Assignment 1

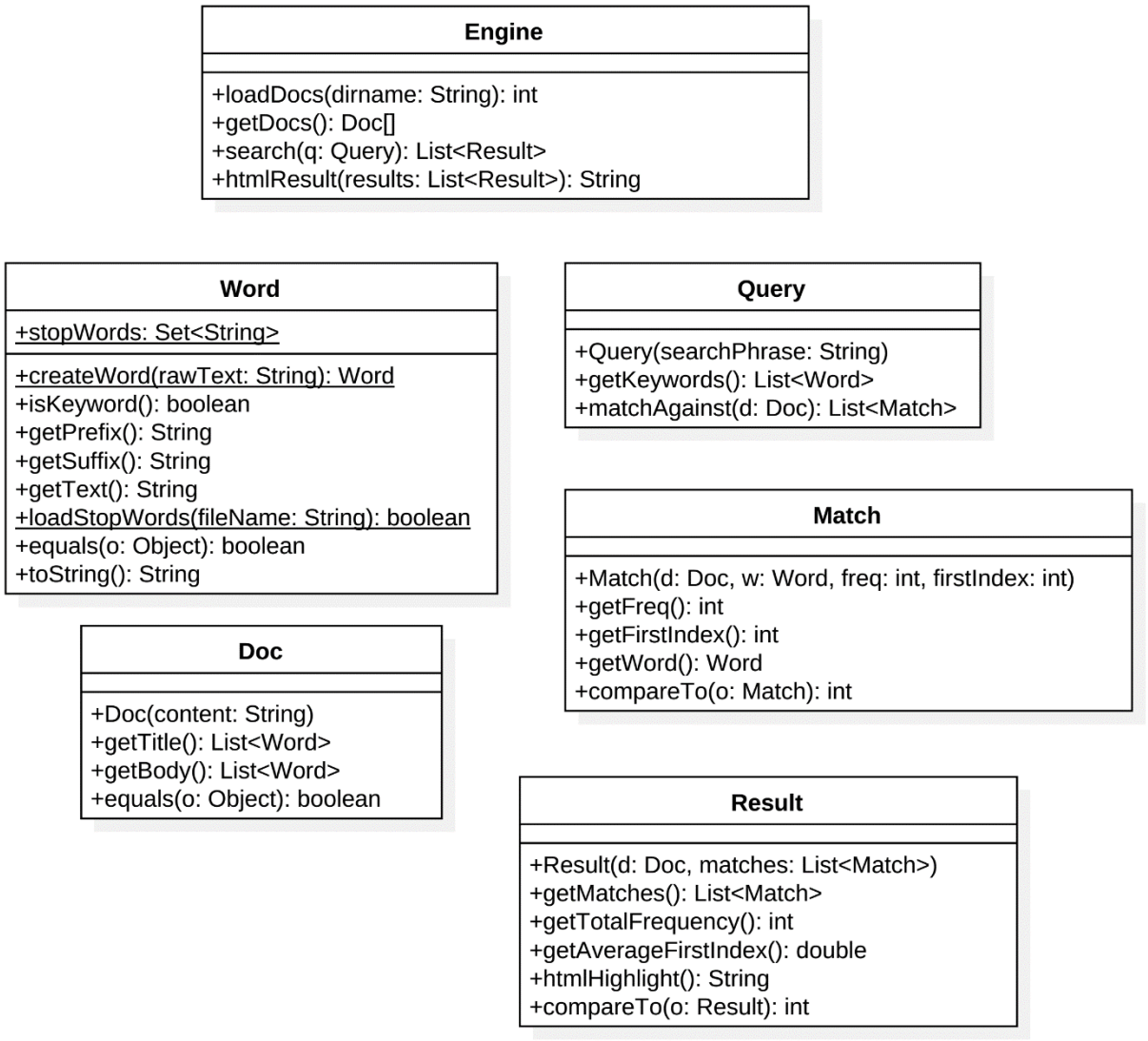
1. Requirements

In this assignment, you're going to build a simple document search engine. This search engine can run queries against a collection of documents. It only searches for the words in queries (called keywords) that aren't in stopwords list. The result is ranked by the frequency of keywords.

This app has these main features:

- Load documents from text files.
- Search for documents by a key phrase.
- Rank search results based on several criteria.
- Generate HTML search results with highlighted keywords.

Program must be written in **Java 8**. The program must contain these classes and methods:



Class	Description

Word	When you split a text by the space (`" "`), you get an array of strings which do not contain space. Some of these strings may be empty. `Word` is the basic class to store each of these strings. Word objects can later be merged together to obtain the original text.
	• The raw text of a word may contain the actual word (the text part) and its surrounding parts (prefix and suffix).
	• Words are divided into valid and invalid words.
	• Valid words are further divided into keywords and stop words (function words).
	The text part of a valid word must have at least one letter and is composed of letters, hyphens and apostrophes. Examples of valid text parts: `Software`, `up-to-date`, `don't`, `doesn't`. There is an exception to the above rule. In case the text part ends with `'s` or `'d`, it should be considered as part of the word's suffix. To clarify this, the string `(other's)` should be converted into a `Word` object which belongs to the valid words category, with the prefix `(`, the text part `other` and the suffix `'s)`. Empty string and non-empty strings which do not have the text part which follow the above rules are invalid words.
	Prefix and suffix of a word are composed of characters which are not alphanumeric. Therefore, the raw texts such as `,se2021.` and `se,se,` should be rendered as invalid words. Invalid words will have empty prefix, empty suffix and text part is the raw text.
	For example, the phrase `in book's "Chapter 5", I've` can be split into 4 sequences `in`, `book's`, `"Chapter` and `5",`. The 1st sequence is the word `in`, a stop word with empty prefix and suffix. The 2nd sequence is the word book with empty prefix and the suffix `'s`. The 3rd sequence is the word `Chapter`, a keyword with the prefix `"` and empty suffix. The 4th sequence is an invalid word. It should be stored as a Word object with empty prefix, empty suffix and the text part of `5",`.
	Public attributes
	`public static Set<String> stopWords;`
	A set of stop words loaded by the `loadStopWords()` method.
	Public methods
	`boolean isKeyword()`
	Returns true if the word is a keyword.
	`public String getPrefix()`
	Returns the prefix part of the word.
	`public String getSuffix()`
	Returns the suffix part of the word.
	`public String getText()`
	Returns the text part of the word.
	`public boolean equals(Object o)`
	Two words are considered equal if their text parts are equal, case-insensitively.
	`public String toString()`
	Returns the raw text of the word.

	`public static Word createWord(String rawText)` Construct and return a complete `Word` object from raw text. *`public static boolean loadStopWords(String fileName)`* Load stop words into the set *`Word.stopWords`* from the text file whose name is specified by `fileName` (which resides under the project's root directory). This method returns `true` if the task is completed successfully and `false` otherwise.

Doc	`Doc` is the class to represent a document which has a title and a body. The title and body of a document are lists of `Word` objects.
	Public methods
	`public Doc(String content)`
	A constructor which receives the raw text of a document and extracts the title and body parts from that. Documents are provided as text files (`.txt`) in the `docs` directory under the project's root directory. To reduce the difficulty of this assignment, each text file contain two lines. The first line is the title and the second line is the body.
	`public List<Word> getTitle()`
	Returns the document's title as a list of `Word` objects.
	`public List<Word> getBody()`
	Returns the document's body as a list of `Word` objects.
	`public boolean equals(Object o)`
	Two `Doc` objects are equal if their titles and bodies contain the same words in the same order. To determine if two words are equal, use the `equals()` method from the `Word` class.

Query	`Query` is the class to represent a user's search query. A `Query` object should store a list of keywords internally.
	Public methods
	`public Query(String searchPhrase)`
	A constructor which receives the raw search phrase from user, then extract only keywords from it.
	`public List<Word> getKeywords()`
	Returns a list of the query's keywords in the same order as they appear in the raw search phrase.
	`public List<Match> matchAgainst(Doc d)`
	Returns a list of matches against the input document. Sort matches by position where the keyword first appears in the document. See the `Match` class for more information about search matches.

Match	A `Match` represents a situation in which a `Doc` contains a `Word`. The search engine's job is to find all documents that are related to a `Query`. Matches are the building blocks of the relationship between documents and a search query. Each Match object only stores information about which Doc contains which Word but also keeps the number of times that the `Word` appears in the `Doc` (frequency) as

	well as the first position which the `Word` appears (first index). When matching a document against a word, the title and body of the document should be combined into a single list of words, with the title placed before the body.
	
	This class must implement the `Comparable<Match>` interface.
	
	Public methods
	
	`public Match(Doc d, Word w, int freq, int firstIndex)`
	
	A constructor to initialize a `Match` object with the document, the word, the frequency of the word in the document and the first position of the word in the document.
	
	`public int getFreq()`
	
	Returns the frequency of the match (as explained above).
	
	`public int getFirstIndex()`
	
	Returns the first index of the match (as explained above).
	
	`public int compareTo(Match o)`
	
	Compare `this` with another `Match` object by the first index. This method obeys the standard behavior specified by Java. `Match` object `A` is greater than `Match` object `B` if the first index of `A` is greater than the first index of `B`.

Result	For a `Query`, the search engine may find a number of related documents. Each document found is represented by a `Result` object. A `Result` object stores information about a related a document, a list of matches found in that document and also three derived properties:
	
	• **match count:** the number of matches, indicated by the size of the list of matches.
	• **total frequency:** the sum of all frequencies of the matches.
	• **average first index:** the average of the first indexes of the matches.
	
	This class must implement the `Comparable<Result>` interface.
	
	Public methods
	
	`public Result(Doc d, List<Match> matches)`
	
	A constructor to initialize a `Result` object with the related document and the list of matches.
	
	`public List<Match> getMatches()`
	
	The method's name explains itself.
	
	`public int getTotalFrequency()`
	
	The method's name explains itself.
	
	`public double getAverageFirstIndex()`
	
	The method's name explains itself.
	
	`public String htmlHighlight()`
	
	Highlight the matched words in the document using HTML markups. For a matched word in the document's title, put the tag `<u>` and `</u>` around the word's text part (the `<u>` tag should not affect the word's prefix and suffix). For a

	matched word in the document's body, surround the word's text part with the tag `<b>` and `</b>`. `public int compareTo(Result o)` These are criteria to determine if `Result A` is greater than `Result B` (in descending order of priority): • `A` has greater *match count* than `B` • `A` has greater *total frequency* than `B` • `A` has lower *average first index* than `B`

Engine	This class represents the search engine.
	Public methods
	`public int loadDocs(String dirname)`
	Loads the documents from the folder specified by `dirname` (which resides under the project's root folder) and returns the number of documents loaded. Refer to the `Doc` class for more information about a `Doc` object.
	`public Doc[] getDocs()`
	Returns an array of documents in the original order.
	`public List<Result> search(Query q)`
	Performs the search function of the engine. Returns a list of sorted search results. Refer to the classes above to know the expected search results.
	`public String htmlResult(List<Result> results)`
	Converts a list of search results into HTML format. The output of this method is the output of `Result.htmlHighlight()` combined together (without any delimiter). Refer to the 3rd line of the file `testCases.html` for a specific example.

You are provided with a project folder containing:

- A folder named `docs` which contains 10 documents saved as text files.
- A `src` folder which contains a package named `engine`. Inside the package, `App.java` has been provided to check your solution locally. This java program checks your solution against an incomplete set of test cases.
- A file named `stopwords.txt` containing the required stop words that you have to load into the search engine.
- A file named `testCases.html` which is used by `App.java`

(*) You shouldn't modify any of the provided files.

Apart from the required public attributes and methods, you are free to add more attributes and methods as you see fit. Refer to the test cases provided in the `App.java` program for specific examples of method outputs.

2. Submission

The provided java source files are put in the `engine` package folder. Rename the `engine` package folder into `a1_sid` where `sid` is your student ID. Put the package folder into a zip file so that when the zip file is opened, you'll see the `a1_sid` folder. **Failure in naming the file or use incorrect file structure as shown will result in no marks being given**.

NO PLAGIARISM: If plagiarism is detected, 0 mark will be given!