CS21120 Assignment, 2024-2025

James Finnis (jcf12@aber.ac.uk)

Release date: 30th October 2024

Hand in deadline: 22nd November 2024, 13:00 via Blackboard

Feedback date: 13th December 2024

Introduction

The purpose of this assignment is to test your ability to

- use appropriate built-in Java types (such as maps, lists and sets) to build a data structure to process some data;
- analyse the performance of your structure;
- code to an interface and follow a specification precisely (which is good practice for industry and will help find bugs).

You will also learn a little about how the sounds of speech can be stored and manipulated in a computer. Learning about an application field is often necessary.

Problem Description

The assignment involves **finding rhymes of English words.** You will do this using a very large set of words and their pronunciations – the CMU Pronouncing Dictionary – which will be provided to you as a text file. You will write some classes to represent this data in a convenient way in memory, and then a method to read the text file into objects of those classes. You will then write some code to help you find rhymes for words.

The CMU Pronouncing Dictionary

This is a very large text file of words and their pronunciations in North American English[1]. Here is a small sample:

```
surplus S ER1 P L AH0 S
surpluses S ER1 P L AH0 S IH0 Z
surprise S ER0 P R AY1 Z
surprise(2) S AH0 P R AY1 Z   # a comment
```

- Each line consists of a word and a pronunciation of that word.
- Anything after a hash character (#) is a comment and must be ignored
- Alternate pronunciations are given by putting (n) in brackets immediately after the word (like “surprise” in the example). There is no space between the word and the brackets.
- The pronunciation consists of ARPABET phonemes (see below) separated by spaces.
- Some of the words are very, very obscure!
- More information: https://github.com/cmusphinx/cmudict

[1] It is very difficult to find a similar dictionary for other varieties of English!

You must download your own copy of the dictionary from the link above or from https://users.aber.ac.uk/jcf12/downloads/cmudict.dict

ARPABET

The CMU dictionary uses a modified version of ARPABET, a way of representing **phonemes** as one or two ASCII letters. Each phoneme is one of the fundamental sounds of a language – in this case American English[2]. The following table shows the ARPABET representation of each phoneme, with an example of a word that contains it. The example has the part of the word which contains the phoneme in brackets.

AA	f(a)ther	**UH**	b(oo)k	**N**	(n)ap
AE	b(a)t	**UW**	b(oo)t	**NG**	so(ng)
AH	b(u)t	**B**	(b)at	**P**	(p)at
AO	c(au)ght, st(o)ry	**CH**	(ch)in	**R**	(r)ed
AW	(ou)t	**D**	(d)og	**S**	(s)ee
AY	b(i)te	**DH**	(th)is	**SH**	(sh)oe
EH	b(e)t	**F**	(f)ish	**T**	(t)ap
ER	b(i)rd	**G**	(g)o	**TH**	(th)ink
EY	b(ai)t	**HH**	(h)at	**V**	(v)an
IH	b(i)t	**JH**	(j)am	**W**	(w)et
IY	b(ee)t	**K**	(c)at	**Y**	(y)es
OW	b(oa)t, c(o)ne	**L**	(l)ight	**Z**	(z)oo
OY	b(oy)	**M**	(m)ap	**ZH**	mea(s)ure

Vowels and stress

Vowel sounds are shown shaded in the table above. When these appear in a pronunciation they will **always** be followed by a stress digit: 0, 1 or 2. This indicates where the stress falls in a word.

- 0 means the vowel is unstressed
- 1 means this is a primary stress
- 2 means this is a secondary stress – less loud or high-pitched than a primary stress.

For example, the word "impacted" has two pronunciations:

```
impacted IH1 M P AE2 K T IH0 D             # IM-pact-(ed)
impacted(2) IH2 M P AE1 K T IH0 D          # im-PACT-(ed)
```

In the first pronunciation, "impacted" is the past tense of a verb: "the spacecraft impacted the asteroid." Here the pronunciation would be "IM-pact-(ed)." The stress pattern is primary, then secondary, then none. The second pronunciation is for when "impacted" is used as an adjective – "**impacted** wisdom tooth." Here the stress falls on the second syllable: "im-PACT-(ed)". The "im" has a little stress, "pact" has the primary stress, and "-ed" has no stress.

[2] ARPABET does not contain all the phonemes that exist – just those for representing North American English well enough to generate recognisable speech! If you are interested in all the sounds that humans make, visit this site: https://www.ipachart.com/

The Assignment

In this assignment, you will write code to represent the pronunciations of words and find those pronunciations quickly. You'll also write code to find rhymes – words which partially match the pronunciations of other words.

Overview of Requirements

These are the assignment tasks:

1. Write and test the Phoneme class, which represents a phoneme in a word.
2. Write and test the Pronunciation class, representing how a word can be pronounced.
3. Write and test the Word class, which represents a word and its pronunciations.
4. Write and test the Dictionary class, which stores all the words, and read the words and their pronunciations from the CMU file.
5. Add code to Pronunciation and Dictionary to find rhymes for words.

You will need to write several classes to conform with Java interfaces provided for you. This is because we will be putting your code into an automated test system – if the classes do not have the right interface, they will not fit into the system and you will get very low marks.

- ARPABET phonemes are represented by the Arpabet enum, which we provide for you.
- Phonemes within words are represented by Phoneme objects, which have an Arpabet value and a numerical stress value. Phoneme must implement the IPhoneme interface.
- A particular pronunciation of a word is represented by a Pronunciation object, which is made up of a sequence of Phonemes. Pronunciation must implement the IPronunciation interface.
- Words in the dictionary are represented by Word objects. The Word class can contain multiple Pronunciations, and must implement the IWord interface.
- Finally, the Dictionary object contains a collection of Words and must implement the IDictionary interface.

Your classes must be created in the **uk.ac.aber.cs21120.rhymes.solution** package.

For this assignment, you are permitted to use only classes from the Java Collections Framework, including the packages java.util, java.lang etc. Any resources you use must be acknowledged in the documentation.

You must also produce a written report as detailed in the section on "Writing the Report" and under each task description.

Your assignment will be marked based on the functionality of your code and the contents of your report as detailed in the section on "Marking". How you are required to submit the code and report is explained under the section on "Submission Format".

Provided Code

As well as this document, the Blackboard page for the assignment also a contains Zip archive of Java source files which you will need to add to your project. The archive is called **CS21120_ProvidedCode.zip.** Each folder in the archive needs to be unzipped into a different directory somewhere in your project.

The **interfaces** folder contains

- The **IPhoneme, IPronunciation, IWord** and **IDictionary** interfaces
- The **Arpabet** enum, which describes the different phonemes in the subset of ARPABET used by the CMU dictionary.

The **tests** folder contains JUnit classes for testing the behaviour of each of your classes.

Setting Up

Before starting work, set up a new empty project and create three packages:

- **uk.ac.aber.cs21120.rhymes.solution**
- **uk.ac.aber.cs21120.rhymes.interfaces**
- **uk.ac.aber.cs21120.rhymes.tests**

Copy all the Java files from the **interfaces** folder in **CS21120_ProvidedCode.zip** into your new interfaces package. Do not copy the files from the test archive just yet – you will need to add these after you create the classes that they test, or the project will not compile.

Testing

There will be a degree of automated testing of your classes – **make sure you strictly conform to the specifications below and in the Submission section to ensure the tests work!** To help you, some JUnit test cases are provided in classes in the tests package. Please use these to help verify your code works as described in each task below and consider adding new tests to them as part of your testing strategy.

General tips

- **Try to complete each task before moving onto the next one.**
- **My tests alone may not prove your classes are correct** – you may need to do some extra testing yourself to make sure there aren't any bugs.
- **If IntelliJ suggests making a method "static" to fix an error, do not listen to it!** In almost every case, only your main method should be static.
- **If you use autocorrect or Copilot, be very sure you know exactly what the suggested code is supposed to do and how it works**. Copilot often makes suggestions which are completely wrong in ways that aren't immediately obvious, and it usually only looks at the file you're working on and not the entire project. And, of course, it doesn't have access to the assignment.
- In your code, **use interfaces wherever you can, not classes**. For example, use IWord wherever you can and not Word itself.
- **DO NOT change the interfaces I have provided.** If your empty method does not compile, it may be that you have made a mistake in your method's signature.
- If you are in doubt about how a particular method should work, **read the code that tests that method**. That may give you a clue.
- **Read the interface code too**. The comments will be helpful.

Task 1: Implementing IPhoneme (10% of marks)

The first class you need to write is the Phoneme class, which represents how phonemes appear in a word. An ARPABET phoneme by itself is one of the items given in the table on page 2, but vowel sounds can be spoken with different stresses (see “Vowels and stress” on page 2) so we need a way of storing this too. A Phoneme object has the following properties:

- An Arpabet enum value for the phoneme itself,
- its associated stress as an integer, which should be -1 if the phoneme is not a vowel.

Your implementation of IPhoneme must be called *Phoneme*, and it must be in the package **uk.ac.aber.cs21120.rhymes.solution.** While the interface is fully described in its Java source file, it is also given here:

- **Arpabet getArpabet()** – return the Arpabet enum value for the phoneme
- **int getStress()** – return the stress value, which should be -1 if the phoneme is not a vowel
- **boolean hasSameArpabet(IPhoneme other)** – return true if the object has the same ARPABET value as the other object (the stress is ignored). This should throw an IllegalArgumentException if the value passed in is null.

The Constructor

Java interfaces do not permit constructors to be specified. However, your class **must** have a constructor with the signature **Phoneme(Arpabet phoneme, int stress).** Our tests will not compile without this! The constructor must throw an IllegalArgumentException when:

- the stress is not in the range -1 to 2 inclusive,
- If the stress is not -1 and the phoneme is not a vowel (there is an isVowel() method in the Arpabet enum to help with this),
- If the stress is -1 and the phoneme is a vowel.

Testing

To test the code, copy the PhonemeTests.java file from the **tests** folder in the **CS21120_ProvidedCode.zip** archive into your tests package. This will produce some errors in IntelliJ because you have not set up JUnit. Hover over the first error in JobTests.java, which will be in red. Then select “Add JUnit 5..” from “More Actions” – **do not select JUnit 4**.

To run the tests, click on the green arrow in the margin next to “public class PhonemeTests” near the start of the file and select “Run PhonemeTests.” You can also add breakpoints and use “Debug PhonemeTests” to trace through problems.

In the report

This should be a very short note on how this part of the implementation went and whether you had any difficulties.

Task 2: Implementing IPronunciation (20% of marks)

Once Phoneme is working, you can move on to Pronunciation. A Pronunciation is a list of IPhoneme objects which together describe how a word might be pronounced. For example, we could represent a possible pronunciation of "record" as

R	EH1	K	ER0	D

where each box is an IPhoneme, itself consisting of an Arpabet value and an optional stress value. Your implementation must be called *Pronunciation,* must implement IPronunciation, and must be in the package **uk.ac.aber.cs21120.rhymes.solution.**

Again, the interface's source code describes the methods in full, but we also list them here:

- **void add(IPhoneme p)** – add a phoneme to the end of the list – this should throw an IllegalArgumentException if the value passed in is null.
- **List<IPhoneme> getPhonemes()** – return the list of phonemes that make up the pronunciation.
- **int findFinalStressedVowelIndex()** – more details of this method are below.
- **boolean rhymesWith(IPronunciation other)** – you will write this method in full later. For now, just write a "stub" version with returns false.

This interface does not need a constructor in its implementations, but the list of phonemes should start out empty.

What is a rhyme? Finding the last stressed vowel

You must write a method called **findFinalStressedVowelIndex** to complete this task. This will help us tell whether two pronunciations rhyme. For this assignment, the kind of rhyme we are going to use is "perfect" or "full" rhyme: the final stressed vowel and all the following phonemes are the same (although the stresses can differ).

Here are some examples – I have underlined the parts that should be the same.

- **achieve** (AH0 CH IY1 V) **believe** (B IH0 L IY1 V) and **reprieve** (R IY0 P R IY1 V) all end in the sequence **IY1 V**. In all these words, the final stressed vowel is IY, and the following sounds are all the same (just V).
- **achieve** (AH0 CH IY1 V) (and the other words above) and **shirtsleeve** (SH ER1 T S L IY2 V) rhyme, because they all end in **IY(something) V** – the stress can be different.
- **cow** (K AW1) and **allow** (AH0 L AW1) both end with **AW1**.
- **his** (HH IH0 Z) and **hitches** (HH IH1 CH IH0 Z) do NOT rhyme, because one ends with IH0 Z, and the other ends with IH1 CH IH0 Z – the stressed vowel in "hitches" is the first vowel, IH1.

We need to be able to find the last stressed vowel in both pronunciations, so we can make sure the phonemes – and all phonemes after that point – are the same in both. That means that as well as finding what the last vowel is, we need to find where it is – in other words, its index in the list of phonemes that make up the pronunciation.

For example, the pronunciation of **intriguing** is IH2 N T R IY1 G IH0 NG. The last stressed vowel is IY1, whose index is 4 (it is the fifth phoneme in the list).

This means that findFinalStressedVowelIndex must:

- return the index of the last vowel whose stress is 1 ("primary" stress), if it exists
- or the index of the last vowel whose stress is 2 ("secondary") if there is no primary
- or the index of the last vowel if there are no stressed vowels at all
- or -1 if there are no vowels at all (there are a few of these in the dictionary)

Testing

To test the code, copy the PronunciationTests.java file from the **tests** folder in the code archive into your tests package and run it as you did the tests for the previous tasks.

In the report

In the report you should describe how the task went and any difficulties you had – particularly in finding the last stressed vowel. You should also give the **average case time complexity** for the add() and findFinalStressedVowelIndex() methods.

Task 3: Implementing IWord (10% of marks)

For this task you need write the class *Word*, which implements IWord. This represents an English word and the different ways it could be pronounced. This class must also be inside the **uk.ac.aber.cs21120.rhymes.solution** package.

To correctly implement the interface, the class must have the following methods:

- **String getWord()** – get the ordinary English spelling of the word;
- **addPronunciation(IPronunciation p)** – add a pronunciation to the word – this should throw an IllegalArgumentException if the pronunciation is null;
- **Set<IPronunciation> getPronunciations()** – return the possible pronunciations of the word.

This class **must** also have a constructor with the signature

- **Word(String word)**

which creates the new word with no pronunciations and sets its standard English spelling.

Testing

To test the code, copy the WordTests.java file from the **tests** folder in the code archive into your tests package and run it as you did the tests for the previous tasks.

In the report

The report should briefly talk about how the task went, but also give the **average time complexity** for the addPronunciation method.

Task 4: Implementing IDictionary and parsing the CMU file (25%)

The last class you need to write is Dictionary, which implements the IDictionary interface. This class handles a collection of words and their various pronunciations. It also reads in and processes the CMU Pronouncing Dictionary and converts it into phonemes, pronunciations and words. To correctly implement the interface, the class must have the following methods:

- **IWord getWord(String s)** – return the word whose English spelling is *s*, or null if the word is not found in the dictionary.
- **void addWord(IWord w)** – add a word to the dictionary. This must throw IllegalArgumentException if the word is already in the dictionary.
- **int getWordCount()** – return the number of words in the dictionary.
- **int getPronunciationCount()** – return the number of pronunciations in the dictionary.
- **void parseDictionaryLine(String line)** – parse a line of text in the format used by the CMU Pronouncing Dictionary, creating a new pronunciation and if necessary a new word.
- **void loadDictionary(String filename)** – load the entire CMU Pronouncing Dictionary from a file.
- **void getRhymes(String word)** – find all the rhymes for all pronunciations of a word (this is for task 5, it should just return null for this task).

It is a good idea to do this task in the following way:

1. Write getWord, addWord, getWordCount and getPronunciationCount, adding the remaining three methods but leaving them as empty "stubs."
2. Run the tests as described below, and get all the tests working that don't need those three methods.
3. Write parseDictionaryLine, test it and get it working.
4. Write loadDictionary, test it and get it working. You will write getRhymes in the final task.

Some string manipulation tips

You have already seen some examples of the CMU dictionary format on page 1. The parseDictionaryLine method requires you to convert lines of text in this format into Phoneme, Pronunciation and Word objects, which will involve manipulating strings. Here are some useful things to know, but you probably won't need all of them:

- You will need to split the line – firstly to remove comments, then again to separate into word and pronunciation, then again to split the pronunciation into phonemes.
- To find out whether a string has a particular character you can use String's **contains** method.
- To find out the position of a character within a string you can use the **indexOf** method.
- To get part of a string using an index and a length you can use **substring**.
- Use the **split()** method to split a string using a delimiter. You may find the "limit" parameter useful.
- You will need to turn strings into Arpabet enum values to create Phonemes. You can use the **Arpabet.valueOf(String s)** static method to do this
- To get the character at a particular position in a string, use String's **charAt** method.
- To see if a character is a digit, use **Character.isDigit(c)**.
- To convert a character to a number, use **Character.getNumericValue(c)**.

Testing

To test the code, copy the DictionaryTests.java file from the **tests** folder in the code archive into your tests package and run it as you did the tests for the previous tasks. As mentioned before, it's a good idea to test the simpler parts of the code first, though the later tests will fail because some methods have not been completed.

To pass the final tests (which test loadDictionary) you will need to tell the test code where the CMU dictionary file is. At the start of the DictionaryTests class is a field called **CMU_DICT**. You should change the value of this field to hold the location of the **cmudict.dict** file which you will have downloaded earlier (see page 2). **This is the only time you should change the code I have provided**.

In the report

Once again, you should write about what went well and what did not. What Java data structures did you use and why? You should also give the **average and worst case complexities** for the following operations:

- Storing a new word in the dictionary
- Getting a word from the dictionary
- Getting the total number of pronunciations
- Loading the entire dictionary

Task 5: Rhyming! (25%)

The final task involves writing two methods to finish earlier classes.

Pronunciation.rhymesWith

First write Pronunciation's rhymesWith method. This takes another IPronunciation as its argument, and returns true if the two pronunciations have the same phonemes from their last stressed vowel onwards (you should use findFinalStressedVowelIndex for this).

Note that stress must be ignored when the phonemes are compared: for example, "cat" and "non-fat" rhyme, even though they are pronounced as "K AE0 T" and "N AA0 N F AE1 T" (rhyming parts shown underlined).

Dictionary.getRhymes()

Then you must write the getRhymes method in Dictionary. This takes a string and returns a set of strings. It finds all the words which rhyme with the given word. If a word has multiple pronunciations, they must all be checked. The returned set should contain the word itself, so the method should always return at least one word.

Testing

To test the code, copy the RhymeTests.java file from the **tests** folder in the code archive into your tests package and run it as you did in previous tasks. In your own testing you may disagree with some of the rhymes your code produces – the CMU dictionary has some questionable data! Check the pronunciations in the file when in doubt.

The tests themselves are split into two – the first block, whose names start with "testRhymesWith", test the rhymesWith method in Pronunciation. These can be run without the Dictionary code working, provided you have written stubs. The second block, which start with "testGetRhymes", test the getRhymes method of Dictionary and need a fully working Dictionary class.

In the report

As before, write about what went well and what did not go well. You should also give the **average case time complexity** for the getRhymes method.

Finally, describe how you would calculate which words in English have the most rhymes. What data structures could you use? What would the average worst case time complexity be?

Writing the report

Your report must be around 1000-1500 words, containing the following sections:

- Task 1: Implementing IPhoneme
- Task 2: Implementing IPronunciation
- Task 3: Implementing IWord
- Task 4: Implementing IDictionary
- Task 5: Rhyming
- Self-evaluation: a paragraph describing which parts you found difficult and why, which parts you think you did well, and what mark you expect to get.

Before writing the first two sections remember to read the "In Your Report" part of each task. **The report must be provided as a PDF document.**

Submission format

The submission must be a ZIP file called **cs21120_<userid>.zip** with <userid> replaced with your Aberystwyth user ID (for example cs21120_jcf12.zip). It must contain:

- A directory called **solution** containing the files from your **solution** package
- A directory called **test** containing any additional test classes you have written
- **A PDF file** containing your report.

There will be an element of automated marking – I will copy your files into my own project and run them against my own tests. It is therefore important that you follow the submission requirements precisely, particularly the correct naming of classes and packages.

Marking

Marks are based on the functionality of your code and the contents of your report, as follows:

• Task 1 – Phoneme	5% for the code	5% for the report
• Task 2 – Pronunciation	10% for the code	10% for the report
• Task 3 – Word	5% for the code	5% for the report
• Task 4 – Dictionary	15% for the code	10% for the report
• Task 5 – Rhyming	10% for the code	15% for the report

Self-Evaluation and Formal Aspects – 10%. This includes the submission format (PDF, ZIP) as well as readability and format of your report, referencing and the quality of your code including comments. All figures, tables and diagrams must include captions, and be numbered and cited where appropriate in the main text.

Academic conduct

As with all such assignments, the work must be your own. Do not share code with your colleagues and ensure that your own code is securely stored and not accessible by your classmates. **Any sources you use should be properly credited with a citation, and any help you get should be acknowledged.** Your report must accurately reflect what you have achieved with your code, any discrepancies between your code and report could be treated as academic fraud. Finding undue similarities between your work and others' could be treated as unacceptable academic practice.

Do not be tempted to use AI to complete the code or the report – the code must be written to conform precisely to the specifications of the interface, and it is very unlikely that it will be able to do this. We will be checking for AI-generated code and text with automated tools and by checking for tell-tale signs.

Your attention is drawn to the University regulation on unacceptable academic practice at https://www.aber.ac.uk/en/academic-registry/handbook/regulations/uap/

Support

If you have any problems with understanding or completing the assignment please ask for help, either by email (jcf12@aber.ac.uk) or via the Blackboard forum for the programming assignment.

If there are any circumstances that affect your work, please visit

https://www.aber.ac.uk/en/academic-registry/handbook/taught-schemes/name-193260-en.html

to see what to do.

You should also contact your year tutor (Angharad Shaw, ais@aber.ac.uk, for second year Computer Science students.)