Project 4: Programming Language Keywords

1 Introduction

How do keywords differ from one programming language to another? How many keywords do various languages have? This project investigates those topics, applying the current class material along the way.

This is also a larger project than most; it is therefore due in three distinct waypoints. It will also require extra care on method signatures, so interfaces are provided and must be used.

2 Objective

Now that we have created and used basic objects, we are moving forward to deeper object interactions. We are adding in two new concepts as well: reading data from a file and storing objects in an array.

This program has no user interaction. But it will certainly have test methods that ensure all objects are behaving as intended, plus a Main class to demonstrate the code's capability.

Once you have completed this project, you'll have a solid idea of these concepts and how to code them:

- Arrays (1D) of objects
- File input using Scanner
- Min/Max algorithms
- Insertion sort

3 Data Files

3.1 Language Files (e.g., languages.txt)

This file is provided for you; do not alter this file. It contains *sample* programming language data, one language per line. Your code should work with other similar files.

The first line contains a count of languages. The second line is a header that lets you know the contents of each column; your code can ignore this line. The third and subsequent lines represent languages. Here is an example of what the first several lines of the file might look like:

```
2
Language,DataFile,TypicalType
ALGOL,algol.txt,Compiled
C,c.txt,Compiled
```

Here, two languages are represented. The language ALGOL's keywords, if present[1], will be in a file called algol.txt. Both languages are typically implemented as compiled languages.

3.2 Keyword Files (e.g., java.txt)

For languages where keywords have been researched and recorded, a keyword file will exist. These files have a count of keywords at the top. Subsequent lines contain one keyword per line. Keywords in the files are sorted by length and then by keyword; we'll sort alphabetically once we have read and stored the keywords.

[1] Consider this a work in progress; as you learn about more languages, you'll add to the languages.txt file and add keywords to the specified keyword files.

Here's a sample of the first few lines of java.txt:

```
50
do
if
for
...
```

4 Classes and Objects

Create these objects, listed in order of increasing complexity (and order of recommended coding).

4.1 Language

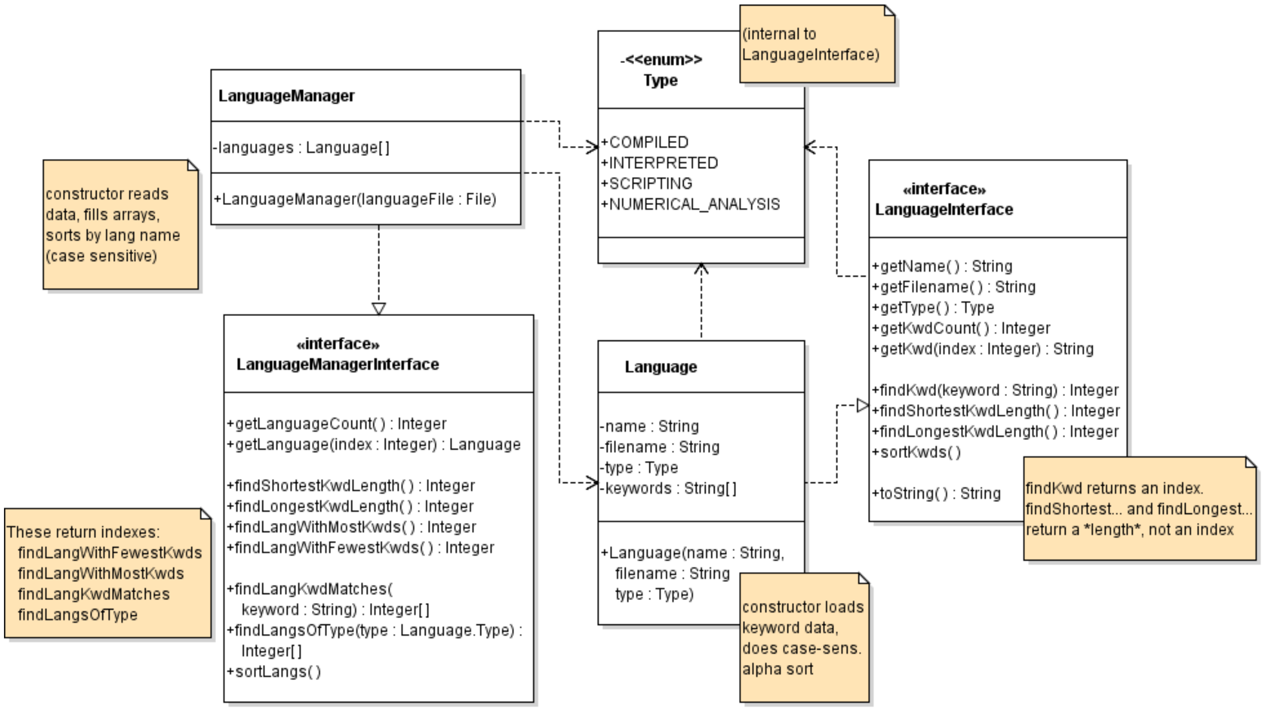
You will create one Language object per programming language in languages.txt. Implement the provided interface LanguageInterface in this class to ensure your signatures are exactly as expected.

4.2 LanguageManager

This class provides a front end for client interaction, providing some useful methods for retrieving interesting information from languages and keywords. Implement the provided interfaces LanguageInterface and LanguageManagerInterface to ensure your signatures are exactly as expected.

5 Project Design

Here is the UML Class Diagram for the project. Interfaces are provided; you'll create the other classes.


LanguageManager
-languages : Language[]
+LanguageManager(languageFile : File)
constructor reads data, fills arrays, sorts by lang name (case sensitive)
«interface» LanguageManagerInterface
+getLanguageCount() : Integer
+getLanguage(index : Integer) : Language
+findShortestKwdLength() : Integer
+findLongestKwdLength() : Integer
+findLangWithMostKwds() : Integer
+findLangWithFewestKwds() : Integer
+findLangKwdMatches(keyword : String) : Integer[]
+findLangsOfType(type : Language.Type) : Integer[]
+sortLangs()
These return indexes: findLangWithFewestKwds findLangWithMostKwds findLangKwdMatches findLangsOfType
-<<enum>> Type
+COMPILED
+INTERPRETED
+SCRIPTING
+NUMERICAL_ANALYSIS
(internal to LanguageInterface)
Language
-name : String
-filename : String
-type : Type
-keywords : String[]
+Language(name : String, filename : String type : Type)
constructor loads keyword data, does case-sens. alpha sort
«interface» LanguageInterface
+getName() : String
+getFilename() : String
+getType() : Type
+getKwdCount() : Integer
+getKwd(index : Integer) : String
+findKwd(keyword : String) : Integer
+findShortestKwdLength() : Integer
+findLongestKwdLength() : Integer
+sortKwds()
+toString() : String
findKwd returns an index. findShortest... and findLongest... return a *length*, not an index

6 Class Details

6.1 Language

The **constructor** should…

- …have preconditions to check for null/empty name and filename, and for null language type.
- …read the data from the associated keyword file, create the keywords array, and fill it in.
- …not throw a FileNotFoundException; instead, let's learn how to handle the exception. Here, we'll simply end up with no keywords if we have trouble finding or reading the file.
- …call for the keywords to be sorted (alphabetically).

- **toString** should have this information, nicely labeled, on a single line: name, type, and keyword count.
- **sortKeywords** should implement an Insertion Sort. Start with the code given in class, on slides. Modify it to sort the keywords alphabetically (the way we'd almost always sort strings).
- **Other methods** likely make sense if you read the class diagram carefully; look at the parameters and returns and see if you can put together the requirements. If you have questions, let's discuss in class.
- Write **preconditions** where they make sense; make this a habit. But remember not to write preconditions that duplicate work Java already does.

6.2 LanguageManager

- The **constructor** should open the specified file, create the languages array, and fill it in with data.
  - It is expected that it will throw FileNotFoundException if it has trouble finding or reading the file. Do you see why this makes sense, and why it's different than the scenario in Language?
  - The constructor should call for languages to be sorted alphabetically, by name. Again, implement an adapted Insertion Sort in the **sortLangs** method.
  - Before any other code, write a precondition test for a null File reference passed as an argument.
- Pay attention to what reasonable **returns** mean in other methods. When *lengths* are requested, the integer returned will indicate the *length*. When *languages* are being requested, integers (or arrays of integers) indicate *positions/indexes* that meet the filtering requested.
- As above, write **preconditions** where they make sense.

6.2.1 Other Notes

- Create proper signals for not-found conditions. When a valid index can't be returned, return -1 as a typical/conventional signal. This also works well with lengths, as -1 is an invalid length.
- Write good loop conditions that make the loop exit condition clear. Do not use keywords `break` or `continue`; there is a (rare) place for those in the real world, but even *there* you should avoid them when simple, straightforward logic can be used.

6.3 Main

Write a runnable main method. In it, demonstrate the capabilities of the code and display the results. You do not need to demonstrate everything, just some high-level stuff you are proud of.

7 Code Implementation

Use *exactly* the method names shown, or instructor test code will fail, and work will be returned. Look carefully at the UML parameter and return data types; they give you clues.

8 Style

- Follow our Course Style Guide found in Canvas.
- Write JavaDoc for Language, covering constructors, methods, and preconditions.

9 Testing

- Write *JUnit5* tests for the Language class; test constructors, sets, gets, and preconditions.
- While you are not required to write tests for LanguageManager, such tests could be advantageous; there are many scenarios you might miss if you do not think like a tester. With arrays and other lists, our algorithms often contain off-by-one and other logic errors; these can only be found through careful testing.

10 Submitting Your Work

- In the past, projects of this size have been a problem for students who wait until the last minute to start the project. This project is therefore due in three waypoints.
- You will be creating multiple java classes (.java) files. BlueJ can create .jar files; be sure and specify "include source" when you use this method. Or compress your files and submit a .zip file instead.
- If your test method relies on any test data files, be sure and include them in your submission.

11 Extra Credit

Write unit tests for the LanguageManager class. Again here, test the constructor, methods, preconditions, etc. One good way to think of this is that your tests should hit every line of code in your source file. If you are using a fancier IDE than BlueJ, there's a way to check this coverage; ask if you are interested.

12 Hints

- Building classes from the ground up, e.g., start with the Language class coding and testing, then move on to LanguageManager. Incrementally develop your solution, building small additions on the solid foundation you have already laid.
- Do not duplicate code; avoid this wherever possible.
- Some algorithms may require some thought before coding. I highly recommend doing work on paper or a whiteboard to come with an algorithm, then writing pseudocode, then finally code.
- The language file is not a tokenized file; it has no traditional token separators. You will need to read the whole record (one line of language data), then break it up into fields (columns) using the String split instance method, which returns an array of strings. This is more good practice on arrays.

(continues…)

13 Grading Matrix

Area	Points
Waypoint A	0%
Language	
General Coding	5%
Constructor	10%
Preconditions	5%
Get methods	5%
Find methods	5%
Waypoint B	
LanguageManager	0%
General Coding	5%
Constructor	10%
Get methods	5%
Find methods	10%
Waypoint C	
Sorting	0%
Keyword sorting	5%
Language sorting	10%
Main class	5%
Testing	10%
Style and Documentation	10%
Extra credit	3%
Total	**103%**