Week 4 Review

This exercise is a simplified demonstration of some of the basics of word searching found in search engines like Google, Bing, and Yahoo. The application uses files and folders on your local hard drive and has limited search rules.

Since this review exercise focuses on File I/O and Exceptions, you're responsible for writing the parts of the application that handle files and exceptions. The fully commented search engine code is provided.

Essentially, you'll write the code to gather the files and folders to be searched. Once you have the list of files, you'll open each file, read its contents, and feed it to the search engine.

You'll also create a logging class, which you'll use to log the progress of the application to a file. Finally, you'll use various custom exception classes to report errors, which are handled by the application.

Step One: Getting started

First, import the `m01-week-4-review` project into Eclipse, and review the initial code:

- `Application.java` is the main class of the application.
- `SearchDomain.java` represents the search domain used by the search engine.
- `SearchDomainException.java` extends `Exception` and reports `SearchDomain`-specific exceptions.
- `SearchEngine.java` contains the indexing and search code that makes up the search engine.
- `SearchEngineException.java` extends `Exception` and reports `SearchEngine`-specific exceptions.
- `TELogException.java` extends `RuntimeException` and reports `TELog`-specific exceptions. You'll implement `TELog` in this exercise.

Test data files are provided for your convenience. They're located under the `data` folder. There are six text files, each with a single line.

Step Two: Create the `TELog` class and log application activity

Searching for text is limited to file reading; there's no writing to files. However, since this exercise focuses on reading and writing to files, you'll record all application activity by writing to a log file.

Begin by creating the `TELog` class in the `com.techelevator.util` package, and adding a single `public static` method named `log(String message)`, which takes a `String` parameter and returns `void`.

You'll handle any exceptions internally within `log()` by catching them, capturing their message into a new `TELogException`, and throwing the new exception.

> Note: you don't need to say the `log()` method `throws TELogException` since the exception is an "unchecked" exception.

The `log()` method must open `logs/search.log` for writing, and append the message to the end of the log file. Then, the file must be closed.

> Note: the `logs` folder already exists in the root of the project.

Once you've written the `TELog` class, log the message "Search application started" after the `Step Two` comment in `Application.java`.

Step Three: Complete the `SearchDomain` class

The `SearchDomain` gathers the list of files to search based on the folder name passed into its constructor. For instance, given the following folder and files:

```
data/file1.txt
data/file2.doc
data/file3.dat
data/file4.txt
data/file5.doc
data/file6.dat
```

A call to the constructor `SearchDomain("data")` builds the internal list of files, as you see above.

Your job is to complete the `private List<String> buildDomain()` method by looping through the folder and gathering filenames.

Locating the folder and looping through the files may cause exceptions. You'll handle them internally within `buildDomain()` by catching them, capturing their message into a new `SearchDomainException`, and throwing the new exception. Use the `SearchDomainException`, which extends `Exception`, and add `throws SearchDomainException` to `buildDomain()`.

Step Four: Instantiate a `SearchDomain`

Under the `Step Four` comment in `Application.java`, instantiate a new `SearchDomain`, passing `"data"` as the parameter.

Log the new `SearchDomain` after instantiating it. A `toString()` method has been defined in the class that prints out the names of the files indexed.

Step Five: Index files

Now that the `SearchDomain` has been established, you'll work on the `SearchEngine` class. The `SearchEngine` has two important methods: `indexFiles()` and `search()`. Before performing any searches, the `SearchEngine` must prepare its index.

The `SearchEngine` class has only one constructor, which takes an instance of `SearchDomain`. Complete the `indexFiles()` method where you see the `Step Five` comment.

In the `indexFiles()` method, do the following:

1. Call `SearchDomain`'s `getFiles()` to retrieve the list of filenames in the domain.
2. Loop through the files in the list using a for loop.
3. Open each file and read the contents of the file one line at a time.
4. Pass each line to the `private indexWords()` method, which is described below.

The `indexWords()` method has two parameters: the line as a `String` and an `int` named `fileID`. The `fileID` is the index of the current file in the list of files.

> Note: The for loop you create in the `indexFiles()` method loops through the list of filenames so you have the index for `fileID`. The first filename in the list has the index of `0`, so its `fileID` is `0`. The next file has the index of `1`, so its `fileID` is `1`.

Log the complete index of words after looping through the list, just before exiting the `indexFiles()` method. A `private` convenience method, `indexedWordsToString()`, has been provided for you.

Step Six: Search for a single word

To perform a word search, you need to instantiate a `SearchEngine`, passing in the instance of the `SearchDomain` created earlier. Add the code to create an instance of a `SearchEngine` directly below the `Step-Six` comment line in `Application.java`.

Then, add code to call the `indexFiles()` on the instance of the `SearchEngine`. Once the files in the domain have been indexed, you can search for words. Add a call to `SearchEngine`'s `search()` method. Start with single words—for example, `search("squirrel")` or `search("Larry")`.

`search()` returns a list of filenames where the search word was found. Display the list using a for loop or foreach loop. If no matching files are found, print a message indicating that to the user. You might also consider writing a convenience method in `Application.java` to handle the chore.

Step Seven: Search for multiple words

Searching for multiple words is similar to searching for a single word: a list of files where the words were found is returned, but the order of the list is more refined.

Using the same instance, call `search("telephone line")`. Provided the `SearchDomain` is still the `"data"` folder, the filenames are returned in an order where the most relevant file is first:

```
data/file2.doc
data/file3.dat
data/file6.dat
data/file1.txt
data/file4.txt
```

The order the words are given in is significant. "Telephone" comes before "line", so it must appear before it in the file. There's an implied "and" between words, so both "telephone" and "line" must be present in the same file. Finally, the distance between the words in the file is key to ranking the files in relevance.

In the first three filenames in the list—`file2.doc`, `file3.dat`, and `file6.dat`—"telephone" and "line" have the same distance between them, so they're equally relevant and could be added to the list in any order.

`file1.txt` and `file4.txt`, on the other hand, have different distances between the two words:

- `file1.txt`: `"...telephone wire, or line"...`;
- `file4.txt`: `"...telephone wire belongs to Larry "The Line..."`.

Because the distance is shorter in `file1.txt`, it's deemed more relevant than `file4.txt`, and appears higher up in the list of filenames.

Relevancy is a key feature in all web search engines and in this application.

Bonus challenges

If you finish early, here are two challenge projects you can work on.

Avoid repeatedly opening and closing log file

Repeatedly opening and closing files is very expensive in terms of performance and can significantly slow an application. The search application is small enough that it doesn't really matter, but there is a technique you can use to handle the problem.

Rather than always creating a new instance of `PrintWriter` in the `log()` method, move the local `PrintWriter` variable out of the method, and make it a `static` class variable. Then, in the method, check if the variable is `null`. If it is, create an instance of `PrintWriter` and assign it to the class variable. The next time the `log()` method is called, the class variable is no longer `null`, and the method can use the existing instance of `PrintWriter` to write the message.

> Note: replace any call to the `PrintWriter`'s `close()` method with a call to the `flush()` method. This way, the file can be kept open and the log message written with each call. Also, the `append` argument `true` can be removed from `new FileOutputStream("logs/search.log")` if you'd like to start with a fresh log each time the application runs.

Use `LocalDate`/`LocalDateTime` in TELog

Log files can grow lengthy over time. Knowing the date and time when a message was logged is almost always critical during debugging. Log file names are also usually date or date-time based.

The Java `LocalDate` and `LocalDateTime` classes are the standards for working with dates and times. Introductory tutorials are available at:

LocalDate LocalDateTime Date and Time Parsing and Formatting

Modify the `FileOutputStream` filename parameter to use `LocalDate` instead of hard-coding `logs/search.log`.

Add a date-time stamp to the log message using `LocalDateTime`.

> Hint: `DateTimeFormatter.ISO_DATE` and `DateTimeFormatter.ISO_DATE_TIME` are useful for building a valid log filename and date-time stamping the message.