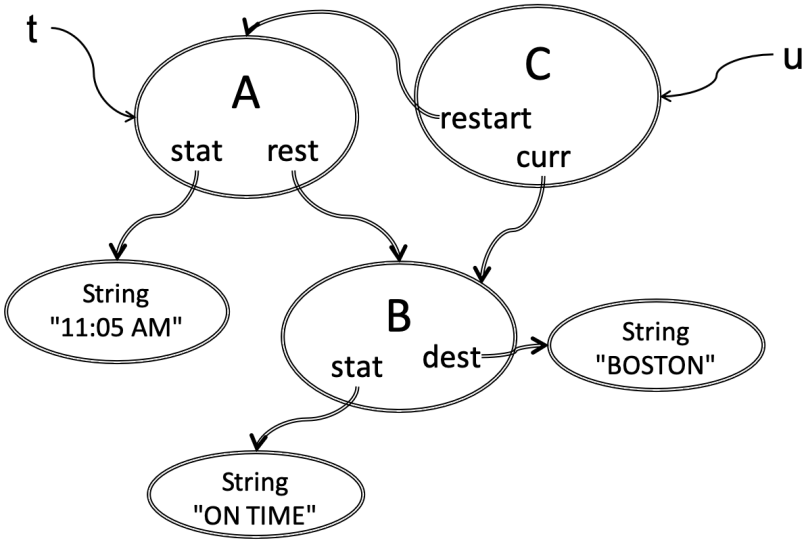
(a) Write a datatype definition that corresponds to the snapshot diagram and implements `ImInfoEntry`.

ImInfoEntry =

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

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

  public String status()      { return ____________________________________ ; }

  public int size()           { return ____________________________________ ; }
}

public class B 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.nextEntry()` is shown in the snapshot diagram below.



(c) Fill in the blanks to implement `nextEntry()` for all three variants.

```
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 ____________________________________________________________ ;
    }
  }
}
```

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

([A-Z]*|[0-9]*|:*| *)+
        matches every legal string?              YES      NO

        rejects at least one illegal string?     YES      NO

[A-Z]*[0-9]*[:]*[ ]*
        matches every legal string?              YES      NO

        rejects at least one illegal string?     YES      NO

.*[A-Z0-9: ]*
        matches every legal string?              YES      NO

        rejects at least one illegal string?     YES      NO
```

(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**.

destination ::= *a correct regular expression from part (a)*
status ::= *a correct regular expression from part (a)*

infoentry ::=

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:

(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:

(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.

Now suppose that a mutable information board entry `MutInfoEntry` also has a `map` operation:

`map: MutInfoEntry x (String -> String) -> 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:

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):

> AF(`destination`, `statuses`, `f`) = the info board entry with current status `f(statuses[0])` and looping through future statuses `f(statuses[1])`, ..., `f(statuses[statuses.length-1])`, `f(status[0])`, 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.

(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`**.

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

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) { ... }

}
```