Section Solutions 7

Problem One: Word Walks

```
import acm.program.*;
import acm.util.*;
import java.io.*;
import java.util.*;
public class WordWalk extends ConsoleProgram {
  /* The name of the dictionary file. */
  private static final String WORDS_FILE = "dictionary.txt";

  /* How many letters overlap between words. */
  private static final int NUM_OVERLAPPING_LETTERS = 2;

  public void run() {
    /* Build up a map from two-letter sequences to the words that start with those
     * sequences.
     */
    HashMap<String, ArrayList<String>> dictionary = loadDictionary();

    /* Continuously ask the user for a word, then do a word walk! */
    while (true) {
      String word = readLine("Enter starting word: ").toUpperCase();
      doWordWalk(word, dictionary);
    }
  }

  /**
   * Given a word and a dictionary, does a word walk from the starting word
   * through the dictionary.
   *
   * @param word The word to start from.
   * @param dictionary The dictionary of words.
   */
  private void doWordWalk(String word,
                          HashMap<String, ArrayList<String>> dictionary) {
    while (true) {
      /* Print the current word. */
      println(word);

      /* Look up what the successor words are. */
      String suffix = word.substring(word.length() - NUM_OVERLAPPING_LETTERS);
      ArrayList<String> options = dictionary.get(suffix);

      /* If no words start with this sequence, we're done. */
      if (options == null)
        break;

      /* Otherwise, choose a random successor word. */
      RandomGenerator rgen = RandomGenerator.getInstance();
      word = options.get(rgen.nextInt(0, options.size() - 1));
    }
  }
```

```
  /**
   * Reads in the dictionary as a map from prefixes to words with that prefix.
   *
   * @return The dictionary, sorted by prefixes.
   */
  private HashMap<String, ArrayList<String>> loadDictionary() {
    try {
      BufferedReader br = new BufferedReader(new FileReader(WORDS_FILE));
      HashMap<String, ArrayList<String>> result =
          new HashMap<String, ArrayList<String>>();

      while (true) {
        String line = br.readLine();
        if (line == null) break;

        /* Determine the prefix of the current word. */
        String prefix = line.substring(0, NUM_OVERLAPPING_LETTERS);

        /* Ensure that there is something in the HashMap for this prefix. */
        if (!result.containsKey(prefix))
          result.put(prefix, new ArrayList<String>());

        /* Add the word to the list of strings starting with its prefix. */
        result.get(prefix).add(line);
      }

      br.close();
      return result;
    } catch (IOException e) {
      throw new ErrorException(e);
    }
  }
}
```

Problem Two: The Average Color

```
import acm.program.*;
import acm.util.*;
import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class AverageXKCDColor extends GraphicsProgram {
  /* The number of columns in the text box. */
  private static final int NUM_COLUMNS = 16;

  /* The words file. */
  private static final String COLORS_FILE = "xkcd-colors.txt";

  /* The text field used for data entry. */
  private JTextField colorNameEntry;

  /* The color data, as stored in lecture. */
  private HashMap<String, ArrayList<Color>> colors;

  public void init() {
    colors = readXKCDColors();
    addInteractors();
    addActionListeners();
  }

  /**
   * Adds the interactors to the window.
   */
  private void addInteractors() {
    /* A nice label so that we know what the box does. */
    add(new JLabel("Enter color: "), SOUTH);

    /* The entry field. */
    colorNameEntry = new JTextField(NUM_COLUMNS);
    add(colorNameEntry, SOUTH);

    /* Make sure that we know to listen for the user pressing
     * ENTER while in the text box.
     */
    colorNameEntry.addActionListener(this);
  }
```

```
  /**
   * When the user hits ENTER, look up the current color.
   */
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == colorNameEntry) {
      /* Get the color name. */
      String colorName = colorNameEntry.getText().toLowerCase();

      /* If the color exists, look it up. */
      if (colors.containsKey(colorName)) {
        setBackgroundToAverage(colorName);
      }
    }
  }

  /**
   * Sets the background color of the window to the average color value
   * for the given color name.  It is assumed that the color exists in
   * the database.
   *
   * @param colorName The name of the color.
   */
  private void setBackgroundToAverage(String colorName) {
    /* Find what colors match the given name. */
    ArrayList<Color> matches = colors.get(colorName);

    /* Track the average color value. */
    int red = 0, green = 0, blue = 0;

    /* Iterate across the colors, updating the total color values. */
    for (Color color: matches) {
       red   += color.getRed();
       green += color.getGreen();
       blue  += color.getBlue();
    }

    /* Compute the average red, green, and blue components. */
    int avgRed   = red   / matches.size();
    int avgGreen = green / matches.size();
    int avgBlue  = blue  / matches.size();
    setBackground(new Color(red, green, blue));
  }

  /**
   * Loads in a table of all of the names of the color data points from
   * the xkcd color naming set.
   *
   * @return A map from the color names to the list of RGB triplets with
   *         that name.
   */
  private HashMap<String, ArrayList<Color>> readXKCDColors() {
    /* To save space, this code is omitted.  You can find it in the
     * lecture code from Lecture 19.
     */
  }
}
```