Assignment Report for Assignment 02

|  |  |  |
| --- | --- | --- |
| Course and Section | CSC 340.01 |  |
| Assignment Name | Assignment 02 |
| Due Date and Time | 06-23-2023 @ 11:59 PM |
|  |  |
| First Name and Last Name | Nhan Nguyen |
| SFSU Email Account | nnguyen14@sfsu.edu |
| First Name and Last Name of Teammate | N/A |
| SFSU Email Account of Teammate | N/A |

**PART A**

**Question Description and Analysis**:

This part of the assignment asks me to choose 5 guidelines in [*Y. Daniel Liang’s 8 Class Design Guidelines*](http://csc340.ducta.net/WEEK-01/JAVAtoCPP-ClassDesignGuidelines.pdf) and discuss them in depth, using at least one page and maybe some codes for each guideline to demonstrate my points.

**Answer**:

1. **Cohesion**

In Java programming, maintaining cohesion is crucial for writing clean, maintainable, and efficient code. Cohesion refers to the extent to which the elements of a class belong together and work towards a single purpose. By adhering to the guideline of ensuring cohesion, developers can create classes that accurately represent individual entities and separate responsibilities effectively. A key aspect of cohesion is ensuring that a class represents a single entity. This means that all the properties, methods, and operations within the class should be directly related to the purpose of that entity. By doing so, we create classes that are focused and encapsulate the necessary functionality. Let's consider an example where we need to model a Car class:

public class Car {

private String make;

private String model;

private int year;

// Constructor, getters, and setters

public void startEngine() {

// Code to start the car's engine

}

public void accelerate() {

// Code to accelerate the car

}

// Other car-related methods

}

In this example, all the properties (make, model, year) and methods (startEngine, accelerate) directly relate to the car entity. This ensures that the class represents a single entity, making it easier for developers to understand its purpose and functionality. By grouping all the relevant properties and methods together, the code becomes more readable, reducing cognitive load and making it simpler to reason about. This leads to enhanced comprehension and facilitates collaboration among team members. Moreover, when a class becomes responsible for too many tasks, it can become complex and difficult to manage. To address this, responsibilities can be separated by creating multiple classes, each focusing on a specific aspect or behavior. This modular design allows developers to isolate and understand individual components independently, simplifying code comprehension and reducing dependencies. Modularity also facilitates code reuse, as classes designed with clear responsibilities can be easily incorporated into other projects without carrying unnecessary baggage.

1. **Consistency**

Consistency in naming is crucial for creating code that is easy to understand and navigate. Developers should choose informative and meaningful names for classes, data fields, and methods. It is essential to avoid using different names for similar operations or variables that serve the same purpose. Consistency fosters clarity and enables developers to quickly identify the purpose and functionality of various components within the codebase.

In Java programming, a common convention is to place data field declarations before constructors and constructors before methods. This practice enhances code readability by providing a logical flow to the code structure. By declaring data fields at the beginning, developers can easily identify the properties associated with a class. Placing constructors next allows for clarity in defining how instances of the class are initialized. Finally, positioning methods after constructors aids in locating and understanding the behavior and functionality provided by the class.

Moreover, it is generally recommended to provide a public no-arg constructor for constructing a default instance of a class. This practice allows users to create objects without specifying any arguments, making the class more accessible and easier to use. If a class does not support a no-arg constructor, it is essential to document the reason for its absence.

In certain scenarios, preventing users from creating objects for a class may be desired. This can be achieved by declaring a private constructor in the class. The Math class in Java is a good example of this approach. By making the constructor private, it is explicitly communicated that the class is not meant to be instantiated, as it provides only static utility methods. Private constructors are a useful tool to enforce design patterns, limit object creation, or ensure that the class is used solely for static operations.

1. **Encapsulation**

Encapsulation is a fundamental principle in object-oriented programming that promotes data hiding and information hiding. In Java programming, encapsulation involves using the private modifier to hide class data from direct access by clients. Encapsulation ensures that the internal state and implementation details of a class are hidden from external clients. By declaring class data as private, we restrict direct access and modification from outside the class, preventing unintended modifications or invalid states. This preserves the integrity of the class, minimizing the risk of unwanted side effects and improving code stability.

Moreover, encapsulation allows us to control how data is accessed and modified by providing controlled access points. This is achieved through the use of getter and setter methods. Getter and setter methods play a crucial role in encapsulation by providing an interface for accessing and modifying private data fields. They act as intermediaries between the internal state of a class and external clients, allowing controlled and consistent interactions. Getter methods expose the values of private data fields, enabling clients to retrieve information while keeping the underlying data protected. Setter methods, when used appropriately, facilitate the controlled modification of data fields, providing a means to update the internal state of the class while enforcing any necessary validation or logic. By selectively providing getter and setter methods, developers can enforce read-only or write-only access to data fields as needed. This allows for fine-grained control over the accessibility and mutability of class data, ensuring data consistency and protecting sensitive information.

1. **Completeness**

Completeness in class design is a critical aspect emphasized by Y. Daniel Liang's guidelines, as it ensures that all necessary components are included to create well-structured and effective Java classes. Let's explore the concept of completeness further through example.

Consider a class named "Person" that represents individuals. To achieve completeness, this class should include appropriate data fields, constructors, and methods. For instance, the data fields could include attributes such as "name," "age," and "address." By encapsulating these fields and making them private, we ensure that they are not directly accessible from outside the class. Getter and setter methods can then be provided to enable controlled access to these fields.

public class Person {

private String name;

private int age;

private String address;

public Person(String name, int age, String address) {

this.name = name;

this.age = age;

this.address = address;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

}

In the example above, the Person class exhibits completeness by including relevant data fields and corresponding getter and setter methods. The constructors allow for different initialization scenarios, such as providing a name, age, and address, enabling flexibility when creating Person objects. In conclusion, completeness in class design, as exemplified by the Person classes, ensures the inclusion of all necessary components in a well-structured manner. By incorporating appropriate data fields, constructors, methods, and considering inheritance and interface implementation, developers can create classes that are readable, reusable, and adaptable. Adhering to these principles fosters code maintainability and facilitates efficient software development.

1. **Interfaces vs. Abstract Classes**

Interfaces provide a powerful mechanism for defining contracts and establishing common behavior across unrelated classes. They consist of abstract methods, which are methods without implementations, and constant variables. By implementing an interface, a class guarantees that it will provide concrete implementations for all the methods defined in the interface. Interfaces support multiple inheritance, allowing a class to implement multiple interfaces simultaneously. This enables greater flexibility and the ability to leverage polymorphism, as objects can be treated as instances of their interface types. Interfaces are commonly used to create abstractions, define standardized contracts, and promote loose coupling between components.

On the other hand, abstract classes serve as partial implementations for subclasses to inherit and extend. They are classes that cannot be instantiated on their own and can contain both abstract and concrete methods. Abstract classes allow developers to define common attributes and behaviors shared by a group of related classes. By providing default implementations for certain methods, abstract classes offer a foundation upon which subclasses can build and specialize. Abstract classes promote code reuse and provide a natural way to organize and structure related classes within an inheritance hierarchy. They are particularly useful when there is a need for shared functionality among a group of classes but the exact implementation may vary.

In summary, interfaces and abstract classes are essential tools in Java programming, offering distinct advantages based on their respective features. Interfaces enable the creation of contracts and support polymorphism, while abstract classes facilitate code reuse and provide a foundation for related classes. By understanding the characteristics and use cases of interfaces and abstract classes, developers can design more flexible, modular, and maintainable software systems.

**PART B**

**Question Description and Analysis**:

This part of the assignment asks me to develop an interactive dictionary with requirements from clients. It also asks me to analysis, design the program and finally, implements using Java with good use of data structures.

**Answer**:

1. **Program Analysis to Program Design**

The Interactive Dictionary application is designed to provide a convenient and efficient way to search for words in a dictionary. To analize, I will examine the problem being solved, the approach to storing data using enum objects, and the data structures employed for the dictionary.

Firstly, the problem being solved by the Interactive Dictionary application is the need for quick and accurate word search functionality. Users may have different requirements when searching for words, such as finding specific parts of speech or filtering the results for distinct or reversed entries. The output shows the program's ability to handle different search queries and provide relevant results. For example, searching for "Arrow" returns a definition for the noun form of the word. Similarly, searching for "Book" returns multiple definitions, including noun and verb forms. The sample output also demonstrates the program’s ability to handle optional parameters. For instance, searching for "Book noun" narrows down the results to only noun definitions of "Book." Adding the "distinct" option doesn't change the results, as there were no duplicate definitions. Furthermore, the program handles the "reverse" option, which appears to reverse the order of the definitions. For example, searching for "reverse noun" displays noun definitions in reverse order. In some instances, the program handles invalid or disregarded parameters gracefully. It provides appropriate feedback when invalid or unnecessary parameters are entered, guiding the user to correct usage. Overall, the sample output showcases a functional dictionary program which aims to address these needs and provide an intuitive interface for word lookup. To tackle this problem, the application provides a search functionality that allows users to input a search key and select various options, such as noun, verb, adjective, adverb,…then distinct, or reverse. By incorporating these options, the application enhances the search experience and provides more flexibility to users.

After brainstorming and considering different ways to go for this application, I think that the application need to utilize enum objects to store data related to dictionary entries. Enum objects are a suitable choice because they allow for defining a fixed set of constant values, which aligns well with the various parts of speech that words can belong to (noun, verb, adjective, adverb,…). Enum objects are also useful for defining additional attributes or behavior associated with each constant value.

In the Interactive Dictionary application, the EnumDictionaryData enum class contains all the words and their definition for the data entry emplyees to fill in. Each constant will have keyword, part of speech, and definition. The EnumPartOfSpeech enum class is used to represent the different parts of speech. Each constant in the enum represents a specific part of speech, such as noun, verb, adjective, and adverb. By utilizing enum objects, the application can ensure type safety, improve code readability, and provide a concise way to represent the available parts of speech.

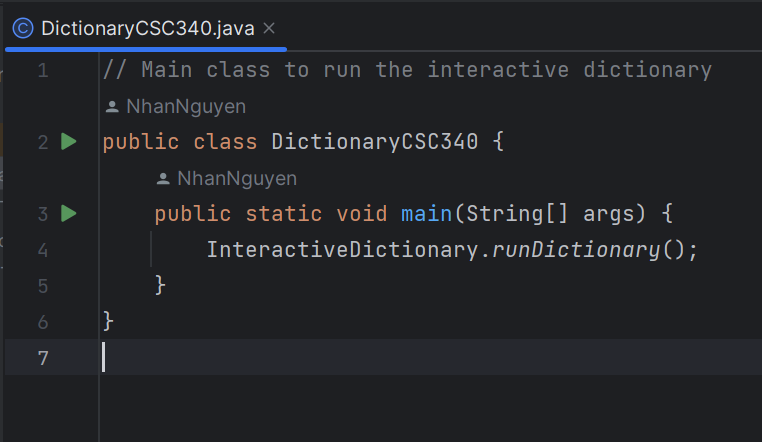
The data structure chosen for the dictionary in the Interactive Dictionary application when loading is a HashMap<String, List<DictionaryData>>. This data structure allows efficient retrieval of dictionary entries based on the search key. The HashMap stores the search words as keys and associates them with a list of DictionaryData objects as values. The DictionaryData class contains information about each dictionary entry, including the word, its part of speech, and other relevant details. The use of a HashMap provides constant-time complexity for average-case lookups, making it a suitable choice for efficient word retrieval. By organizing the dictionary entries based on the search keys, the application can quickly locate and retrieve the relevant data when performing a search operation. The Interactive Dictionary application addresses the problem of efficient word search by providing a user-friendly interface and incorporating various search options. By utilizing enum objects to store data related to dictionary entries, the application ensures type safety and improves code readability. The choice of a HashMap as the data structure for the dictionary enables efficient retrieval of dictionary entries based on search keys. These design choices contribute to a robust and efficient solution for word lookup in the Interactive Dictionary application.

In summary, the analysis highlights the problem-solving approach, the use of enum objects for data storage, and the selection of appropriate data structures for the dictionary. By considering these aspects, I can move to the implementation process to make an Interactive Dictionary application that provides an effective and user-friendly solution for word search and retrieval.

1. **Program Implementation**

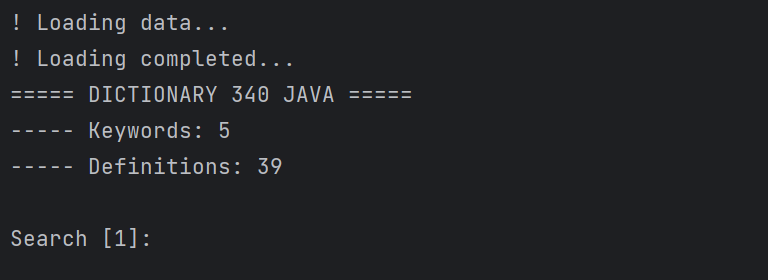
***Now I will demonstrate the program:***

\_ The program allows users to search for words and their definitions based on various parameters such as search key, part of speech, distinct results, and reverse sorting. First, I will execute the runDictionary() method in the main void, which is the entry point for the interactive dictionary:



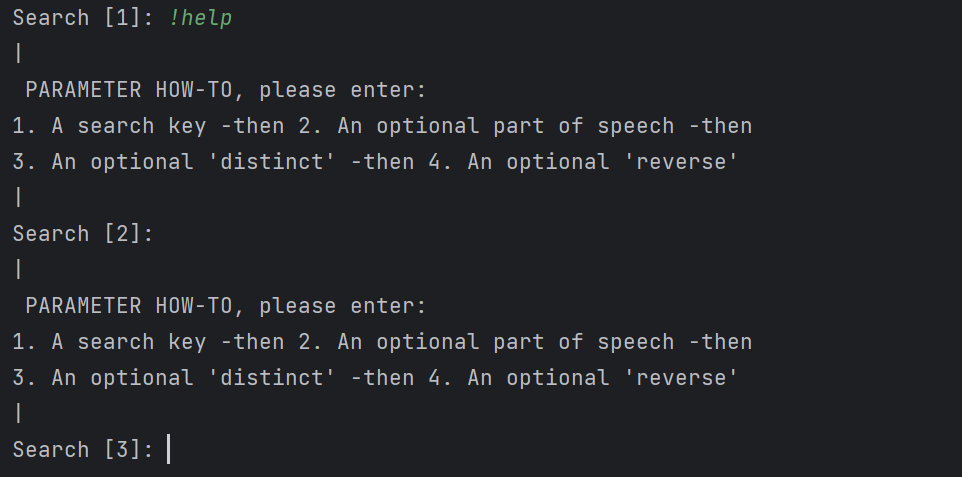
\_Then, the loadDictionaryData() method is called during the initialization of the InteractiveDictionary class. This method loads the dictionary data from the EnumDictionaryData enum into a Map called dictionary, where the search key is mapped to a list of dictionary entries.

\_Then it starts the interactive dictionary loop. It prompts the user for input and processes it until the user want to end the program:



\_Inside the loop, the program reads the user's input and splits it into parts using whitespace as a delimiter.

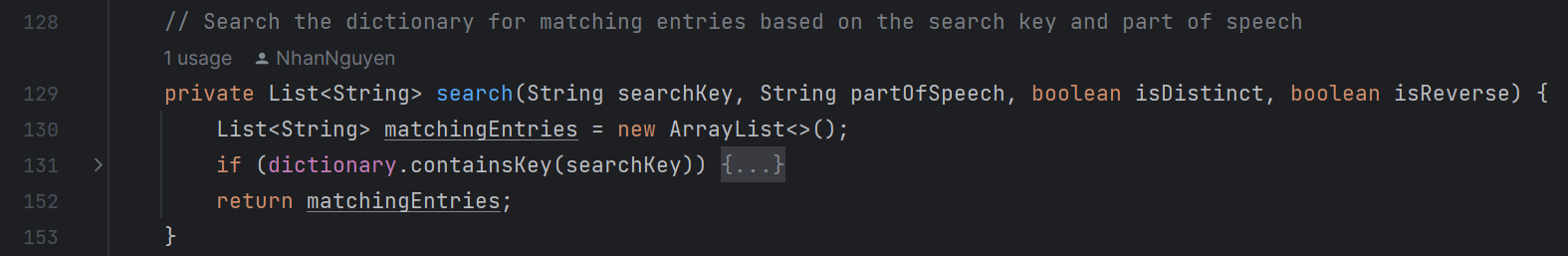
\_If the user enters !help or provides no (or just whitespaces) input, a help message is displayed, explaining how to use the parameters:



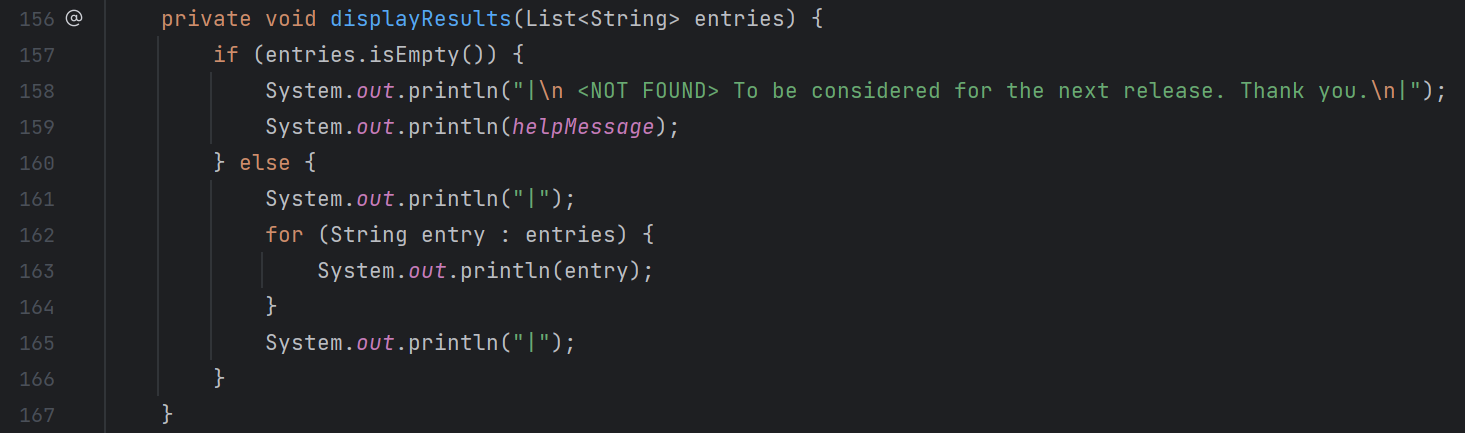
\_If the user enters a valid search key, the program checks if a part of speech is provided and assigns it to the partOfSpeech variable if found. It also checks for the optional parameters distinct and reverse and sets their corresponding boolean flags.

\_After processing the user's input, the program calls the search() method, which searches the dictionary for matching entries based on the search key, part of speech, distinct flag, and reverse flag.

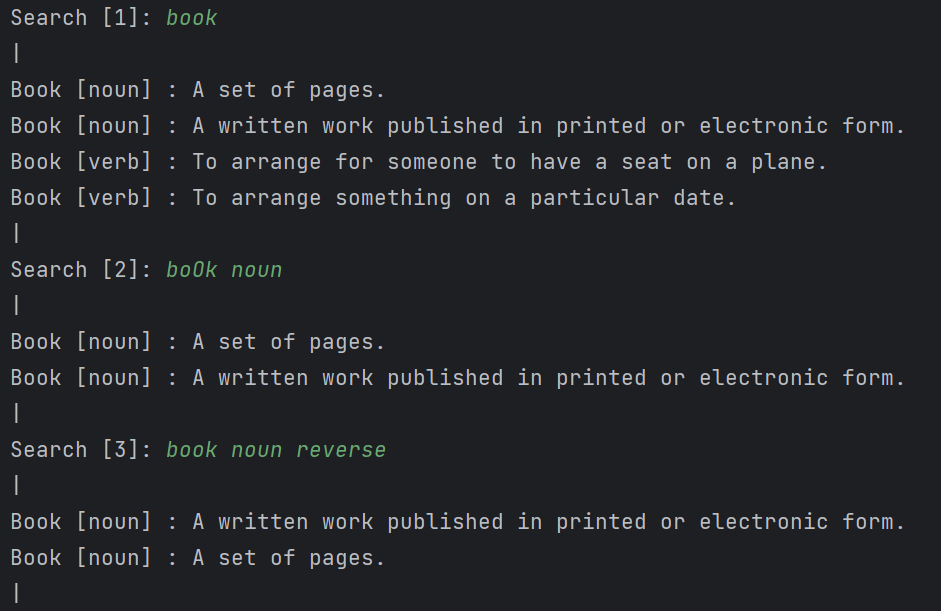
\_The matching entries are returned as a list of strings and returned by the matchingEntries variable:

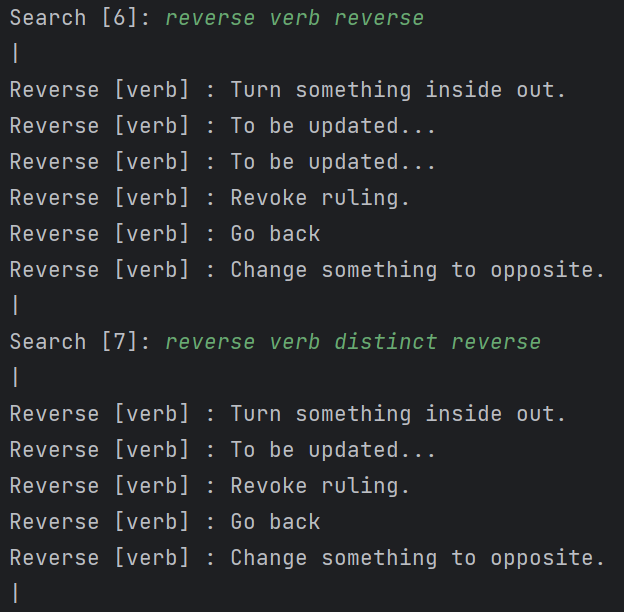


\_The program then calls the displayResults() method, which prints the search results:

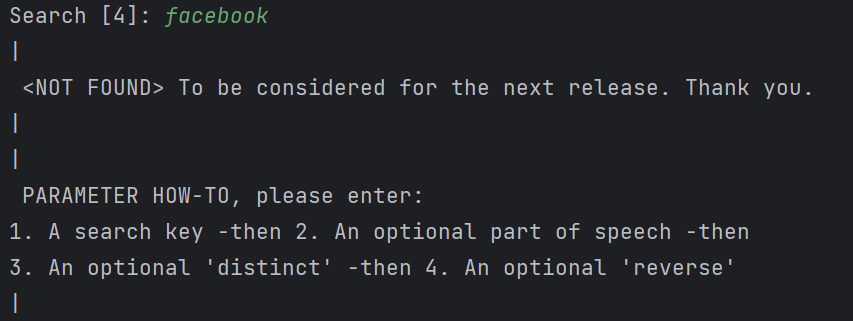


\_Then, the matching entries are displayed:

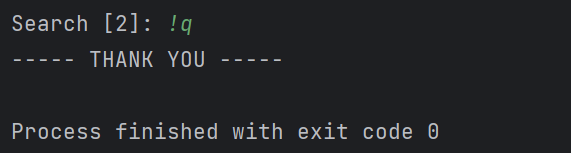




\_If no matching entries are found, a "NOT FOUND" message is displayed, along with the help message:



\_Finally, the loop continues, and the process repeats until the user enters !q, which is used to stop the program:

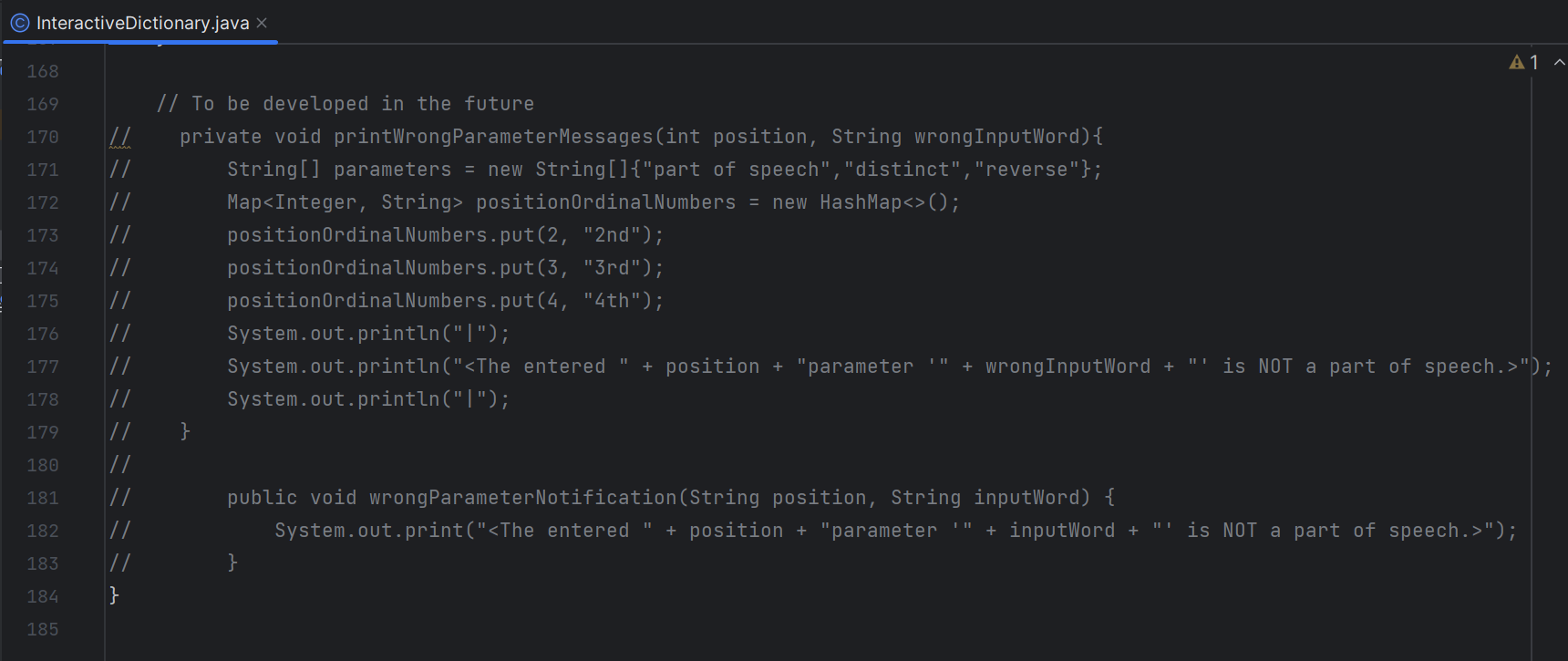


***To improve the program, I am thinking about these points:***

\_Use Enum class in a more effective way: I will not need to name the data like ARROW\_NOUN\_1, BOOK\_NOUN\_1, BOOK\_NOUN\_2, BOOK\_VERB\_1, BOOK\_VERB\_2, which cause many duplicate keywords and have to number the name like that. I’m thinking about changing the data type for the constructot of the enum class to store a map contains multiple defintions and using the keywords one time.

\_ Create an EnumOptions enum: Define an enum called EnumOptions to handle different options and their behavior. For example, I can include options like DISTINCT, REVERSE, and any other options I or the clients want to support more in the future.

\_ Modify the parameter handling messages: I will try to update the parameter handling messages to provide more informative and detailed feedback to the user. The program could benefit from additional error handling and input validation. For example, checking if the user provides valid part of speech or parameter values and displaying appropriate error messages. So, I will consider adding a method to handle parameter error messages and use it where necessary (which I commented in the code file for future development):



\_Implement fuzzy search: In addition to exact keyword matches, it would be helpful to include fuzzy search capabilities. Fuzzy search allows users to find words even if they make minor spelling mistakes or typos. If a search key is not found in the dictionary, I can provide suggestions for similar words or alternative search keys. This can help users find relevant entries even if they don't enter the exact keyword.

\_ Implement wildcard search: Allow users to use wildcard characters (e.g., '\*' or '?') in their search queries to match any character or a single character, respectively. This will broaden the search capability and make it more flexible.

\_ Store dictionary data in an external file or database: Instead of hard-coding the dictionary data in the program, you can store it in an external file (e.g., .CSV or .TXT) or a database. This will make it easier to update and maintain the dictionary data separately from the program code.

\_ Add a feature to add new entries to the dictionary: Allow users to add new entries to the dictionary during runtime. This can be helpful for expanding the dictionary or customizing it based on specific needs.

\_ Implement unit tests: Write unit tests to validate the functionality of individual methods and ensure that the program behaves as expected. This will improve the overall reliability and maintainability of the code.