Programming Assignment 0: Search Engine

Full Specification

This assignment is intended to be a review and warm-up for CSE 123. It will require you to use the skills and concepts that you should be familiar with from your prior programming experience. It will also serve as an introduction to your first IDE, Visual Studio Code. This is designed to help everyone review and practice the programming skills that will be necessary to succeed in CSE 123. While we don't necessarily expect everyone to find this assignment *easy*, if you find yourself having major difficulties with any of the content, please contact the course staff to get support!

Learning Objectives

By completing this assignment, students will demonstrate their ability to:

- Write functionally correct Java programs that meet a provided specification using compound data types
- Write functionally correct Java classes to represent new, compound data types
- Identify errors in a Java program's state or behavior, and implement fixes for identified errors
- Use the Visual Studio debugger to go through a program line by line

Background

Search engines are powerful tools used to help users find information relevant to their needs among very large sets of documents (most often the World Wide Web). While web-based search engines date back to at least the early 1990s, they became more widely used in the mid-90s as both the size and usage of the Web increased. In 1998, Google launched its search engine based on the PageRank algorithm and almost immediately became the dominant engine, a status which it maintains to this day, with almost 90% of all web searches taking place on Google.

While search engines are generally very large and complex systems, at their core, they rely on two key operations: *indexing*, which makes the vast amounts of data being searched easier and more efficient to work with; and *ranking*, which tries to identify which results are most relevant to the user. Users interact with a search engine by entering a *query*, or a sequence of words or tokens that represent what they are looking for (such as `best coffee near me` or `wireless headphones` or `British adventure novels`). The query is used to identify relevant documents in the index, which are then presented to the user in order of rank.

In this assignment, you will implement simple versions of the indexing, ranking, and query operations to build a basic search engine for media (books, movies, etc.).

Assignment Structure

For this assignment, we have broken down the larger problem of building the search engine into a series of smaller problems, each presented on its own coding slide. Apart from the debugging activity, we recommend you proceed through each slide in sequence, then combine the results you produce together in the "Search Engine" slide. However, you are also welcome to complete all your work in the Debugging and Search Engine slides themselves. Regardless of which approach you take, ***be sure to upload all of your code to the Debugging and Search Engine slide, as only work on those slides will be graded.***

Debugging

One of the many benefits of working out of an IDE like VSCode is the ability to enhance your debugging experience. It's important that you are able to take an active role in verifying how your code works and if the output is what you expect. Beyond tests, to support you in this process, we've included a short activity on the Debugging slide designed to help you get comfortable using VSCode and its debugger.

The `Book` Class

First, we'll need to implement a way to represent the media we'll be searching. We've provided you with an interface called `Media` that can be used to represent many different types of media (movies, songs, books, etc.). You will **write a Java class called `Book` that implements the provided `Media` interface and represents a book.** For books, the artists are considered to be the author(s).

Here is the provided `Media` interface for reference:

▼ Expand

```
import java.util.*;

/**
 * An interface to represent various types of media (movies, books, tv shows, songs,
 */
public interface Media {

    /**
     * Gets the title of this media.
     *
     * @return      The title of this media.
     */
    public String getTitle();

    /**
     * Gets all artists associated with this media.
```

```
     *
     * @return      A list of artists for this media.
     */
    public List<String> getArtists();

    /**
     * Adds a rating to this media.
     *
     * @param score     The score for the new rating. Should be non-negative.
     */
    public void addRating(int score);

    /**
     * Gets the number of times this media has been rated.
     *
     * @return      The number of ratings for this media.
     */
    public int getNumRatings();

    /**
     * Gets the average (mean) of all ratings for this media.
     *
     * @return      The average (mean) of all ratings for this media.
     *              If no ratings exist, returns 0.
     */
    public double getAverageRating();

    /**
     * Gets all of the content contained in this media.
     *
     * @ returns    The content stored in a List of strings. If there is no content,
     */
    public List<String> getContent();

    /**
    * Produce a readable string representation. of this media
    *
    * If the media has zero ratings, the format will be:
    * "<title> by [<artists>]"
    *
    * If the media has at least one review, the format will be:
    * "<title> by [<artists>]: <average rating> (<num ratings> ratings)"
    *
    * The average rating displayed will be rounded to at most two decimal places.
    *
    * @ returns     The appropriately formatted string representation
    */
    public String toString();
```

```
}
```

Constructors

Your class should have two constructors:

```
public Book(String title, List<String> authors)
```

- Creates a book with the provided title, list of author(s), and no content.

```
public Book(String title, List<String> authors, Scanner contentScanner)
```

- Creates a book with the provided title, list of author(s), and content in the given `Scanner`.

The author(s) should *not* be able to be modified by a client after creation. The `Scanner` parameter to the constructor should be used to access the content of the book, which is what should be returned by the `getContent` method of the `Media` interface. You should treat each token you read from the `Scanner` as a single piece of content. (Most likely, this will be the actual words in the book, but you do not need to assume that — just read the tokens from the `Scanner`.)

`toString`

In addition to the methods required by the interface, your `Book` class should include a `toString()` method to produce a readable string representation. If the book has zero ratings, the string representation should be:

```
<title> by [<authors>]
```

If the book has at least one review, the string representation should be:

```
<title> by [<authors>]: <average rating> (<num ratings> ratings)
```

The average rating should be rounded to at most two decimal places *in the string representation only*. (The `getAverageRating` method should return the actual average without rounding.)

Inverted Index

Next, we will create an index for our search engine to use. In particular, we will use an approach called an *inverted index* that maps content to locations where that content can be found. In this case, we will map the content obtained from the `getContent` method of the `Media` interface to each

`Media` object that contains that content.

Write a method in the `SearchClient` class called `createIndex` that creates an inverted index for a list of documents. Your method should take one parameter, a list of `Media` objects. Your method should return a map where the keys are individual tokens that appear within each of the `Media` objects (as returned by the `getContent` method), and the values are sets of `Media` objects in which those tokens appear.

Suppose we have a list of `Media` objects `docs` that, when printed out, produces the output:

```
docs = [Up by [Pete Docter], Spirited Away by [Hayao Miyazaki], Interstellar by [Chris
```

and where each element contained the following tokens:

```
docs.get(0).getContent() = [I, am, going, to, Paradise, Falls, if, it, kills, me]
docs.get(1).getContent() = [Once, you, do, something, you, never, forget]
docs.get(2).getContent() = [It, was, never, meant, to, die, here]
```

In this case, an inverted index would return the following map:

```
{am=[Up by [Pete Docter]], die=[Interstellar by [Christopher Nolan]], do=[Spirited Awa
 falls=[Up by [Pete Docter]], forget=[Spirited Away by [Hayao Miyazaki]], going=[Up by
 here=[Interstellar by [Christopher Nolan]], i=[Up by [Pete Docter]], if=[Up by [Pete
 it=[Up by [Pete Docter], Interstellar by [Christopher Nolan]], kills=[Up by [Pete Doc
 me=[Up by [Pete Docter]], meant=[Interstellar by [Christopher Nolan]],
 never=[Spirited Away by [Hayao Miyazaki], Interstellar by [Christopher Nolan]],
 once=[Spirited Away by [Hayao Miyazaki]], paradise=[Up by [Pete Docter]],
 something=[Spirited Away by [Hayao Miyazaki]], to=[Up by [Pete Docter], Interstellar
 was=[Interstellar by [Christopher Nolan]], you=[Spirited Away by [Hayao Miyazaki]]}
```

The method should be case-insensitive (ex, treat "It" and "it" as the same word)**.** This should be done by normalizing the keys to be lowercase.

The keys of the returned map should be in sorted order (`TreeMap`)**, while the sets in the values should prefer fast lookup speed** (`HashSet`)**.**

Search Queries

Finally, we'll put the pieces together into our simplified search engine by implementing a way to get only the documents that are relevant to a user's search.

Search

Write a method in the `SearchClient` class called `search` that takes two parameters: an index in the

format created by your `createIndex` method and a String representing the query for the search. Your method should do the following:

- The method should consider the `query` **case-insensitively**.
    - This should be done by normalizing the `query` to be lowercase.
- Split the `query` into individual **tokens**.
    - More details below.
- Return the list of media that contained at least one **token**, where media with higher **token** counts should appear earlier in the list.
    - If two media instances have the same token count, then either order is acceptable.

To give a potential example, consider `docs` from earlier:

```
docs = [Up by [Pete Docter], Spirited Away by [Hayao Miyazaki], Interstellar by [Chris
```

and its inverted index:

```
{am=[Up by [Pete Docter]], die=[Interstellar by [Christopher Nolan]], do=[Spirited Awa
 falls=[Up by [Pete Docter]], forget=[Spirited Away by [Hayao Miyazaki]], going=[Up by
 here=[Interstellar by [Christopher Nolan]], i=[Up by [Pete Docter]], if=[Up by [Pete
 it=[Up by [Pete Docter], Interstellar by [Christopher Nolan]], kills=[Up by [Pete Doc
 me=[Up by [Pete Docter]], meant=[Interstellar by [Christopher Nolan]],
 never=[Spirited Away by [Hayao Miyazaki], Interstellar by [Christopher Nolan]],
 once=[Spirited Away by [Hayao Miyazaki]], paradise=[Up by [Pete Docter]],
 something=[Spirited Away by [Hayao Miyazaki]], to=[Up by [Pete Docter], Interstellar
 was=[Interstellar by [Christopher Nolan]], you=[Spirited Away by [Hayao Miyazaki]]}
```

If the query was "to paradise", then the returned list would be `[Up by [Pete Docter], Interstellar by [Christopher Nolan]]`. Note that `Up` appears before `Interstellar` because it contains the tokens "to" and "paradise", whereas `Interstellar` only contains the token "to".

Tokenizing Strings

To accomplish this, you have two options

1. Using `Scanner`

- You can initialize a `Scanner` object with the given `query`, and repeatedly call `.next()` to retrieve each token.
- Ex:

```
Scanner queryScanner = new Scanner(query);
while (queryScanner.hasNext()) {
    String token = queryScanner.next();
    (...)
}
```

2. Using the `split` method of the `String` class.

- The `split` method of the `String` class takes a String delimiter (e.g., "what to split by") as a parameter and returns your original large String as an array of smaller Strings. You can call the `split` method with the provided `TOKEN_SEPARATOR` variable.
    - If you are curious, the delimiter `String` passed to `split` is called a *regular expression*, which is a string that uses a particular syntax to indicate patterns of text. **You do not need to have a deep understanding of regular expressions to use this method.**
- Ex:

```
String[] tokens = query.split(TOKEN_SEPARATOR);
for (String token : tokens) {
    (...)
}
```

Testing Requirements

We have provided an incomplete `Testing.java` file that you should update lines according to the guiding comments within. You should only have to address 16 TODO's within this file so that it compiles and accurately tests your implementations.

WARNING: We've provided you a test that checks if your `Testing.java` file compiles and no tests fail. It does not check that the appropriate updates were made according to the comments within the file. It is your responsibility to make sure that you're updating the file correctly.

NOTE: You do not need to worry about following the Code Quality or Commenting guide when implementing your own tests. Testing will **never** count towards code quality or commenting grades.

We recommend reading over this file to better understand how to write JUnit tests. As the quarter progresses, we will be providing you with less testing guidance, so if you have any questions or confusion, it's best to ask them now!

Implementation Guidelines

As always, your code should follow all guidelines in the Code Quality Guide and Commenting Guide. In particular, pay attention to these requirements and hints:

- You should avoid re-implementing the functionality of already existing methods by just calling those existing methods.
- You should make all of your fields private, and you should reduce the number of fields to only those that are necessary for solving the problem.
- Program defensively by making sure the client never has a direct reference to mutable internal fields. In other words, make defensive copies of passed-in objects in your constructor, and make defensive copies of objects returned by getter methods.
- Each of your fields should be initialized inside of your constructor(s).
- When applicable, reduce redundancy by using the `this()` keyword to call another constructor in the same class.
- When possible, avoid redundant method calls. i.e., method calls that serve no functional purpose.
- You should comment your code following the Commenting Guide. You should write comments with basic info (a header comment at the top of your file), a class comment for every class, and a comment for every method other than main.
    - Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.
- Any additional helper methods created, but not specified in the spec, should be declared ***private***.

Feeling Stuck?

While we expect this assignment to be review, it's still OK if you find this assignment a bit challenging! Remember that learning is a challenging process, and you don't have to do it alone!

- You can visit the Introductory Programming Lab (IPL) to talk with a TA about programming concepts or get help on assignments.
- You can stop by instructor office hours to discuss course concepts, get help on assignments, or discuss the course in general.
- You can post questions on the discussion board! You can make questions public (anyone can see them) or private (only course staff can see them). This is a great way to asynchronously get help on an assignment or ask questions about the course.

It is OK to get stuck and feel challenged by this assignment. However, note that this is intended to be a warm-up for the type of programming we will be doing for the rest of the quarter, and the tasks we will be solving in future weeks will be more complex than these problems and rely on a solid grasp of the skills practiced in this assignment. If you feel like you cannot do this assignment at all, we recommend reaching out to the course instructor (mtrienv@cs.washington.edu) or the CSE undergrad advisors (ugrad-adviser@cs.washington.edu) to discuss more about academic planning and which programming course might be a good fit for your goals.

Submission

When you are ready to submit, go to the " Final Submission " slide, read the statement, and fill in the box, then click "Submit" in the upper-right corner. You may submit as many times as you want until the due date.

You can see your previous submissions by clicking the three dots icon in the upper-right and selecting "Submissions and Grades." By default, we will grade your latest submission from before the deadline. However, if you would like us to grade a different submission, you can select that submission on the left side of the window and click "Set final." Note that we will not grade any submission made after the deadline-- if you mark a submission after the deadline as final, we will grade your most recent on-time submission instead.

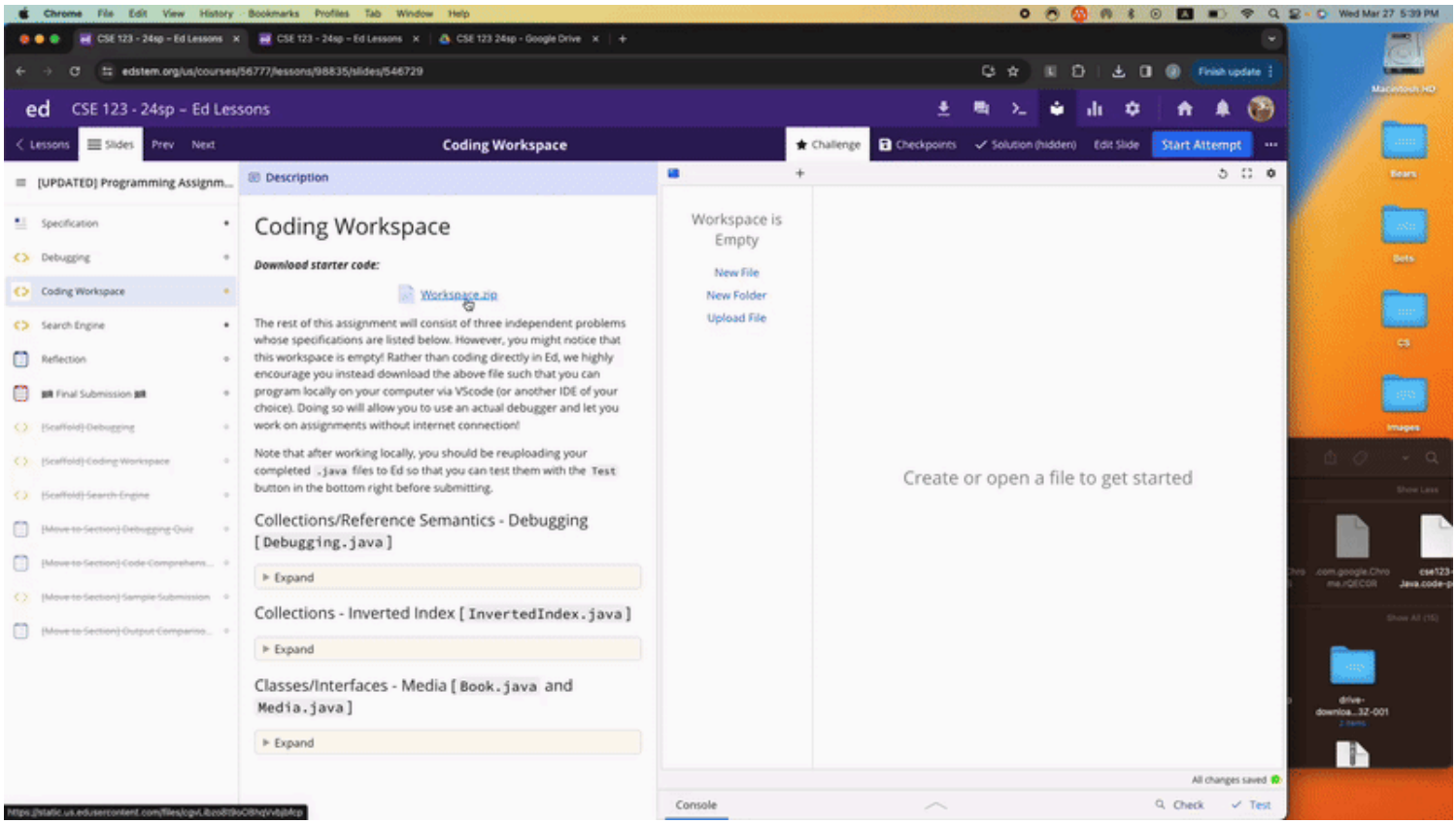
Please make sure you are familiar with the resources and policies outlined in the syllabus and the assignments page.

Opening Code in VSCode

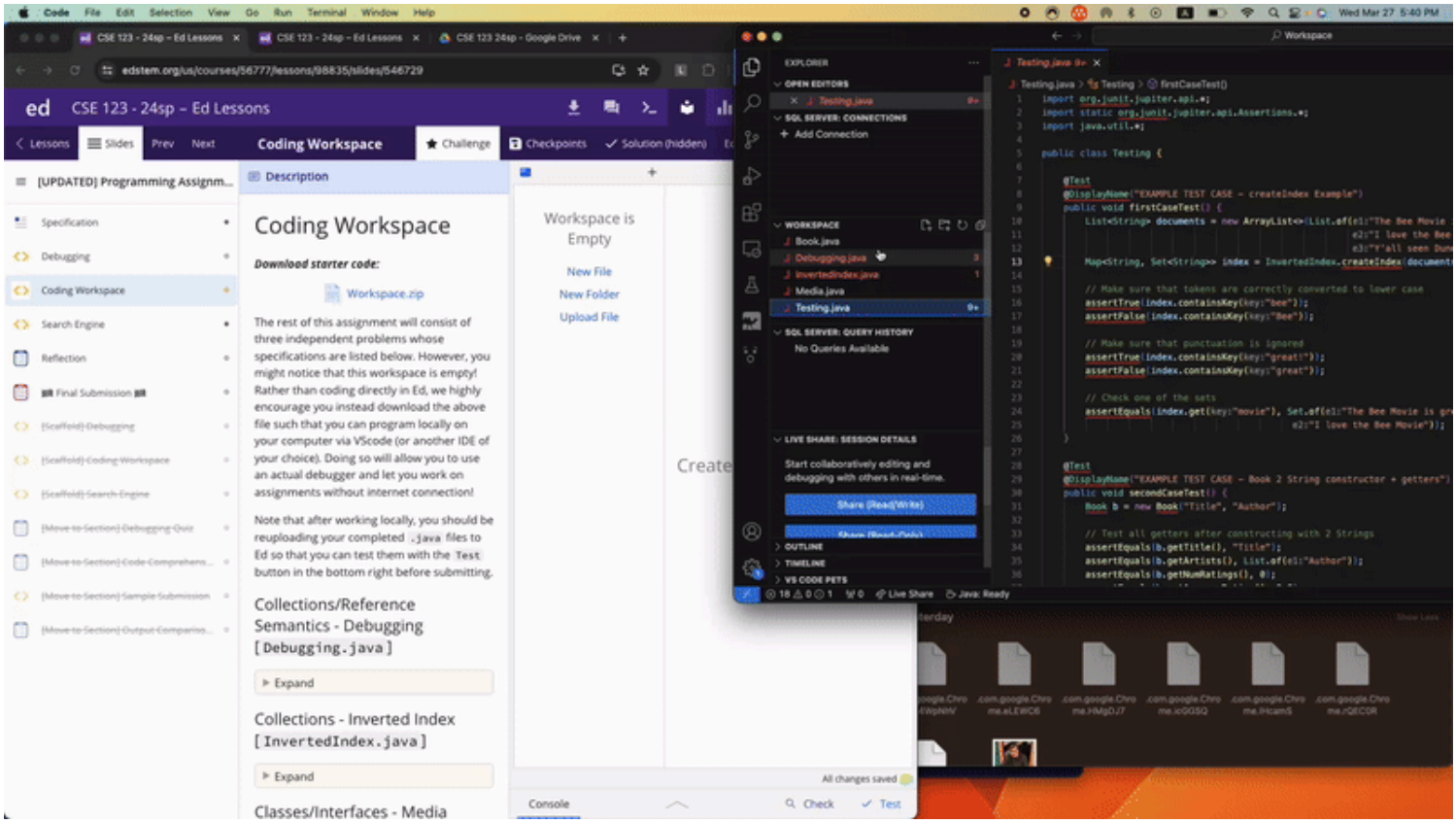
The general process we're expecting you to follow when working on and submitting HW assignments is slightly different from previous courses now that you have an IDE (VSCode)! You should

1. Download the provided `.zip` file
2. Find it in `Finder` / `File Explorer`
3. Unzip it to get the folder with relevant files.
4. In VSCode, click File > Open Folder, and select the recently unzipped folder to open it!

This process is outlined in the following `.gif` (mac shown, but the process would be the same for windows with File Explorer)



Then, once you've finished, you should re-upload your code by copy-pasting or drag-dropping the files back into Ed! At this point you can run tests via the `Test` button.


CSE 123 - 24sp – Ed Lessons
edstem.org/us/courses/56777/lessons/98835/slides/546729
Lessons
Slides
Prev
Next
Coding Workspace
Challenge
Checkpoints
Solution (hidden)
[UPDATED] Programming Assignm...
Specification
Debugging
Coding Workspace
Search Engine
Reflection
Final Submission
Description
Coding Workspace
Download starter code:
Workspace.zip
The rest of this assignment will consist of three independent problems whose specifications are listed below. However, you might notice that this workspace is empty! Rather than coding directly in Ed, we highly encourage you instead download the above file such that you can program locally on your computer via VScode (or another IDE of your choice). Doing so will allow you to use an actual debugger and let you work on assignments without internet connection!
Note that after working locally, you should be reuploading your completed .java files to Ed so that you can test them with the Test button in the bottom right before submitting.
Collections/Reference Semantics - Debugging [Debugging.java]
Expand
Collections - Inverted Index [InvertedIndex.java]
Expand
Classes/Interfaces - Media
Workspace is Empty
New File
New Folder
Upload File
All changes saved
Console
Check
Test
EXPLORER
OPEN EDITORS
SQL SERVER: CONNECTIONS
Add Connection
WORKSPACE
Book.java
Debugging.java
InvertedIndex.java
Media.java
Testing.java
SQL SERVER: QUERY HISTORY
No Queries Available
LIVE SHARE: SESSION DETAILS
Start collaboratively editing and debugging with others in real-time.
Share (Read/Write)
OUTLINE
TIMELINE
VS CODE PETS
Live Share
Java: Ready
Testing.java
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import java.util.*;
public class Testing {
@Test
@DisplayName("EXAMPLE TEST CASE - createIndex Example")
public void firstCaseTest() {
List<String> documents = new ArrayList<>(List.of(e1:"The Bee Movie
e2:"I love the Bee
e3:"Y'all seen Dun
Map<String, Set<String>> index = InvertedIndex.createIndex(document
// Make sure that tokens are correctly converted to lower case
assertTrue(index.containsKey(key:"bee"));
assertFalse(index.containsKey(key:"Bee"));
// Make sure that punctuation is ignored
assertTrue(index.containsKey(key:"great!"));
assertFalse(index.containsKey(key:"great"));
// Check one of the sets
assertEquals(index.get(key:"movie"), Set.of(e1:"The Bee Movie is gr
e2:"I love the Bee Movie"));
}
@Test
@DisplayName("EXAMPLE TEST CASE - Book 2 String constructor + getters")
public void secondCaseTest() {
Book b = new Book("Title", "Author");
// Test all getters after constructing with 2 Strings
assertEquals(b.getTitle(), "Title");
assertEquals(b.getArtists(), List.of(e1:"Author"));
assertEquals(b.getNumRatings(), 0);

[GRADED] Debugging

This slide **IS** graded.

One of the many benefits of working out of an IDE like VSCode is the ability to enhance your debugging experience. It's important that you are able to take an active role in verifying how your code works and if the output is what you expect. Beyond tests, to support you in this process, we've included a short activity on the Debugging slide designed to help you get comfortable using VSCode and its debugger.

Hopefully, you've had a chance to complete the Software Setup for this course. If not, no worries—you'll want to finish that first before moving on with this slide.

Once you have set up the Visual Studio "IDE" (integrated development environment – an application that helps you code), download and open the file below. You will have to determine the correct code needed to unlock the query by using the debugger.

Download starter code:

P0_Debugging.zip

NOTE: You should only need to update `line 30` of the provided file: `unlock("00000");` to the defusal code you determine using the VSCode debugger!

WARNING: We're trying our best to encourage you to use the VSCode debugger here, so you may find that printlns don't actually print anything! This is expected behavior :)

This Debugger Guide will help guide you through the process!

After you've used the VS Code Debugger to crack the code, you should upload your completed `QueryCracker.java` file to Ed so that you can test with the `Test` button in the bottom right before submitting.

Debugger Tutorial

[NOT GRADED] Media

This slide is **NOT** graded.

Download starter code:

P0_Media.zip

This is the first of two slides where you will implement the beginning parts of your assignment. However, you might notice that this workspace is empty! Rather than coding directly in Ed, we highly encourage you to instead download the above file so that you can program locally on your computer via VSCode (or another IDE of your choice). Doing so will allow you to use an actual debugger and will let you work on assignments without an internet connection!

Note that after working locally, you should be reuploading your completed `.java` files to Ed so that you can test them with the `Test` button in the bottom right before submitting.

Expand

```
til.*;

/**
 * An interface to represent various types of media (movies, books, tv shows, songs,
 */
public interface Media {

    /**
     * Gets the title of this media.
     *
     * @return      The title of this media.
     */
    public String getTitle();

    /**
     * Gets all artists associated with this media.
     *
     * @return      A list of artists for this media.
     */
    public List<String> getArtists();

    /**
     * Adds a rating to this media.
     *
```

```
     * @param score     The score for the new rating. Should be non-negative.
     */
    public void addRating(int score);

    /**
     * Gets the number of times this media has been rated.
     *
     * @return      The number of ratings for this media.
     */
    public int getNumRatings();

    /**
     * Gets the average (mean) of all ratings for this media.
     *
     * @return      The average (mean) of all ratings for this media.
     *              If no ratings exist, returns 0.
     */
    public double getAverageRating();

    /**
     * Gets all of the content contained in this media.
     *
     * @ returns    The content stored in a List of strings. If there is no content,
     */
    public List<String> getContent();

    /**
    * Produce a readable string representation. of this media
    *
    * If the media has zero ratings, the format will be:
    * "<title> by [<artists>]"
    *
    * If the media has at least one review, the format will be:
    * "<title> by [<artists>]: <average rating> (<num ratings> ratings)"
    *
    * The average rating displayed will be rounded to at most two decimal places.
    *
    * @ returns     The appropriately formatted string representation
    */
    public String toString();
}
```

Constructors

Your class should have two constructors:

```
public Book(String title, List<String> authors)
```

- Creates a book with the provided title, list of author(s), and no content.

```
public Book(String title, List<String> authors, Scanner contentScanner)
```

- Creates a book with the provided title, list of author(s), and content in the given `Scanner`.

The author(s) should *not* be able to be modified by a client after creation. The `Scanner` parameter to the constructor should be used to access the content of the book, which is what should be returned by the `getContent` method of the `Media` interface. You should treat each token you read from the `Scanner` as a single piece of content. (Most likely, this will be the actual words in the book, but you do not need to assume that — just read the tokens from the `Scanner`.)

`toString`

In addition to the methods required by the interface, your `Book` class should include a `toString()` method to produce a readable string representation. If the book has zero ratings, the string representation should be:

```
<title> by [<authors>]
```

If the book has at least one review, the string representation should be:

```
<title> by [<authors>]: <average rating> (<num ratings> ratings)
```

The average rating should be rounded to at most two decimal places *in the string representation only*. (The `getAverageRating` method should return the actual average without rounding.)

[NOT GRADED] Inverted Index

This slide is **NOT** graded.

Download starter code:

P0_InvertedIndex.zip

To get started, download the zip file above, and paste in your implementation of `Book.java` from the previous slide.

Similar to the previous slide, rather than coding directly in Ed, we highly encourage you to instead download the above file such that you can program locally on your computer via VSCode (or another IDE of your choice). Doing so will allow you to use an actual debugger and will let you work on assignments without an internet connection!

Note that after working locally, you should reupload your completed `.java` files to Ed so that you can test them with the `Test` button in the bottom right before submitting.

NOTE: The tests for your `createIndex` implementation are tied to a working `Book` implementation. This means that you should be passing all `Book` tests before moving on to implementing `createIndex`!

WARNING: We've noticed that sometimes VSCode will automatically include an import for `javax.print.attribute.standard.Media;`. Please delete this if you notice it present within your `InvertedIndex.java` file!

Inverted Index

Next, we will create an index for our search engine to use. In particular, we will use an approach called an *inverted index* that maps content to locations where that content can be found. In this case, we will map the content obtained from the `getContent` method of the `Media` interface to each `Media` object that contains that content.

Write a method in the `SearchClient` class called `createIndex` that creates an inverted index for a list of documents. Your method should take one parameter, a list of `Media` objects. Your method should return a map where the keys are individual tokens that appear within each of the `Media` objects (as returned by the `getContent` method), and the values are sets of `Media` objects in which those tokens appear.

Suppose we have a list of `Media` objects `docs` that, when printed out, produces the output:

```
docs = [Up by [Pete Docter], Spirited Away by [Hayao Miyazaki], Interstellar by [Chris
```

and where each element contained the following tokens:

```
docs.get(0).getContent() = [I, am, going, to, Paradise, Falls, if, it, kills, me]
docs.get(1).getContent() = [Once, you, do, something, you, never, forget]
docs.get(2).getContent() = [It, was, never, meant, to, die, here]
```

In this case, an inverted index would return the following map:

```
{am=[Up by [Pete Docter]], die=[Interstellar by [Christopher Nolan]], do=[Spirited Awa
 falls=[Up by [Pete Docter]], forget=[Spirited Away by [Hayao Miyazaki]], going=[Up by
 here=[Interstellar by [Christopher Nolan]], i=[Up by [Pete Docter]], if=[Up by [Pete
 it=[Up by [Pete Docter], Interstellar by [Christopher Nolan]], kills=[Up by [Pete Doc
 me=[Up by [Pete Docter]], meant=[Interstellar by [Christopher Nolan]],
 never=[Spirited Away by [Hayao Miyazaki], Interstellar by [Christopher Nolan]],
 once=[Spirited Away by [Hayao Miyazaki]], paradise=[Up by [Pete Docter]],
 something=[Spirited Away by [Hayao Miyazaki]], to=[Up by [Pete Docter], Interstellar
 was=[Interstellar by [Christopher Nolan]], you=[Spirited Away by [Hayao Miyazaki]]}
```

The method should be case-insensitive (ex, treat "It" and "it" as the same word)**.** This should be done by normalizing the keys to be lowercase.

The keys of the returned map should be in sorted order (`TreeMap`)**, while the sets in the values should prefer fast lookup speed** (`HashSet`)**.**

[GRADED] Search Engine

This slide **IS** GRADED.

Download starter code:

P0_SearchEngine.zip

You might notice that this workspace is empty! Rather than coding directly in Ed, we highly encourage you to instead download the above file so that you can program locally on your computer via VSCode (or another IDE of your choice). Doing so will allow you to use an actual debugger and will let you work on assignments without an internet connection!

Note that after working locally, you should reupload your completed `.java` files to Ed so that you can test them with the `Test` button in the bottom right before submitting.

WARNING: We've noticed that sometimes VSCode will automatically include an import for `javax.print.attribute.standard.Media;`. Please delete this if you notice it present within your `SearchClient.java` file!

NOTE: You don't need to worry about reuploading the books directory as it's already included in the scaffold!

Search Queries

Finally, we'll put the pieces together into our simplified search engine by implementing a way to get only the documents that are relevant to a user's search.

Most of the work has been done for you here, but you'll have to integrate your implementations from the helper slides. This will involve 3 main steps:

1. Paste your implementation of `Book` within `Book.java`
2. Paste your implementation of `createIndex` @ line 45

Then, fill in the method in the `SearchClient` class called `search` (located at line 50) that takes two parameters: an index in the format created by your `createIndex` method and a String representing the query for the search.

NOTE: the line numbers might change after pasting in your `createIndex` implementation, but is marked with a "TODO" comment so you can search for within the file

Write a method in the `SearchClient` class called `search` that takes two parameters: an index in the

format created by your `createIndex` method and a String representing the query for the search. Your method should do the following:

- The method should consider the `query` **case-insensitively**.
    - This should be done by normalizing the `query` to be lowercase.
- Split the `query` into individual **tokens**.
    - More details below.
- Return the list of media that contained at least one **token**, where media with higher **token** counts should appear earlier in the list.
    - If two media instances have the same token count, then either order is acceptable.

To give a potential example, consider `docs` from earlier:

```
docs = [Up by [Pete Docter], Spirited Away by [Hayao Miyazaki], Interstellar by [Chris
```

and its inverted index:

```
{am=[Up by [Pete Docter]], die=[Interstellar by [Christopher Nolan]], do=[Spirited Awa
 falls=[Up by [Pete Docter]], forget=[Spirited Away by [Hayao Miyazaki]], going=[Up by
 here=[Interstellar by [Christopher Nolan]], i=[Up by [Pete Docter]], if=[Up by [Pete
 it=[Up by [Pete Docter], Interstellar by [Christopher Nolan]], kills=[Up by [Pete Doc
 me=[Up by [Pete Docter]], meant=[Interstellar by [Christopher Nolan]],
 never=[Spirited Away by [Hayao Miyazaki], Interstellar by [Christopher Nolan]],
 once=[Spirited Away by [Hayao Miyazaki]], paradise=[Up by [Pete Docter]],
 something=[Spirited Away by [Hayao Miyazaki]], to=[Up by [Pete Docter], Interstellar
 was=[Interstellar by [Christopher Nolan]], you=[Spirited Away by [Hayao Miyazaki]]}
```

If the query was "to paradise", then the returned list would be `[Up by [Pete Docter], Interstellar by [Christopher Nolan]]`. Note that `Up` appears before `Interstellar` because it contains the tokens "to" and "paradise", whereas `Interstellar` only contains the token "to".

Tokenizing Strings

To accomplish this, you have two options

1. Using `Scanner`

- You can initialize a `Scanner` object with the given `query`, and repeatedly call `.next()` to retrieve each token.
- Ex:

```
Scanner queryScanner = new Scanner(query);
while (queryScanner.hasNext()) {
    String token = queryScanner.next();
    (...)
}
```

2. Using the `split` method of the `String` class.

- The split method of the String class, which takes a String delimiter (e.g., "what to split by") as a parameter and returns your original large String as an array of smaller Strings. You can call the `split` method with the provided `TOKEN_SEPARATOR` variable.
    - If you are curious, the delimiter `String` passed to `split` is called a *regular expression*, which is a string that uses a particular syntax to indicate patterns of text. **You do not need to have a deep understanding of regular expressions to use this method.**
- Ex:

```
String[] tokens = query.split(TOKEN_SEPARATOR);
for (String token : tokens) {
    (...)
}
```

Testing Requirements

We have provided an incomplete `Testing.java` file that you should update lines according to the guiding comments within. You should only have to address 16 TODO's within this file so that it compiles and accurately tests your implementations.

WARNING: We've provided you a test that checks if your `Testing.java` file compiles and no tests fail. It does not check that the appropriate updates were made according to the comments within the file. It is your responsibility to make sure that you're updating the file correctly.

NOTE: You do not need to worry about following the Code Quality or Commenting guide when implementing your own tests. Testing will **never** count towards code quality or commenting grades.

We recommend reading over this file to better understand how to write JUnit tests. As the quarter progresses, we will be providing you with less testing guidance, so if you have any questions or confusion, it's best to ask them now!

You can hit `Check` **(not** `Test`**)** to see a descriptive output of running your tests in `Testing.java`.

Implementation Guidelines

As always, your code should follow all guidelines in the Code Quality Guide and Commenting Guide.

In particular, pay attention to these requirements and hints:

- You should avoid re-implementing the functionality of already existing methods by just calling those existing methods.
- You should make all of your fields private, and you should reduce the number of fields to only those that are necessary for solving the problem.
- Program defensively by making sure the client never has a direct reference to mutable internal fields. In other words, make defensive copies of passed-in objects in your constructor, and make defensive copies of objects returned by getter methods.
- Each of your fields should be initialized inside of your constructor(s).
- When applicable, reduce redundancy by using the `this()` keyword to call another constructor in the same class.
- When possible, avoid redundant method calls. i.e., method calls that serve no functional purpose.
- You should comment your code following the Commenting Guide. You should write comments with basic info (a header comment at the top of your file), a class comment for every class, and a comment for every method other than main.
    - Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.
- Any additional helper methods created, but not specified in the spec, should be declared ***private***.

Reflection

For this week's reflection, we would like everyone to watch and engage with the following video.

Step 1: Watch The moral bias behind your search results | Andreas Ekström (9m 19s)

Question 1

Step 2: Write your own comment below, responding to one of the following prompts in its entirety:

Prompt 1: At Google, who's responsibility do you think it is to come up with moral rules and judgements surrounding search engine ranking results as described in the video? (Executives, managers, software engineers, other). Do you think that they should have that power / responsibility and why? If not, who do you think should have this responsibility?

Prompt 2: Do you think search engine providers (Google, Bing, etc.) have an obligation to remind their users that "unbiased, clean search results" can't truly exist as mentioned within the video? Why? If you do think so, are they currently acting on that obligation? Why might that be the case?

Prompt 3: How do you feel about the video's claim that software reflects the biases of the programmer? Do you agree / disagree? Why? Give an example of a biased program / application in your day-to-day life.

Please label which prompt you are answering. For full credit, all questions within a prompt

must be clearly answered. In particular, we suggest using the ACE (answer, cite, and explain) format. For a meaningful response, it may be helpful to pose some counterexamples, connect in terms of your own experience, add additional support from course materials, and be as specific as possible in your own reasoning

Question 2

Step 3: The following questions will ask that you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

What skills did you learn and/or practice with working on this assignment?

Question 3

What did you struggle with most on this assignment?

Question 4

What questions do you still have about the concepts and skills you used in this assignment?

Question 5

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

Question 6

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

Question 7

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

Final Submission

Final Submission

Fill out the box below and click "Submit" in the upper-right corner of the window to submit your work.

Question

I attest that the work I am about to submit is my own and was completed according to the course Academic Honesty and Collaboration policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type "yes" as your response.)