CIS 1200 Final Exam May 7, 2024

Name: _____

PennKey (penn login id, e.g., `sweirich`): _____

I certify that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: _____ Date: _____

- Please wait to begin the exam until you are told it is time for everyone to start.
- When you begin, please start by writing your PennKey at the bottom of all the odd-numbered pages in the rest of the exam.
- There are 120 total points. The exam length is 120 minutes.
- For coding problems: aim for accurate syntax, but we will not grade your code style for indentation, spacing, etc.
- There are 17 pages in the exam and an Appendix for your reference. Please do not submit the Appendix.
- Do not spend too much time on any one question. Be sure to recheck all of your answers.
- **This exam is based around the implementation of the puzzle game called `Wordle`. Appendix A contains an overview of that game and its implementation. Please read that page before starting the exam itself.**
- Good luck!

1. **Warmup: OCaml BSTs and Java `TreeSet`s** (15 points)

   Before you start implementing your Wordle game, you want to make sure that you can transfer your knowledge from OCaml to Java, especially about binary search trees. In particular, you recall that the Java `TreeSet` class implements a mutable `Set`. **For reference, documentation about the `TreeSet` class appears in Appendix H.**

   You also remember that an OCaml type definition for BSTs looks like this:

```
(* Generic binary trees, from HW 3 *)
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

   and that you can create simple trees containing a single element with the following function:

```
let leaf (i:'a) : 'a tree = Node(Empty, i, Empty)
```

   You can also construct a larger tree with this code:

```
let t1 : int tree =
  Node (Node (leaf 3, 4, Node (leaf 5, 6, leaf 7)))
```

   Which we draw as:

```
t1 =    4
       / \
      3   6
         / \
        5   7
```

   Now, consider the following OCaml functions, each called `f`, that work with binary search trees.

   For each function, you will need to do three things (1) determine the result when applied to `t1` and other appropriate arguments, (2) identify the method of Java's `TreeSet` class that provides the most similar functionality to `f`, and (3) answer whether the function assumes the *binary search tree (BST) invariant* and would produce an incorrect result for trees that do not satisfy this property.

(a)
```
let rec f (t:'a tree) : int =
  begin match t with
    | Empty -> 0
    | Node (lt, v, rt) -> 1 + f lt + f rt
  end
```

   Result of `f t1`: _____

   Most similar `TreeSet` method: _____

   This function uses the BST invariant: True ☐ False ☐

Repeat definition of `t1`*:*

```
let t1 : int tree =
  Node (Node (leaf 3, 4, Node (leaf 5, 6, leaf 7)))
```

(b)
```
let rec f (t:'a tree): 'a =
  begin match t with
  | Empty -> failwith "no such element"
  | Node (lt, v, Empty) -> v
  | Node (lt, v, rt) -> f rt
  end
```

Result of `f t1`: _____

Most similar `TreeSet` method: _____

This function uses the BST invariant: True ☐ False ☐

(c)
```
let rec f (t:'a tree) : bool =
  begin match t with
  | Empty -> true
  | _ -> false
  end
```

Result of `f t1`: _____

Most similar `TreeSet` method: _____

This function uses the BST invariant: True ☐ False ☐

(d)
```
let rec f (t:'a tree) (n:'a) : bool =
  begin match t with
    | Empty -> false
    | Node(lt, x, rt) ->
      x = n || if n < x then f lt n else f rt n
  end
```

Result of `f t1 6`: _____

Most similar `TreeSet` method: _____

This function uses the BST invariant: True ☐ False ☐

(e)
```
let rec f (t:'a tree) (x: 'a) (y : 'a) : 'a tree =
  begin match t with
  | Empty -> Empty
  | Node (lt , v , rt) ->
    if v >= x && v < y then
      Node (f lt x y, v, f rt x y)
    else if v < x then f rt x y
    else f lt x y
  end
```

Result of `f t1 4 6`: _____

Most similar `TreeSet` method: _____

This function uses the BST invariant: True ☐ False ☐

2. **Understanding OO Class Definitions** (21 points)

The following questions refer to the classes defined in Appendix B that represent the guessed letters in the *Wordle* game and to the `Set` interface and `TreeSet` class from the Collections Framework (see Appendix H).

Which of the following code blocks is legal Java code that will not cause any compile-time (i.e. type checking) or run-time errors? If it is legal code, check the "Legal Code" box and answer the questions that follow it. If it is not legal, check one of the "Not Legal" options and explain why. You can assume each block below is independent and written in some static method defined in the `Wordle` class and that the appropriate imports have been made at the top of the file.

(a)

```
Letter p = Letter.BLANK;
Letter q = new Letter(' ');
boolean ans = (p == q);
```

☐ Legal Code

A. The static type of `p` is _____ .

B. The dynamic class of `p` is _____ .

C. The value of `ans` is _____ .

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(b)

```
Letter p = new CheckedLetter('A', Color.BLACK);
String q = "A";
boolean ans = p.toString().equals(q);
```

☐ Legal Code

A. The static type of `p` is _____ .

B. The dynamic class of `p` is _____ .

C. The value of `ans` is _____ .

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(c)

```
Letter p = new Letter('A');
Letter q = new CheckedLetter('A', Color.BLACK);
boolean ans = q.equals(p);
```

□ Legal Code

A. The static type of `p` is _____ .

B. The dynamic class of `p` is _____ .

C. The value of `ans` is _____ .

□ Not Legal — Will compile, but will throw an `Exception` when run

□ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(d)

```
Letter p = new Letter(' ');
boolean ans = p.equals(p);
```

□ Legal Code

A. The static type of `p` is _____ .

B. The dynamic class of `p` is _____ .

C. The value of `ans` is _____ .

□ Not Legal — Will compile, but will throw an `Exception` when run

□ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(e)

```
Set<String> tree = new TreeSet<String>();
tree.add("A");
boolean ans = tree.contains("A");
```

□ Legal Code

A. The static type of `tree` is _____ .

B. The dynamic class of `tree` is _____ .

C. The value of `ans` is _____ .

□ Not Legal — Will compile, but will throw an `Exception` when run

□ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(f)

```
Set<CheckedLetter> tree = new TreeSet<CheckedLetter>();
tree.add(Letter.BLANK);
boolean ans = tree.contains(Letter.BLANK);
```

- ☐ Legal Code
  - A. The static type of `tree` is _____ .
  - B. The dynamic class of `tree` is _____ .
  - C. The value of `ans` is _____ .
- ☐ Not Legal — Will compile, but will throw an `Exception` when run
- ☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(g) Note, the `first` method is part of the `TreeSet` class (Appendix H) and the `toUpperCase` method is part of the `String` class (not shown).

```
Set<Object> tree = new TreeSet<Object>();
tree.add("A");
int ans = tree.first().toUpperCase();
```

- ☐ Legal Code
  - A. The static type of `tree` is _____ .
  - B. The dynamic class of `tree` is _____ .
  - C. The value of `ans` is _____ .
- ☐ Not Legal — Will compile, but will throw an `Exception` when run
- ☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

3. **Game Model and Invariants** (15 points)

The state of the game is stored by the `Model` class shown in Appendix C. This class includes the following instance variables.

```
private final Letter[][] board;   // the game board
private final String secretWord;  // word the player is trying to guess
private final int numGuesses;     // how many total guesses (usually 6)

private boolean hasWon;       // did the player win?
private int currGuess;        // which guess are they on?
private int currLetter;       // how many letters have they entered?
```

The `board` is a two-dimensional array that stores each letter that the player guesses, indexed by guess number and position within each guess. As the player types letters on the keyboard, the fields `currGuess` and `currLetter` track how many guesses the player has tried and how many letters have been entered for the current guess. If `currLetter` is `5`, then the player has entered a complete word for their current guess and if `currGuess` is equal to `numGuesses` then the player has lost.

For example, in the state shown in Figure 1 in Appendix A, `currGuess` is equal to `3`, `currLetter` is equal to `4`, and the game is still in progress so `hasWon` is **`false`**. The first three rows of the `board` contain instances of class `CheckedLetter`, while the remaining rows contain instances of class `Letter`. After the player hits enter, in the state in Figure 2, the current word has been checked, so the first *four* rows of the board contain instances of class `CheckedLetter`. Furthermore, the guess the correct word, so `hasWon` is `True`.

Which of the properties below make good invariants for the class `Model`? In other words, which properties should be true after the instance variables have been initialized and remain true throughout the execution of the application?

(a) True ☐ False ☐ `secretWord` is a reference to a `String` containing 5 uppercase characters

(b) True ☐ False ☐ if `hasWon` is **`true`**, then `currGuess < numGuesses`

(c) True ☐ False ☐ `0 <= currGuess < numGuesses`

(d) True ☐ False ☐ `board` is not **`null`**

(e) True ☐ False ☐ if `0 <= currGuess < numGuesses`, then `board[currGuess]` is a reference to an array of length 5

(f) True ☐ False ☐ if `0 <= currGuess < numGuesses` and `0 <= currLetter < 5` then `board[currGuess][currLetter]` is not `null`

(g) True ☐ False ☐ if `board[currGuess][currLetter]` is a reference to an object, then its dynamic class is `Letter`

Note that all of the instance variables of the `Model` class are marked **`private`**. But, is this enough to encapsulate the game state? For each of the access methods below, determine whether it preserves encapsulation or could be used by a client to modify the game state and perhaps violate an invariant!

(h)

```
public int getCurrLetter() { return currLetter; }
```

This method preserves encapsulation: True □ False □

(i)

```
public String getSecretWord() {
    return secretWord;
}
```

This method preserves encapsulation: True □ False □

(j)

```
public Letter[] getGuess() {
    return board[currGuess];
}
```

This method preserves encapsulation: True □ False □

(k)

```
public Letter getLetter(int i, int j) { return board[i][j]; }
```

This method preserves encapsulation: True □ False □

4. **Unit Testing** (15 points)

   Make sure that you have read through the `Model` class shown in Appendix C.

   This state of the `Wordle` game should only only be modified by the three methods: `add`, `back`, and `check`. These methods update the mutable instance variables of the `Model` class as the user adds new letters, removes their last letter, and registers their current guess.

   The following unit test demonstrates game play and methods of the `Model` class.

```
@Test
void testWin() {
    Model model = new Model("OCAML", 1); // game with one guess
    model.add('O');
    model.add('C');
    model.add('A');
    model.add('M');
    model.add('L');
    assertEquals(model.getCurrLetter(), 5); // user typed 5 letters
    assertTrue(model.playing());   // guess hasn't yet been checked
    assertTrue(model.check());     // OCAML is a valid word
    assertTrue(model.playerWin()); // user won the game
}
```

   (a) Complete the following unit test for the player losing the game. Note that "STACK" is a valid word.

```
@Test
void testLoss () {
   Model model = new Model("OCAML", 1);  // game with one guess
   model.add('S');
   model.add('T');
   model.add('A');
   model.add('C');
   model.add('K');

   _____________________________________(model.check());

   _____________________________________(model.playing());

   assertEquals (model.getCurrGuess(), ________________);

   assertEquals (model.getCurrLetter(), ________________);

}
```

(b) Complete the following unit test for the `back` method, which is called when the user types the backspace character while the game is still in progress.

```
@Test
void testAddBack() {
    Model model = new Model("OCAML", 1);  // game with one guess
    model.add('O');
    model.add('C');
    model.back();    // remove 'C'

    assertEquals(model.getLetter(0, 0).getChar(), ____________________);

    assertEquals(model.getLetter(0, 1), ____________________________);

    assertEquals(model.getCurrLetter(), ____________________);

    assertEquals(model.getCurrGuess(), ____________________);

}
```

(c) Complete the definition of the `back` method. This method removes the player's most recently typed letter of their current guess. (If the player has not typed any letters, then the method has no effect.)

```
/* This method assumes that model.playing() is true. */
public void back() {









}
```

(d) Consider this definition of the `add` method.

This method adds the new character `c` as a `Letter` to `board` as long as the user has not completed their current word. If the current word is complete, this method has no effect.

```
/* This method may assume that c is a letter character and that
   the game is still playing. */

    public void add(char c) {
        if (currLetter < 5) {
            board[currGuess][currLetter] = new Letter(c);
            currLetter++;
        }

    }
```

What would happen if this method were called with the `'*'` character (i.e. a nonletter character)?

What would happen if this method were called after the player has lost (i.e. if this method is called after `check`, and the player is out of guesses.)

5. **Java True/False** (15 points)

The following questions refer to the structure of the `Wordle` class, shown in Appendix A.

(a) True □ False □ The type `View` is a subtype of `Object`.

(b) True □ False □ The class `View` is a subclass of `Wordle`.

(c) True □ False □ The methods and constructors in class `View` may refer to the private instance variable `board` of class `Model` (see Appendix C).

(d) True □ False □ The use of `SwingUtilities.invokeLater` is a static method call.

The following questions refer to the inner class `View`, shown in Appendix D.

(f) True □ False □ The constructor for the `View` class is `static` which is why it can be used to initialize the `view` instance variable in the `Wordle` class.

(g) True □ False □ The call to `super.paintComponent` on line 16 refers to a member of class `JPanel`.

(h) True □ False □ Line 21 is an example of the use of dynamic dispatch.

The following questions refer to the inner class `Listener` of the `Wordle` class, shown in Appendix E, and to the classes of the Java Swing library. For reference, documentation for the Swing library appears in Appendix J.

(i) True □ False □ The class `Listener` is a subclass of `JPanel`.

(j) True □ False □ The type `Listener` is a subtype of `KeyListener`.

(k) True □ False □ The type `KeyListener` is a subtype of `Object`.

(l) True □ False □ The class `Listener` inherits a method called `keyReleased`.

(m) True □ False □ The methods and constructors in class `Listener` may refer to the private instance variables of class `Wordle`.

(n) We can add a button to restart the game by modifying the `Wordle` constructor (shown in appendix E) to include the following code after the definition of `statusPanel`. What single line of code should we add to this definition? Pressing the `"new game"` button should restart the game using `"FINAL"` as the secret word and giving the player six guesses.

```
JButton restart = new JButton("new game");
statusPanel.add(restart);
restart.addActionListener( e -> {

    _____________________________________________________________________________
    view.repaint();
    panel.requestFocusInWindow();
});
```

6. **I/O and Java Iterators** (19 points)

The Wordle game checks to make sure that each five letter word entered by the user is a valid word in the dictionary. But what words are valid?

To answer this question, the `Wordle` class maintains a collection of `VALID_WORDS` in a static instance variable. It uses the static method `makeValidWords` to initialize this collection.

Recalling the `WordScanner` example from class (see Appendix F), you decide to implement an iterator that will help you pull out five letter words from some source file as part of the implementation of `makeValidWords` (see Appendix G).

In this problem, you will define a `FilterIterator` that you can use to compositionally build the iterator that you need out of a `WordScanner` and a *predicate* object.

A *predicate* object is an instance of the following interface:

```
public interface Predicate<E> {
    boolean test(E x);
}
```

Predicate objects act as first-class testing functions. In this case, the predicate determines whether an element should be returned by the `FilterIterator`. For example, you can create a `FilterIterator` that returns only those strings produced from a `WordScanner` that have five letters as in the following test code.

```
@Test
public void testFilter() {
    Reader r = new StringReader("one.three.two");
    WordScanner ws = new WordScanner(r);
    FilterIterator fi = new FilterIterator(ws, new Predicate<String> () {
        @Override
        public boolean test(String x) {
            return (x.length() == 5);
        }
    });
    assertTrue(fi.hasNext());
    assertEquals("three", fi.next());
    assertFalse(fi.hasNext());
}
```

On the next page, complete the implementation of the `FilterIterator`. This iterator should should produce only those values from the underlying iterator that satisfy the predicate. **You may assume that the underlying iterator never produces `null` values.**

(There is nothing to answer on this page.)

```
public class FilterIterator<E> implement Iterator<E> {

    private Iterator<E> it;
    private Predicate<E> f;

    // TODO: add additional instance variables here




    public FilterIterator(Iterator<E> it, Predicate<E> f) {
        if (it == null || f == null) {
          throw new IllegalArgumentException();
        }
        this.it = it;
        this.f = f;
        // TODO: finish constructor









    }


    // TODO: implement hasNext and next methods, as well as any helper
    // methods. You may also use the next page if you need more space,
    // but please clearly indicate on this page if you do.

}
```

Additional space for implementation of `FilterIterator`.

7. **Exceptions, Collections and Polymorphism** (20 points)

As part of the `makeValidWords` method, shown in Appendix G, the `FilterIterator` can be used to add all valid words to the `validWords` collection.

(a) Which operations inside the **`try`** block (lines 5-15) could potentially thow a `FileNotFoundException` or an `IOException`? Mark all that apply. Documentation for the `FileReader` class appears in Appendix I.

- ☐ `new FileReader("notes.txt")`
- ☐ `new WordScanner(r)`
- ☐ `new FilterIterator<>(ws, predicate)`
- ☐ `fs.hasNext()`
- ☐ `fs.next()`
- ☐ `r.close()`

(b) Which of the following classes would it be correct and efficient to use to fill in the blank on line 3? (Here, we consider a collection to be efficient if it is possible to determine that a word is invalid without searching the entire collection.) Mark all that apply.

- ☐ `ArrayList<String>`
- ☐ `LinkedList<String>`
- ☐ `TreeSet<String>`
- ☐ `TreeMap<Integer,String>`
- ☐ `HashSet<String>`
- ☐ `Object`

(c) Which of the following lines in the definition of `makeValidWords` is an example of the use of subtype polymorphism? Mark all that apply.

- ☐ `Reader r = new FileReader("notes.txt");`
- ☐ `WordScanner ws = new WordScanner(r);`
- ☐ `Predicate<String> predicate = ((String x) -> x.length() == 5);`
- ☐ `FilterIterator<String> fs = new FilterIterator<>(ws, predicate);`

(d) Which of the following lines in the definition of `makeValidWords` is an example of the use of generics (i.e. parametric polymorphism)? Mark all that apply.

- ☐ `Reader r = new FileReader("notes.txt");`
- ☐ `WordScanner ws = new WordScanner(r);`
- ☐ `Predicate<String> predicate = ((String x) -> x.length() == 5);`
- ☐ `FilterIterator<String> fs = new FilterIterator<>(ws, predicate);`