CIS 1200 Final Exam May 7, 2024

SOLUTIONS

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 (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`: 5

   Most similar `TreeSet` method: *size*

   This function uses the BST invariant: True ☐ False ☒

Repeat definition of `t1`*:*

```
let t1 : int tree =
   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`: `7`

Most similar `TreeSet` method: *last*

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`: **`false`**

Most similar `TreeSet` method: *isEmpty*

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`: **`true`**

Most similar `TreeSet` method: *contains*

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`: `Node (Empty, 4, leaf 5)`

Most similar `TreeSet` method: *subset*

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 `Letter`.

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

C. The value of `ans` is **`false`**.

☐ 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 `Letter`.

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

C. The value of `ans` is **`true`**.

☐ 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 `Letter`.

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

C. The value of `ans` is **`false`**.

☐ 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):

_

NOTE: All answers received points for this question because the ASM lectures do not discuss the behavior of == on objects with different dynamic classes.

(d)

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

☒ Legal Code

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

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

C. The value of `ans` is **`true`**.

☐ 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 `Set<String>`.

B. The dynamic class of `tree` is `TreeSet<String>`.

C. The value of `ans` is **`true`**.

☐ 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):

_

NOTE: the String class implements the Comparable interface.

(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):
Cannot add object of type `Letter` *to set containing* `CheckedLetter`

Note that even though the `CheckedLetter` *does not implement the* `Comparable` *method, this is not the source of the compilation error.*

(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):
There are multiple type errors.
The type of tree is `Set`*, which doesn't contain a* `first` *method.*
The type of `tree.first()` *is* `Object`*, so cannot invoke* `toUpperCase` *method. And* `toUpperCase` *returns a* `String`*.*

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`
NOTE: If the player loses, then `currGuess` *equals* `numGuesses` *as described above*

(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`
This question was removed from the exam and all answers accepted.

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 ☐

NOTE: Strings are immutable so the game state cannot be modified.

(j)

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

This method preserves encapsulation: True ☐ False ☒

NOTE: The array of letters could be modified once returned.

(k)

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

This method preserves encapsulation: True ☒ False ☐

NOTE: The fields of Letter and CheckedLetter are final.

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);
    model.add('S');
    model.add('T');
    model.add('A');
    model.add('C');
    model.add('K');
    assertTrue(model.check()); // Stack is a valid word
    assertFalse(model.playing());
    assertEquals(model.getCurrGuess(), 1);
    assertEquals(model.getCurrLetter(), 0);

}
```

(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);
    model.add('O');
    model.add('C');
    model.back();
    assertEquals(model.getLetter(0, 0).getChar(), 'O');
    assertEquals(model.getLetter(0, 1), Letter.BLANK);
    assertEquals(model.getCurrLetter(), 1);
    assertEquals(model.getCurrGuess(), 0);
}
```

(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.)

```
public void back() {
    if (currLetter > 0) {
        currLetter--;
    }
    board[currGuess][currLetter] = Letter.BLANK;
}
```

NOTE: currLetter is the index after the most recently entered letter. So the decrement needs to happen before updating the board.

(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)?

ANSWER: if `c` *is not a space or a letter character, then the* `Letter` *constructor would throw an* `IllegalArgumentException`

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.)

ANSWER: if the player has lost, then at the end of the `check` *method,* `currGuess` *would be equal to* `numGuesses` *and* `currLetter` *would be equal to 0. If this method is called in this state, it will throw an* `ArrayIndexOutOfBounds` *exception.*

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`. *View is an innerclass not a subclass*

(c) True ☐ False ☒ The methods and constructors in class `View` may refer to the private instance variable `board` of class `Model` (see Appendix C). *Private members can only be accessed in the same class or inner classes. As View is an innerclass of* `Wordle`*, it cannot access the private members of* `Model`.

(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`. *The superclass of View is JPanel.*

(h) True ☒ False ☐ Line 21 is an example of the use of dynamic dispatch. *The dynamic class determines which* `draw` *method will be called. For some positions in the board, the dynamic class could be* `CheckedLetter`.

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`. *Listener extends KeyAdapter*

(j) True ☒ False ☐ The type `Listener` is a subtype of `KeyListener`. *Listener extends KeyAdapter, which implements KeyListener*

(k) True ☒ False ☐ The type `KeyListener` is a subtype of `Object`. *All interfaces are subtypes of Object*

(l) True ☒ False ☐ The class `Listener` inherits a method called `keyReleased`. *The superclass KeyAdapter includes this method, so it is inherited by Listener*

(m) True ☒ False ☐ The methods and constructors in class `Listener` may refer to the private instance variables of class `Wordle`. *Listener is an inner class of Wordle so has access to its private members*

(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.

```
Button restart = new JButton("new game");
statusPanel.add(restart);
restart.addActionListener( e -> {
            model = new Model("FINAL", NUM_GUESSES);
            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);
    FilterIterator2 fi = new FilterIterator2(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> implements Iterator<E> {
    private Iterator<E> it;
    private Predicate<E> f;

    public E nextElement;  // value of next element to return

    // this version only works if it.next() never returns null

    FilterIterator(Iterator<E> it, Predicate<E> f) {
        this.it = it;
        this.f = f;
        // need to find the first element that satisfies the predicate
        findNext();
    }

    // look at elements in the underlying iterator until one
    // that satisfies the test is found, or until there are no more
    // elements. If nextElement is non null then there is another element.
    private void findNext() {
        nextElement = null;
        while (it.hasNext() && nextElement == null) {
            E n = it.next();
            if ( f.test(n) ) {
                nextElement = n;
            }
        }
    }

    @Override
    public boolean hasNext() {
        return (nextElement != null);
    }

    @Override
    public E next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        E n = nextElement;
        findNext();
        return n;
    }
}
```

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")`
  Documentation indicates that the constructor could throw FileNotFoundException
- ☐ `new WordScanner(r)`
- ☐ `new FilterIterator<>(ws, predicate)`
- ☐ `fs.hasNext()`
- ☐ `fs.next()`
- ☒ `r.close()`
  Documentation indicates that the constructor could throw IOException

(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>`
  Membership testing is inefficient if there are many words
- ☐ `LinkedList<String>`
  Membership testing is inefficient if there are many words
- ☒ `TreeSet<String>`
- ☐ `TreeMap<Integer,String>`
  Doesn't have the right functionality
- ☒ `HashSet<String>`
- ☐ `Object`
  Doesn't have the right functionality

(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");`
  FileReader is a subtype of Reader
- ☐ `WordScanner ws = new WordScanner(r);`
  The constructor is called with an argument of static type Reader.
- ☒ `Predicate<String> predicate = ((String x) -> x.length() == 5);`
  The anonymous inner class is a subtype of `Predicate<String>`
- ☐ `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);`

A Wordle overview

The *Wordle* game gives a player six chances to guess a five letter word. After each guess, if the word is valid, the game indicates which of the letters of the word are correct (shown in green), are misplaced (shown in yellow), or are incorrect (shown in dark gray). If after the sixth guess, the player has not determined the word, then the player loses.
A sample game in progress is shown below on the left. By typing the letter 'L', the player completes the guess of the word "OCAML", thus winning the game (as shown on the right).

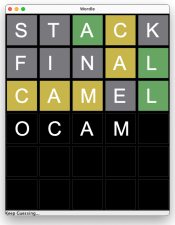

Figure 1: *Wordle* game in progress. The player has made three guesses.

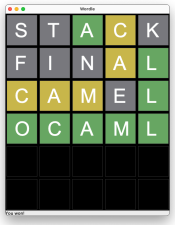

Figure 2: The player types 'L' and hits enter to win.

Take a moment now to familiarize yourself with the code below and in Appendices C-E. This code is used by the class `Wordle`, shown below.

```
public class Wordle {
   public static final int NUM_GUESSES = 6;
   public static final Color CORRECT = new Color(108,169,101); // green
   public static final Color MISPLACED = new Color (200,182,83); // yellow
   public static final Color WRONG = new Color(120,124,127); // dark gray

   // Appendix G: collection of valid words
   public static final Collection<String> VALID_WORDS = makeValidWords();
   private static Collection<String> makeValidWords() { ... }

   // Appendix C: game logic
   private Model model = new Model("OCAML", NUM_GUESSES);

   // Appendix D: game display, inner class View
   private View view = new View();
   private JLabel statusLabel = new JLabel("Guess a letter");
   private class View extends JPanel { ... }

   // Appendix E: inner class Listener and Wordle Constructor
   private class Listener extends KeyAdapter { ... }
   public Wordle() { ... }

   public static void main(String[] args) {
       SwingUtilities.invokeLater(() -> new Wordle());
   }
}
```

B Wordle: Letter and CheckedLetter classes

The following class declarations are in the same package as the `Wordle` class and are used to represent the letters that the player guesses during the game.

```
1  // Represents either a blank or a letter entered by the user
2  public class Letter {
3      public static final Letter BLANK = new Letter(' ');
4
5      private final char data; // must be uppercase letter or blank
6
7      public Letter (char ch) {
8          if (!(ch == ' ' || Character.isLetter(ch))) {
9              throw new IllegalArgumentException();
10         }
11         this.data = Character.toUpperCase(ch);
12     }
13
14     public char getChar() { return data; }
15
16     public void draw (Graphics g, int x, int y) {
17         // draw the letter
18         g.setColor(Color.WHITE);
19         g.setFont(new Font("Arial", Font.PLAIN, 60));
20         g.drawChars(...);
21
22         // draw a border around the letter
23         g.setColor(Color.DARK_GRAY);
24         g.drawRect(...);
25     }
26
27     @Override
28     public String toString() { return String.valueOf(data); }
29
30     @Override
31     public boolean equals(Object o) { return (this == o); }
32 }
```

```
1  // Represents a letter in a guess that has been checked against the secret word
2  public class CheckedLetter extends Letter {
3      private final Color color;
4
5      public CheckedLetter(char c, Color color) {
6          super(c);
7          this.color = color;
8      }
9
10     @Override
11     public void draw (Graphics g, int x, int y) {
12         g.setColor(color);
13         g.fillRect(...);    // draw the background
14         super.draw(g,x,y);  // draw the letter and border
15     }
16 }
```

C Wordle: Game Logic (Model)

```
public class Model {
    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?

    // is the game still in progress
    public boolean playing() { return (!hasWon && currGuess < numGuesses); }
    public boolean playerWin() { return hasWon; }
    public boolean playerLoss() { return currGuess >= numGuesses; }

    // getters
    public Letter getLetter(int i, int j) { return board[i][j]; }
    public Letter[] getGuess() { return board[currGuess]; }
    public String getSecretWord() { return secretWord;  }
    public int getCurrGuess() { return currGuess; }
    public int getCurrLetter() { return currLetter; }

    // constructor
    public Model(String word, int numGuesses) {
        if (word.length() != 5 || numGuesses < 0) {
            throw new IllegalArgumentException("invalid args");
        }
        for (char c : word.toCharArray()) {
            if (!Character.isLetter(c) || !Character.isUpperCase(c)) {
                throw new IllegalArgumentException("invalid word");
            }
        }
        this.secretWord = word;
        this.numGuesses = numGuesses;
        this.board = new Letter[numGuesses][];
        for (int i = 0; i < numGuesses; i++) {
            board[i] = new Letter[5];
            for (int j = 0; j < 5; j++) {
                board[i][j] = Letter.BLANK;
            }
        }
        this.hasWon = false;
        this.currGuess = 0;
        this.currLetter = 0;
    }

    // update the board with a new letter added by the player
    public void add(char c) { // not shown }

    // remove the most recently entered letter by the player
    public void back() { // not shown  }
```

(`Model` class continued)

```
    public static String letterArrayToString(Letter[] letters) {
        char[] arr = new char[letters.length];
        for (int i = 0; i < letters.length; i++) {
            arr[i] = letters[i].getChar();
        }
        return new String(arr);
    }

    // if the current guess has enough letters, return whether it is valid
    // if so, update the letters in the guess to CheckedLetters and
    // also either set hasWon to true or advance to the next guess
    public boolean check() {
        if (currLetter != secretWord.length()) { return false; }

        Letter[] curr = board[currGuess];

        if (!Wordle.VALID_WORDS.contains(letterArrayToString(curr))) {
            return false;
        }

        hasWon = true;
        for (int i = 0; i < secretWord.length(); i++) {
            char c = curr[i].getChar();
            if (c == secretWord.charAt(i)) {
                curr[i] = new CheckedLetter(c, Wordle.CORRECT);
            } else if (secretWord.indexOf(c) != -1) {
                curr[i] = new CheckedLetter(c, Wordle.MISPLACED);
                hasWon = false;
            } else {
                curr[i] = new CheckedLetter(c, Wordle.WRONG);
                hasWon = false;
            }
        }

        if (!hasWon) {
            currGuess++;
            currLetter = 0;
        }
        return true;
    }

} // end of class Model
```

D Wordle:View inner class

The following class is a member of class `Wordle`, defined on page 1, and is related to the display of the game.

```
1  // inner class of Wordle
2  //
3  private class View extends JPanel {
4
5      public View() {
6          setBackground(Color.BLACK);
7      }
8
9      @Override
10     public Dimension getPreferredSize() {
11         return new Dimension(Letter.SIZE * 5, Letter.SIZE * NUM_GUESSES);
12     }
13
14     @Override
15     public void paintComponent(Graphics g) {
16         super.paintComponent(g);
17
18         for (int i = 0; i < NUM_GUESSES; i++) {
19             for (int j = 0; j < 5; j++) {
20                 Letter l = model.getLetter(i,j);
21                 l.draw(g,j*Letter.SIZE, i*Letter.SIZE);
22             }
23         }
24     }
25 }
```

E Wordle:Listener inner class and Wordle Constructor

The following declarations are members of class `Wordle`, defined in Appendix 1, and are related to the GUI and its interaction.

```
// instance variables for GUI
private JLabel statusLabel = new JLabel("Guess a letter");
private View view = new View();

class Listener extends KeyAdapter {
    @Override
    public void keyTyped(KeyEvent e) {
        if (model.playing()) {
            char c = e.getKeyChar();
            if (Character.isLetter(c)) {
                model.add(Character.toUpperCase(c));
            } else if (c == '\b') {    // backspace char
                model.back();
                statusLabel.setText("Keep Guessing...");
            } else if (c == '\n') {   // newline char
                if (!model.check()) {
                    statusLabel.setText("Invalid word");
                }
                if (model.playerWin()) {
                    statusLabel.setText("You won!");
                }
                if (model.playerLoss()) {
                    statusLabel.setText("You lost!");
                }
            }
            view.repaint();
        }
    }
}

public Wordle() {
    JFrame frame = new JFrame("Wordle");
    JPanel panel = new JPanel();
    JPanel statusPanel = new JPanel();

    frame.setContentPane(panel);
    panel.setLayout(new BorderLayout());
    panel.add(view, BorderLayout.CENTER);
    statusPanel.add(statusLabel);
    panel.add(statusPanel, BorderLayout.PAGE_END);
    panel.addKeyListener(new Listener());

    frame.pack();
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    panel.requestFocusInWindow();
}
```

F WordScanner class

```
public class WordScanner implements Iterator<String> {
    private final Reader r;  // reference to the input source
    private int c;  // invariant: c == -1 if there is no next word, or
    // Character.isLetter(c) == true and c is the start of the next word

    public WordScanner(Reader r) {
        if (r == null) {
            throw new NullPointerException();
        }
        this.r = r;
        try {
            this.c = r.read();
            skipNonLetters();
        } catch (IOException e) {
            this.c = -1;
        }
    }

    private void skipNonLetters() throws IOException {
        while (c != -1 && !Character.isLetter(c)) {
            c = r.read();
        }
    }

    @Override
    public boolean hasNext() {
        return (c != -1);
    }

    @Override
    public String next() {
        StringBuffer ans = new StringBuffer();
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        try {
            while (Character.isLetter(c)) {
                ans = ans.append((char)c);
                c = r.read();
            }
            skipNonLetters();
        } catch (IOException e) {
            c = -1;
        }
        return ans.toString();
    }
}
```

G Wordle: makeValidWords

```
1  private static Collection<String> makeValidWords() {
2
3     Collection<String> words = ____________________________________;
4
5     try {
6          Reader r = new FileReader("notes.txt");
7          WordScanner ws = new WordScanner(r);
8          Predicate<String> predicate = ((String x) -> x.length() == 5);
9          FilterIterator<String> fs = new FilterIterator<>(ws, predicate);
10         while (fs.hasNext()) {
11             String word = fs.next().toUpperCase();
12             words.add(word);
13         }
14         r.close();
15
16     } catch (FileNotFoundException ex) {
17         System.out.println("Cannot find file");
18     } catch (IOException ex) {
19         System.out.println("IOException occurred");
20     }
21     return words;
22 }
```

H Java Docs: collections

class `TreeSet<E>` **implements** `Set<E>`

---

public boolean `add(E e)`

Adds the specified element to this set if it is not already present.

public void `clear()`

Removes all of the elements from this set.

public `Object clone()`

Returns a shallow copy of this TreeSet. (The elements themselves are not cloned.)

public boolean `contains(Object o)`

Returns **true** if this set contains the specified element. More formally, returns true if and only if this set contains an element e such that (o==null ? e==null : o.equals(e)).

public `E first()`

Returns the first (lowest) element currently in this set.

public `E last()`

Returns the last (highest) element currently in this set.

public `E ceiling(E e)`

Returns the least element in this set greater than or equal to the given element, or null if there is no such element.

public boolean `isEmpty()`

Returns **true** if the set contains no elements.

public int `size()`

Returns the number of elements in this set.

public `SortedSet<E> subSet(E fromElement, E toElement)`

Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive.

I Java Docs: Iterator and IO

interface Iterator<E>

---

boolean hasNext()

Returns true if the iteration has more elements. (In other words, returns true if `next()` would return an element rather than throwing an exception.)

E next()

- **Returns:** the next element in the iteration
- **Throws:** `NoSuchElementException` - if the iteration has no more elements

class IOException **extends** Exception

Signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations.

class FileNotFoundException **extends** IOException

Signals that an attempt to open the file denoted by a specified pathname has failed.

class FileReader **implements** Reader

FileReader(String fileName)**throws** FileNotFoundException

Creates a new FileReader, given the name of the file to read from. Throws: FileNotFoundException - if the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

int read()**throws** IOException

Reads a single character. Returns: The character read, or -1 if the end of the stream has been reached. Throws: IOException - If an I/O error occurs.

public void close()**throws** IOException

Closes the stream and releases any system resources associated with it. Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException. Closing a previously closed stream has no effect. Throws: IOException - If an I/O error occurs

J Java Docs: Swing

interface KeyListener

---

void keyPressed(KeyEvent e)

Invoked when a key has been pressed.

void keyReleased(KeyEvent e)

Invoked when a key has been released.

void keyTyped(KeyEvent e)

Invoked when a key has been typed.

class KeyAdapter **implements** KeyListener

---

void keyPressed(KeyEvent e)

Invoked when a key has been pressed. Does nothing.

void keyReleased(KeyEvent e)

Invoked when a key has been released. Does nothing.

void keyTyped(KeyEvent e)

Invoked when a key has been typed. Does nothing.