CSC 335 Project 3: Madlibs/ArrayMap

Due:

GitHub Classroom Link:

Project Description

For this project, we will use the Java Collections framework to define a custom collection that ***is*** a **`Map`**. To exemplify its robustness, you will write a word Madlibs-esque word puzzle which relies on the **`Map`** structure to function. This game should first be written using the Java `HashMap` collection, but once your `ArrayMap` has been successfully implemented and tested, you will replace every instance of `HashMap` in your Madlibs game with an instance of your `ArrayMap`

Example Run

```
There are many (1: ADJ) ways to choose a/an (2: NOUN) to read. First,
you could ask for recommendations from your friends and (3: PLURAL
NOUN).

Enter a position to replace: 1
Enter a replacement "ADJ": interesting

There are many (1: interesting) ways to choose a/an (2: NOUN) to read.
First, you could ask for recommendations from your friends and (3:
PLURAL NOUN).

...

There are many (1: interesting) ways to choose a/an (2: book) to read.
First, you could ask for recommendations from your friends and (3:
family).


Puzzle complete! Would you like to play again?
```

Implementation

In regards to the implementation of Madlibs, you are to read in a file of Madlibs templates, which we have provided as **templates.txt** and then randomly select one to begin the game with. In doing this processing, you will find that the templates have parenthetical pairs of indexes mapped to parts of speech. .You should store this in a Map.

You will also need to read a file of words and their associated parts of speech, also provided as **parts_of_speech.txt**. Create a HashMap to store this file as well. If a user tries to replace a template mapping with a word not in the Map, or a word that doesn't have the appropriate part of speech, do something reasonable.

MVC

You are required to create the game portion of the project using the MVC architectural pattern. You must have the following 3 classes:

1. `Madlibs` – This is the main class. When run, the game will begin, similarly to how a new game of Mastermind was played in the previous project.
2. `MadlibsController` – This class contains all of the string replacement logic, all methods **must** be useful to the view, but not actually contain any code that prints to the console.
3. `MadlibsModel` – This class contains all of the data used to represent the Madlibs game. In specific, you will need to store the Map of empty spaces (and their according part of speech) to words.

ArrayMap

In the first version of your Madlibs program, the Map you used was Java's HashMap. However, for this project, you will replace that with a map that you create according to the rules of the java.util Framework.

An ArrayMap will be a map (dictionary) that is implemented using two arrays: one of keys and one of values. Both arrays will be of type Object.

Your ArrayMap will be a templated (generic type) that will allow maps of any type to any other type. It will be declared as:

public class ArrayMap<K, V> extends AbstractMap<K,V>

You are required to provide the implementations of several methods to make your map work.

@Override
public V put(K key, V value)

This method adds key and value to your map. If key already exists, the new value replaces the old one, and the old one is returned.

@Override
public int size()

This method returns the number of mappings that the object contains.

@Override
public Set<Entry<K, V>> entrySet()

Returns a Set of key, value pairs contained in an Entry object. The AbstractMap class provides a concrete SimpleEntry class that we can use to hold them.

You will need to provide a concrete set, which you will do via a private inner class:

private class ArrayMapEntrySet extends AbstractSet<Entry<K,V>>

You will need to implement these methods in ArrayMapEntrySet:

@Override
public int size()

This method returns the size of the set (and of the Map).

@Override
public boolean contains(Object o)

This method should return true if the Set contains an Entry equal to the the one represented by the parameter. If o is not an Entry, this is trivially false. If it is, validate that the key and the value are actually part of the Map.

@Override
public Iterator<Entry<K,V>> iterator()

This returns an iterator that walks over the Set of Entries in the Map. This iterator will also be implemented as a private inner class:

private class ArrayMapEntrySetIterator<T> implements Iterator<T>

And must provide implementations of:

@Override
public boolean hasNext()

Returns true if there are more items in the Set of Entries being iterated over.

@Override
public T next()

Returns an Entry (an AbstractMap.SimpleEntry<V,E> for us) that represents the next mapping in our Map.

@Override
void remove()

The JavaDoc for this method states:

Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to next(). The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Throws:

IllegalStateException - if the next method has not yet been called, or the remove method has already been called after the last call to the next method

Using ArrayMap for Madlibs

Once you've built and tested your ArrayMap class by itself, you will replace HashMap in your Madlibs project with your ArrayMap. Use generic Map references and construct new ArrayMaps. This should require minimal code changes.

Do not assume your ArrayMap will only be tested against the Cryptogram program. Make sure it works for any reasonable application.

This means that you will need to do the ArrayList-style growth on your key and value arrays. You will need to have an initial capacity and when put() is unable to find any space in the arrays, you will need to double their sizes and copy the old elements over.

Make sure you have a test suite that exercises your code to convince yourself that it works independently of the Madlibs program.

Requirements

- The ArrayMap class needs all of the methods and inner classes as explained above
- A reasonable Test Suite that convinces you that your ArrayMap works (we will grade with our own test suite for correctness, but yours should exist and do what we want our test suite to do).
- Your test suite should 100% branch coverage on ArrayMap, ArrayMapEntrySet, and ArrayMapEntrySetIterator
- You may use any of the templated methods provided by AbstractMap or AbstractCollection, and these also will serve as good methods to use in your JUnit tests since they implicitly call the methods you have overridden.
- Documentation using javadoc for each method and class.

Submission

As always, the last pushed commit prior to the due date will be graded.