Boggle

If you are not familiar with the game Boggle, the game is played with 16 dice that have letters on all faces. The dice are randomly deposited into a four-by-four grid so that the players see the 16 letters on the top faces. Each player has a limited amount of time to identify as many words as can be located on the board. Words can be formed using adjacent letters in any direction but must be at least three letters long and cannot reuse letters or wrap around the board. Players accumulate points based on the number of letters in each word found.

In this assignment, you are going to write a program that lets the user play a one-player version of Boggle. Compared to the first two assignments, this one is big, so you should start immediately. The assignment complements the material covered in class, and you should ask questions in class to make sure that we cover everything that you need to know to complete the assignment.

Objectives

In this assignment you will:

1. Gain additional programming experience on a larger program.
2. Become familiar with more of the fundamental classes in the Java Platform, including `StringBuffer`, `Random`, the Java collections classes, and basic file operations.
3. Write a nontrivial recursive method for searching a data structure.
4. Learn to use automated testing to ensure your program's correctness.
5. Decide appropriate data structures to use in a real program.

Program Architecture

The program is divided into three main classes. `BoggleBoard` represents the board itself. It knows how to construct a board, determine whether a given word can be found on the board, and score a word that has been found. `BoggleDict` represents a dictionary of valid words. If the user guesses a word, the word must be in the dictionary and findable on the board. `Boggle` is application itself; it has a reference to a `BoggleDict` and a `BoggleBoard` and provides the user interface to tie them together. `Boggle` is already written for you, although if you finish early you may want to explore the code; I'll happily answer questions about how it works.

The Board

A `BoggleBoard` class is provided, but only the user interface code is written for you. You need to do the following:

1. You should add a second constructor that takes a `String` specifying the initial letters on the board. For example, `new BoggleBoard("thisisbogglegame")` should construct a board with the following layout:

T	H	I	S
I	S	B	O
G	G	L	E
G	A	M	E

By being able to construct a `BoggleBoard` with any letters you want, you will be able to better test your code.

2. Add a `toString` method. This will also be useful for testing. You should probably take a look at `java.lang.StringBuffer` if you are unfamiliar with that class.

3. Write the `shake` method that assigns random letters to each of the positions in `board`. The class already has an instance of `java.util.Random` for you; take a look at that class to see how to generate random numbers. Important: Whenever you assign to a position in `board`, you need to set the text of the corresponding label using

   ```
   labels[i].setText(charToString[newLetter]);
   ```

4. Write a `wordExists` method that takes a `String` (assumed to be a lowercase word) and returns true if the word can be found in the board. For now, it's fine for a letter to be used multiple times in the same word. Think recursively. (You'll probably need a helper function.) For your convenience, I've provided a `getPossibleMovesStartingAt` method that will let you iterate over the adjacent spaces on the board.

5. Write a `static` method `computeScore` that takes a `String` and returns the number of points the word is worth. Points are assigned as follows:

Word length	Points
3 or 4	1
5	2
6	3
7	5
8 or more	11

After you have reached this point, you should test what you have written. Use DrJava's Interactions pane to construct some `BoggleBoard`s and test that the methods you wrote work correctly: the constructors should produce proper boards, the `toString` method should display them correctly, `wordExists` should locate some words and not others, and `computeScore` should return the correct point value for different words.

Testing

You're going to be modifying your `BoggleBoard` implementation to support additional features, so it'd be nice if DrJava could automatically run these tests every time you change your implementation. This would ensure that you don't accidentally break code that works now. Fortunately DrJava can do this: the Test button invokes JUnit, which is a standard unit testing framework for Java. The complete documentation is online at `http://www.junit.org/` if you want to read it later, but I'll walk you through the process of writing tests for our `List` class.

To test `List.java`, write another file `ListTest.java`. (It can be called anything, but the convention is to append `Test` to the class name.)

```
public class ListTest extends junit.framework.TestCase {
```

If you're going to construct some instances that will be used for several different tests, you can declare private variables for them. You don't have to do this; you can simple construct the instances whenever you need them.

```
  private List empty;
  private List singleton;
```

You need a simple public constructor that just calls `super`:

```
  public ListTest(String name) {
    super(name);
  }
```

If you declared private variables, now write optional protected `setUp` and `tearDown` methods to construct whatever state you need for the tests. This can be used for establishing network connections, but here I'm just building `Lists`.

```
  protected void setUp() throws java.io.IOException {
    empty = List.fromString("");
    singleton = List.fromString("3");
  }

  protected void tearDown() {
    empty = null;
    singleton = null;
  }
```

The only reason I have `setUp` throwing an exception is that my `List.fromString` does.

Now you write the tests. Each one is a public method whose name starts with `test`. The method should not return anything. JUnit will use reflection to locate and call these methods.

```
  public void testListLength() {
    assertTrue("empty.length() should be 0", empty.length() == 0);
    assertTrue("singleton.length() should be 1",
               singleton.length() == 1);
  }

  public void testListSum() {
    assertTrue(empty.sum() == 0);
    assertTrue(singleton.sum() == 3);
  }
}
```

`TestCase` (the superclass) inherits lots of `assert` methods: `assertTrue`, `assertNull`, `assertNotNull`, `assertEquals`, `assertSame`. (And there's a basic `fail` method that causes the test to fail.) Each one (including `fail`) has an optional first `String` parameter that is printed if the test fails, though the failure will be apparent even if you omit the message. The `assertEquals` method actually has implementations for all types: `boolean`, `byte`, `char`, `short`, `int`, `long`, `double` and `float` (with a `delta` parameter), and `Object` (which of course uses the `equals` method).

That's it. With `ListTest` the current document in DrJava, press the Test button. It will show you the results of any failed tests.

Now you should write a file to let you test `BoggleBoard`. You should be sure to test all the different classes of input. (When testing `List`, for example, we should be sure to test the empty list, one-element lists, and many-element lists.) If you find a bug in your code as a result of some output, and then you fix the bug, you should make that output into a test case to make sure you don't accidentally reintroduce the bug later.

The Dictionary

In order for the computer to determine whether the user's guess is a valid word, your program needs a dictionary. As of version 1.2, Java finally has an acceptably rich set of data structures for you to use. They all live in the `java.util` package. You should look through the documentation so you know become familiar with it. (Don't be confused by the abstract `Dictionary` class. In data structures terminology, *dictionary* is another word for *map*, and the `Dictionary` class is now deprecated.)

Here is a summary of the important collection interfaces and classes in `java.util`. We will talk about all of these data structures in class, but this table will help you find the corresponding Java implementations.

		Implementations			
		Hash table	Resizable array	Balanced tree	Linked list
Interfaces	`Set`	`HashSet`		`TreeSet`	
	`List`		`ArrayList`		`LinkedList`
	`Map`	`HashMap`		`TreeMap`	

You need to write the `BoggleDict` class. It should behave like a `Set` because the `Boggle` class actually stores an instance in a variable of type `Set`. (Be careful to choose an appropriate implementation!) The class should have a constructor that takes a filename and loads the word list from that file. A word list is provided with the assignment, one word per line. (Unfortunately it may have some words with fewer than three letters; you should ignore them.) You should take a look at the documentation for `java.io.BufferedReader`.

Hint: The implementation of `BoggleDict` is actually surprisingly short.

Running the Game

After you have implemented `BoggleDict`, you can finally play the game. Compile all of your files. From the command line (not DrJava's Interactions pane), type

```
java Boggle
```

to start the game. If you want to build a double-clickable JAR file, type

```
jar cfm Boggle.jar MANIFEST words_ospd.txt *.class
```

Then you should be able to start `Boggle.jar` in the same way that you start DrJava.

Extending the Board

Now it's time to enhance `BoggleBoard`. In the real game of Boggle, "Q" doesn't appear alone on any of the cubes; it is shown as "Qu". You need to modify your

`wordExists` method to treat every occurrence of "Q" in `board` as "Qu". You'll also want to fix your `toString` method. To make the graphical label show "Qu" for "Q", replace the line

```
charToString[i - 'a'] =
new Character(Character.toUpperCase((char)i)).toString();
```

at the top of `BoggleBoard` with

```
charToString[i - 'a'] =
(i == 'q' ? "Qu" :
   new Character(Character.toUpperCase((char)i)).toString());
```

Be sure to run your tests after you've changed `wordExists` so you can be more confident that you didn't break what already worked. You should add some additional tests to be sure that "Qu" is handled properly now.

The other change you need to make is to keep a letter from being used twice in the same word. To do this, you're going to need to pass an additional object with each recursive call. The new object should be a record of which positions have already been used in the current word and should not be used again. Think about what data structure would be most appropriate.

Submission

The assignment is due on Thursday, July 18, at 10:00 AM. You should submit the following files: `BoggleBoard.java`, `BoggleBoardTest.java`, and `BoggleDict.java`. Beginning with this assignment, you are required to start each source file with a valid comment. The comment can describe the contents of the file or simply be the name of the file. Besides being a good programming practice to tell readers what they're looking at, this lets our online submission system check that you're submitting source files rather than something else.