Com S 227
Spring 2015
Assignment 2
300 points

Due Date: Tuesday, March 31, 11:59 pm (midnight)
"Late" deadline (25% penalty): Wednesday, April 1, 11:59 pm

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy acknowledgement* on the Homework page on Blackboard. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Note: Our second exam is Monday, April 6, which is just six days after the due date for this assignment. ***It will not be possible to have the assignments graded and returned to you prior to the exam****.*

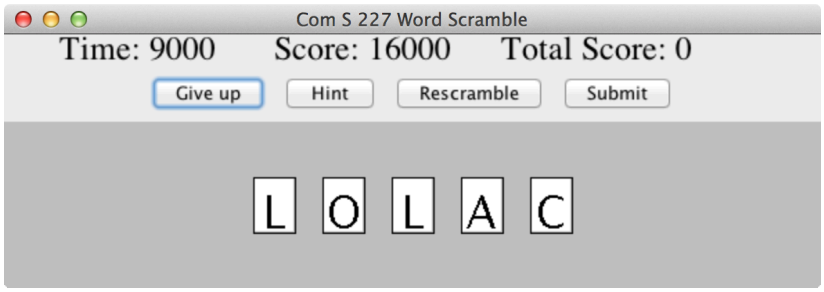
Please start the assignment as soon as possible and get your questions answered right away!

Introduction

The purpose of this assignment is to give you some practice writing loops, using arrays, reading files, and, most importantly, to get some experience putting together a working application involving several interacting Java classes.

There are four classes for you to implement: `WordPair`, `Words`, `WordScrambler`, and `ScoreCalculator`. As always, your primary responsibility is to implement these classes according to the specification and test them carefully.

These four classes can be used, along with some other components, to create a game in which a user attempts to guess a scrambled word by rearranging the letters. In particular, we will provide you with code for a GUI (graphical user interface), based on the Java Swing libraries, that allows a user to rearrange letters by dragging them on the screen.



The sample code includes a documented skeleton for each of the four classes you are to create in the package **`hw2`**. The additional code is in the packages **`gui`** and **`util`**. The **`gui`** package is the code for the GUI, described in more detail in a later section. The **`util`** package includes a text-based user interface **`TextUI`**, also described in more detail below, plus a class **`PermutationGenerator`**.

The class **`PermutationGenerator`** is fully implemented and you should not modify it. It is used to generate random permutations. For our purposes, a *permutation* is an array of *n* integers in which each integer 0 through *n - 1* occurs exactly once, where *n* is the length of the array. For example, [2, 0, 4, 1, 3] is a permutation of length 5. The design of the system is such that the only sources of randomness are a **`Random`** object and/or **`PermutationGenerator`** that are under the control of the client code (e.g., one of the UIs). There should be no hidden random behavior in any of the classes you are implementing.

You should not modify the code in the **`util`** or **`gui`** packages.

Specification

The specification for this asssignment includes

- this pdf,
- any "official" clarifications posted on Piazza, and
- the online javadoc: http://www.cs.iastate.edu/~cs227/homework/hw2/doc/

Overview of the package hw2

The class `WordScrambler`

The class **`WordScrambler`** is a utility class containing two static methods, both called **`scramble`**, for arranging the letters of a string using a given permutation generator. Rearranging the string "CDEFG" according to the permutation [2, 0, 4, 1, 3] means we take the character at index 2, then the character at index 0, then the character at index 4, and so on, to obtain "ECGDF".

The two **`scramble`** methods have the forms:

```
public static String scramble(String word, PermutationGenerator gen)
public static String scramble(String word, int fixedCount, PermutationGenerator gen)
```

The **`scramble`** methods must use the given **`PermutationGenerator`** to produce a permutation of appropriate length, and apply it to the word as described in the example above. In the second form, all characters in with indices less than **`fixedCount`** are left fixed. There are additional examples in the **`WordScrambler`** javadoc.

To write tests for these methods, use the fact that a **`Random`** object constructed with a given seed will always produce the exact same results. For example, you could write

```
Random rand = new Random(137);
PermutationGenerator gen = new PermutationGenerator(rand);
String result = WordScrambler.scramble("ABCD", gen);
System.out.println(result);
System.out.println("Expected BDAC");
```

This works because the first 4-element permutation produced by the generator above is always [1, 3, 0, 2].

The class `Words`

The class **`Words`** represents a list of strings initialized from a file. After initialization, the client can call the method:

```
public String getWord(Random rand)
```

which uses the given **`Random`** to select a word from the list as described in the javadoc. A **`Words`** object is initialized using a filename. Note that both the constructor and the **`getWord`** method are declared to throw **`FileNotFoundException`**. This does not mean that either method *must* throw

FileNotFoundException, it only means that it *may* throw **FileNotFoundException**. Do not remove the **throws** declarations. Whether they are actually necessary depends on how you decide to implement the class. You might just have your constructor record the filename, and then open and read the file in **getWord**. Alternatively, you could open and read the file in the constructor and store an array or **ArrayList** of the words.

Remember that any file you open must be located in the project directory, as reviewed in Lab 7. The sample code includes two word files you can try, **words.txt** (a few words) and **words2.txt** (many words).

To test your code, use the fact that a **Random** object constructed with a given seed will always produce the exact same results. For example, suppose you have a file words.txt containing 5 words:

```
asparagus
bereft
catatonic
morbid
orangutan
```

Then you could write a test such as the following:

```
public static void main(String[] args) throws FileNotFoundException
{
  Words wordList = new Words("words.txt");
  Random rand = new Random(17);
  String aWord = wordList.getWord(rand);
  System.out.println(aWord);
  System.out.println("Expected: bereft");
}
```

since the first call to **rand.nextInt(5)** is guaranteed to return 1.

The class **WordPair**

The core logic is in the class **WordPair**. A **WordPair** is constructed with two strings, a "real" word and a "scrambled" word in which the letters in the real word are permuted (rearranged). The idea is that the player sees the scrambled form of the word only, and the basic operations are to move a character to the left or to the right to reposition it within the word, using the methods **moveLeft()** and **moveRight()**.

As an example, we could write the following test code:

```
public class WordPairTest
{
  public static void main(String[] args)
  {
    WordPair wp = new WordPair("EGGPLANT", "NTAGPEGL");
    System.out.println(wp.getScrambledWord());
    System.out.println("Expected NTAGPEGL");
    wp.moveRight(0, 6);  // move character 0, six spaces to right
    System.out.println(wp.getScrambledWord());
    System.out.println("Expected TAGPEGNL");
    wp.moveLeft(4, 4);  // move character 4, four spaces to left
    System.out.println(wp.getScrambledWord());
    System.out.println("Expected ETAGPGNL");
  }
}
```

Look at the examples carefully and note that the **`moveX`** *operations are not just exchanging two characters; the intervening characters are shifted.*

There are two other mutator methods, **`doLetterHint()`** and **`rescramble()`**. The idea of a *letter hint* is to place a letter in its correct position the scrambled word, starting with the first letter and proceeding to the right for subsequent hints. After being correctly placed by a hint, a letter should be *fixed* in place in the scrambled word and not be allowed to move. The number of letters that are fixed is always given by the method **`getNumLetterHints()`**. It should always be the case that for any index **`i < getNumLetterHints()`**, the scrambled word has the correct letter at index i.

As always, the best way to make this clear is to come up with some test cases, for example:

```
    wp = new WordPair("EGGPLANT", "NTAGPEGL");
    wp.doLetterHint();
    System.out.println(wp.getScrambledWord());
    System.out.println("Expected ETAGPNGL");
    wp.doLetterHint();
    System.out.println(wp.getScrambledWord());
    System.out.println("Expected EGATPNGL");
    System.out.println(wp.getNumLetterHints());
    System.out.println("Expected 2");
```

The **`rescramble()`** method provides another kind of hint by permuting all the non-fixed letters. All characters to the left of **`getNumLetterHints()`** should remain in place. The argument is a **`PermutationGenerator`**:

```
    public void rescramble(PermutationGenerator gen)
```

After rescrambling, the new value of the scrambled word should be the string returned by the call

```
WordScrambler.scramble(getScrambledWord(), getNumLetterHints(), gen)
```

(See the **WordScrambler** javadoc for an example of what this method does.)

The class ScoreCalculator

The class **ScoreCalculator** includes basic configuration information for scoring a scrambled word game. The most important method is

```
public int getPossibleScore(int elapsedMillis)
```

The idea is that the user interface will run a timer to see how long it is taking the player to solve the puzzle. The player's score is dependent on this elapsed time. The **getPossibleScore** method returns the answer to the question: "What would the player's score be, if the puzzle was correctly solved in the given amount of elapsed time?"

A **ScoreCalculator** is constructed with the following parameters:

- **millisPerLetter** - multiplied by length of word to get maximum score
- **hintPenalty** - amount added to player's time for each letter hint
- **rescramblePenalty** - amount added to player's time for each rescramble
- **incorrectGuessPenalty** - amount added to player's time for each incorrect guess

When a round of the game is started, the **ScoreCalculator** is initialized using the **start()** method. The score starts at a maximum value that is based on the length of the word, multiplied by the parameter **millisPerLetter**. Initially there are no penalties. The methods **applyHintPenalty()**, **applyRescramblePenalty()**, and **applyIncorrectGuessPenalty()** record the penalties.

As usual, the best way to figure out what all this means is to write some test cases and think about what should happen:

```
// millisPerLetter = 5000 (5 seconds)
// hint penalty = 2 seconds
// rescramble penalty = 1/10 of a second
// incorrect guess penalty = 1 second
ScoreCalculator sc = new ScoreCalculator(5000, 2000, 100, 1000);

sc.start(10); // for a ten-letter word
System.out.println(sc.getPossibleScore(0));
System.out.println("Expected 50000");
System.out.println(sc.getPossibleScore(1000));
System.out.println("Expected 49000");
```

```
System.out.println(sc.getPossibleScore(20000));
System.out.println("Expected 30000");

sc.applyHintPenalty();
sc.applyHintPenalty();
System.out.println(sc.getPossibleScore(0));
System.out.println("Expected 46000");
```

The text-based UI

The **util** package includes the class **TextUI**, a text-based user interface for a scrambled word game based on the components you are implementing. It has a **main** method and you can run it after you get the required classes implemented. The code is not complex and you should be able to read it without any trouble. It is provided for you to illustrate how the classes you are implementing might be used to create a complete application. Although this user interface is clunky, it has the advantage that it is easy to read and understand how it is calling the methods of your code.

The GUI

There is also a graphical UI in the **gui** package. The GUI is built on the Java Swing libraries. This code is complex and specialized, and is somewhat outside the scope of the course. You are not expected to be able to read and understand it. However, you might be interested in exploring how it works. In particular it is sometimes helpful to look at how the UI is calling the methods of the classes you are writing.

The main method is in **gui.UIMain**. You can try running it, and you'll see the initial window, but until you start implementing the required classes 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 Swing examples linked on Steve's website. See 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 . A large proportion of other Swing tutorials found online are out-of-date and often wrong.

Testing and the SpecChecker

As always, you should try to work incrementally and write tests for your code as you develop it.

Do not rely on the UI code for testing! Trying to test your code using a UI is very slow, unreliable, and generally frustrating. *In particular, when we grade your work we are NOT going to run either of UIs, we are going to verify that each method works according to its specification.*

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specfications, ask questions when things require clarification, and write your own unit tests.

Since the test code is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see the following message in the console output:

```
2 out of 2 tests pass.
```

This SpecChecker will also offer to create a zip file for you that will package up the four required classes. Remember that your instance variables should always be declared `private`, and if you want to add any additional "helper" methods that are not specified, they must be declared `private` as well.

> See the document "SpecChecker HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links" if don't remember how to import and run a SpecChecker.

Importing 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. *However, neither of the UIs will not run correctly until you have implemented the basic functionality of* **`WordPair`**.

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`**, **`gui`**, and **`util`** folders from Explorer/Finder into the **`src`** folder in Eclipse.

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

Getting started

At this point we expect you basically know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification.

Here are a few observations. The four classes that you are writing are completely independent of each other, with the small exception of the **`rescramble()`** method of **`WordPair`**, (which requires a call to **`WordScrambler.scramble)`**. So to a great extent you can work on the four classes separately in any order. Don't try to use the UI for debugging; doing so is slow and frustrating. Make sure each component works correctly on its own before expecting the whole application to work!

Note that the skeleton for the **`getPossibleScore()`** method in **`ScoreCalculator`** returns a score of zero. The skeleton the **`getWord()`** method of **`Words`** always returns the word "AARDVARK". The skeleton for the **`scramble()`** methods of **`WordScrambler`** just return whatever word is given without modifying it. *All of these values are incorrect*, of course, but they can be used temporarily as default values for trying out the rest of the system.

So the most interesting place to start might be with the basic functionality of **`WordPair`**: the constructor and the two accessors **`getRealWord`** and **`getScrambledWord`**. After that you should be able actually start the UI code and see the default word "AARDVARK". (Check that the method **`isSolutionPossible`** is returning the dummy value **`true`**, otherwise the UI will exit immediately.)

You will not be able to move the letters around until you implement **`moveLeft`** and **`moveRight`**, though. It may be tempting to keep playing with the UI, but take a break and make some carefully written unit tests for these methods. There are many ways to implement these methods - you could use substring operations, arrays of **`char`**, **`ArrayList<Character>`**, or **`StringBuilder`**.

A **`StringBuilder`** is like a string whose elements can be modified, and then easily converted into a String. For example, try this out:

```
StringBuilder sb = new StringBuilder("AARDVARK");
sb.insert(4, 'X');
String result = sb.toString();
System.out.println(result);
```

StringBuilder is in the package **`java.lang`** so you don't need an import statement. See the API documentation for details.

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 has the public methods fully documented, there is not quite as much to do. Remember the following:

- You must add an @author tag with your name to the javadoc at the top of each of the four classes you write.
- 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.
- Avoid redundant instance variables
- Accessor methods should not modify instance variables

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, and **for this assignment you are welcome to post and discuss test code**. (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.)

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Homework page on Blackboard, before the submission link will be visible to you.

Please submit, on Blackboard, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw2.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`hw2`**, which in turn contains four files:

```
hw2/ScoreCalculator.java.
hw2/WordPair.java
hw2/Words.java
hw2/WordScrambler.java
```

Please LOOK at the file you upload and make sure it is the right one!

Submit the zip file to Blackboard using the Assignment 2 submission link and **verify** that your submission was successful by checking your submission history page. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under “Syllabus, office hours, useful links.”

We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw2**, which in turn should contain the four files listed above. Make sure you are turning in `.java` files, not the `.class` files. You can accomplish this easily by zipping up the **src** directory of your project. (The zip file will include the other `.java` files in the project, but that is ok.) The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and not a third-party installation of WinRAR, 7-zip, or Winzip.