Com S 227  
Fall 2013  
Assignment 2  
300 points

Due Date: Monday, November 4, 11:59 pm
“Late” deadline (25% penalty): Tuesday, November 5, 11:59 pm

General information

This assignment is to be done on your own. If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions; see below.

Your work will not be accepted for credit unless we have on file a signed “Acknowledgment of Academic Dishonesty Policy.” These are normally completed as part of Lab 2; otherwise, you can download and print the form and give it to your instructor or TA: http://www.cs.iastate.edu/~cs227/acknowledgment.pdf.

Please start the assignment as soon as possible and get your questions answered early. Reading a technical specification like this document is not like reading a story from beginning to happy ending. You will probably have to read it all multiple times before things start to click. You can start writing and testing code before you fully understand all the details, though. There is a “Getting Started” section towards the end of this document that might have some useful advice.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder assignment2. If you don’t find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag assignment2. But remember, do not post any source code for the classes that are to be turned in. It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the button labeled “pre” to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post “private” so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form “read all my code and tell me what’s wrong with it” will generally be ignored.
Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled “Official Clarification” are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

**Introduction**

For this assignment you will implement the logic for a simple game in which the player attempts to rearrange a sequence of letters into a known word. There are three classes for you to implement: *ScrambledWordGame*, *ScoreCalculator*, and *WordScrambler*. None of these classes is very complex although there are a couple of interesting loops to develop. In addition, you will create a JUnit test class, *ScrambledWordGameTests*, for the *ScrambledWordGame* component.

The purposes of this assignment are:

- To give you some practice using loops and strings in the context of classes
- To give you a chance to work with a system that has multiple components
- To encourage you to write JUnit tests as you develop your code

You do not need arrays or ArrayLists to do this assignment.

**Note about sharing test ideas**

The JUnit test for *ScrambledWordGame* is to be turned in as part of the assignment. Therefore, *you may not distribute or share any actual test code for this class*. However, you are welcome to share ideas for *test cases*. That is, if you come up with ideas for test cases that you can describe in English such as

“With scrambled word ‘rocan’ after calling moveRight(1, 2), getScrambledWord should return ‘rcaon’”

“After calling start(“abc”, “xyz”), isSolutionPossible() should return false”
you can post such statements on Piazza or otherwise share them with your colleagues. Also, you may share test code for the other two classes (ScoreCalculator and WordScrambler).

Overview

The application consists of six components, three of which are to be implemented by you. The three classes found in the package hw2.gui constitute a graphical user interface for the game. These classes are fully implemented and you should not modify them. Your task is to implement the game logic to be used by the UI.

The game logic is divided into three classes whose usage is summarized below. The detailed specification is given by the Javadoc, which you should read carefully.

The class ScrambledWordGame

The class ScrambledWordGame contains the logic for storing and manipulating a word whose letters can be rearranged. It encapsulates two strings: the “real” word that the player is trying to identify, and a “scrambled” word that is currently being displayed to the player. There are operations for moving a character to the right or left within the scrambled word and for checking whether the current scrambled word matches the real word (i.e., the puzzle is solved). There is also a method for checking whether the puzzle is solvable at all, and a method for generating a “hint” that shows which letters are in the correct positions.

When initially created, a ScrambledWordGame just returns empty strings for the real and scrambled words. A round of the game is started using the method

    void start(String realWord, String scrambledWord)

That is, the caller (in our case the UI) supplies the initial scrambled form of the word. Thereafter, the caller can manipulate the scrambled form using the methods

    void moveLeft(int index, int howFar)

    void moveRight(int index, int howFar)

which move the character at index the indicated distance left or right. The UI will call one of these methods when the characters are moved on the screen by the player. The UI always displays the resulting string returned by the getScrambledWord() method.
The class ScoreCalculator

A ScoreCalculator defines a strategy for keeping score. The player’s score depends on how long it takes to solve the puzzle. Each word starts with an initial maximum score that depends on the length of the word. As the player is trying to solve the puzzle, this score continually decreases until the player either solves the word or runs out of time. In addition, there are penalties for wrong guesses or for getting hints.

The constructor has three parameters to initialize the attributes of a ScoreCalculator:

```java
public ScoreCalculator(int millisPerLetter, int hintPenalty, int incorrectGuessPenalty)
```

The initial maximum score for a word is just the number of letters in the word times the given value `millisPerLetter`. That is, the initial score is the number of milliseconds the player will be allowed to solve the puzzle. The length of the word is provided as an argument to the `start` method that is called by the UI when starting a round. The score is determined very simply: it is the initial maximum, less the number of elapsed milliseconds and less any accumulated penalties for hints or wrong guesses. (However, the score never goes below zero.) The key method of this class is `getScore(int elapsedMillis)`, which returns the score. The usage model is that the UI will keep track of the elapsed time in the game, and will call `getScore` approximately every one-tenth of a second and update the display with the returned value. It is worth noting that the ScoreCalculator does not internally keep track of time; it is the job of the UI to compute the elapsed milliseconds and send that value as an argument to `getScore`. All the `getScore` method really does is answer the question, “what would the current score be if the given number of milliseconds had elapsed?”

When the player presses the “hint” button, the UI will call `deductHintPenalty()` on the ScoreCalculator, and when the player presses the “submit” button and the solution is incorrect, the UI will call `deductIncorrectGuessPenalty()`. These methods should cause the score to decrease by the respective amounts given in the constructor.

The class WordScrambler

The class WordScrambler has two responsibilities. First, it has a method `getWord()` that returns a word chosen at random from the words in a text file. Second, it has a method `scramble(String word)` that will randomly permute the letters in a given word. These methods are invoked by the UI when the player starts a new round of the game.
The **WordScrambler** should encapsulate a single instance of **Random** to be used for both methods. There are two constructors, one of which allows the caller to supply a seed for the random number generator. This should cause the results of the methods to be entirely reproducible, which is useful for testing.

The text file should just contain a single word per line. The sample code includes some examples of word files. If you want more words, it is easy to find dictionaries online. To change the text file or the seed for the random number generator, edit the main method in **UIMain.java**.

**The class ScrambledWordGameTests**

Part of this assignment is a set of JUnit tests for the **ScrambledWordGame** class. Your test code should be put in the class **ScrambledWordGameTests**. The sample code includes a very short skeleton of a JUnit test to get you going. You can review Lab 5 if you don’t remember anything about JUnit.

Your test cases must follow the same format as the ones given in the sample code:

- Each test case focuses on just one operation or sequence of operations
- There is a short message string stating what is being tested and the expected outcome
- There is an assertion of the form `assertEquals(message, expected value, actual value);`

*Note: the message string should accurately document the test case, so there is no additional javadoc required for the method of your JUnit test.*

We very strongly encourage you to develop the tests at the same time as you write the code. It is much easier to get the code right when you can specify what it is supposed to do, and check whether it is doing it!

There is not a hard and fast rule for how many test cases you need, only that you should try to be reasonably complete. A good strategy is to begin by writing simple statements in words describing how the class should behave, e.g.,

> “After calling start(foo, bar), getScrambledWord() should return ‘bar’”

These statements become the message strings in your test cases. Try to think about what might go wrong and test against it. Remember to check “boundary” cases, such as moving the very first or last letter in a word. As an example, you might take a look at the unit tests that we used for the Car class from homework 1, [http://www.cs.iastate.edu/~cs227/labs/lab5/CarUnitTests.java](http://www.cs.iastate.edu/~cs227/labs/lab5/CarUnitTests.java)
You are also encouraged to unit test the other two classes you are writing. However, the class `ScrambledWordGameTests` should contain ONLY your tests for `ScrambledWordGame`. Please put the others in a different class.

The sample code

The sample code includes a complete skeleton of the four classes you are writing. It is distributed as a complete Eclipse project that you can import. It should compile without errors out of the box.

1. Download the zip file. You don’t need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for “Select archive file”.
4. Browse to the zip file you downloaded and click Finish.

If for some reason you have problems with this process, or if the project does not build correctly, you can construct the project manually as follows:
1. Unzip the zip file containing the sample code.
2. In Windows Explorer or Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse
4. In the Package Explorer, navigate to the src folder of the new project.
5. Drag the hw2 folder from Explorer/Finder into the src folder in Eclipse.
6. Add JUnit 4 to the build path.

(If in particular, if you have an older Java installation you may have build issues since the project is configured to use Java 7.)

About the UI

The UI is built on the Java Swing libraries. This code is somewhat complex and specialized, and is somewhat beyond the scope of the course. However, you might be interested in exploring how it works. In particular it is often helpful to look at how the UI is calling the methods of the classes you are writing.

The main method is in `hw3.gui.UIManager`. You can try running it, and you’ll see the initial window, but until you start implementing the game logic you’ll just get an error when you click Start. All that the main class does is to initialize the components and start up the UI machinery. The class `WordGameUI` contains most of the UI code and defines the main panel, which consists of three sub-panels – one for the current score, one for the buttons, and one for the letters that
can be moved around with the mouse. The buttons are standard Swing components, but the movable letters are based on a custom component, the *WordCanvas*, that uses the graphics libraries to draw the rectangles with the letters and allow them to be moved with the mouse.

The interesting part of any graphical UI is in the *callback* methods. These are the methods invoked when an event occurs, such as the user pressing a button. If you want to see what’s going on, start looking at `WordGameUI.java` near the bottom of the file. Look for the classes called `StartButtonHandler`, `SubmitButtonHandler`, etc. (These are “inner classes” of `WordGameUI`, a concept we have not seen yet, but it means they can access the `WordGameUI`’s instance variables.)

If you are interested in learning more, there is a collection of simple and well-commented Swing examples linked on Steve’s website, see [http://www.cs.iastate.edu/~smkautz/](http://www.cs.iastate.edu/~smkautz/) and look under “Other Stuff”. The absolute best comprehensive reference on Swing is the official tutorial, [http://docs.oracle.com/javase/tutorial/uiswing/TOC.html](http://docs.oracle.com/javase/tutorial/uiswing/TOC.html).

### Suggestions for getting started

Download the project and make sure it compiles without errors.

The three game classes you are implementing are completely independent of each other, so you can develop and test them separately.

For `ScrambledWordGame`, develop the unit tests at the same time as you start writing the code. Start with the constructor, and the methods `start`, `getRealWord` and `getScrambledWord`. These are easy. The methods `moveLeft` and `moveRight` can actually be implemented without loops (look into the `substring()` methods in the `String` class). Test these methods and make sure you always get the correct value for `getScrambledWord`. At this point, you can run the UI and the letters should move correctly when you select and drag them. However, in order for the UI to actually start, you’ll need to do two things:

- Temporarily stub in the `isSolutionPossible` method so it always returns true.
- Temporarily stub in the `WordScrambler` methods to return usable test values, for example, have `getWord` return “apple” and have `scramble()` return “palep”.

Since the score calculator is always returning zero from `getScore`, you won’t see the timer counting down at the top of the window. But `ScoreCalculator` is really easy, so you might do
that next and unit test it. Then when you run the UI, you should see the number at the top steadily decreasing to zero.

For the WordScrambler, you’ll need to decide how to read the file and select a word at random. There are a couple of ways to go about this. One easy solution is to record the filename and read the file to count the number of words. Say this number is \( N \). Whenever \texttt{getWord} is called you can generate a random number \( k \) between 0 and \( N \), and read the file again to select the \( k \)th word.

Alternatively, if you are already comfortable with ArrayLists, you could read the file once into an ArrayList and then select a random index in the list whenever \texttt{getWord} is called.

For the \texttt{scramble} method, there are as always many ways to go about it. If you are not sure where to begin, you might start with a helper method of the following form and then think about how it might be useful:

```java
// returns the string obtained when character at index i is removed from s
String removeChar(String s, int i)
```

In the UI, the effect of the “Hint” button is to highlight in cyan the letters that are in the correct position in the scrambled word. This depends on the \texttt{getHint} method of \texttt{ScrambledWordGame}. (The hint disappears the next time a letter is moved.) If the “Submit” method is clicked for a correct solution, then all letters should be highlighted in cyan. This depends on the method \texttt{checkSolution} (which is easy).

Note that the method \texttt{isSolutionPossible()} is invoked once by the UI after it calls \texttt{start()}. The idea is that if (e.g. due to an error in your \texttt{scramble} method) \texttt{start()} is called with a scrambled word that isn’t actually a rearrangement of the letters in the real word, the UI will display a warning and will not continue with the round.

**The specchecker?**

There is no specchecker, since you are being provided with skeletons. Just be careful not to change the public API. Any instance variables or helper methods you add should be private.

**Style and documentation**

Roughly 15% of the points will be for documentation and code style. The general guidelines are the same as in homework 1. However, since the skeleton code is fully documented, there is not that much to do. Remember the following:
• You must add an \texttt{@author} tag with your name to each class javadoc.
• You must javadoc each instance variable and helper method that you add.
• Since the code includes some potentially tricky loops to write, you ARE expected to add internal (//-style) comments, where appropriate, to explain what you are doing inside the longer methods. A good rule of thumb is: if you had to think for a few minutes to figure out how to do something, you should probably include a comment explaining how it works. Internal comments always precede the code they describe and are indented to the same level.
• Keep your formatting clean and consistent.

\textbf{What to turn in}

Please submit, on Blackboard, a zip file that contains the directory \texttt{hw2}, which contains the four required files:

\begin{verbatim}
    hw2/ScoreCalculator.java
    hw2/ScrambledWordGame.java
    hw2/ScrambledWordGameTests.java
    hw2/WordScrambler.java
\end{verbatim}

Normally it is sufficient to right-click on the \texttt{src} directory of your project and select “Send To -> Compressed/zipped file”. \textit{Do not zip up the entire project, just the src directory!} It is entirely acceptable to include additional test classes you have created, and to include the UI code in directory \texttt{hw2/gui}. Use the default Windows or Mac zip utility, \texttt{not} Winrar or 7zip or whatever.

\begin{center}
\begin{tabular}{|p{\textwidth|}}
\hline
Remember to CHECK your submission after you upload it on Blackboard by looking at your submission history. You WILL lose points if your submission is incorrect. \\

\textit{Download the zip file you submitted and check it carefully. Extract the files into a clean, temporary directory and \textbf{look at them} before you submit. Are they .java files? Are all the required files present in the archive? Are they in the directory \texttt{hw2}? Are they the latest, working versions of your code?} \\
\hline
\end{tabular}
\end{center}

If you are not sure how to do any of these things, see the document “Assignment submission HOWTO” which can be found on the Blackboard Assignments page or linked on Piazza.