Introduction

Digital libraries have transformed how we access and explore vast collections of books, making it easy to search, sort, and find exactly what we need with just a few keystrokes. The basic search engine is at the heart of this and many other modern conveniences.

In this lab, you will create a Java search engine. Your application will allow users to search for a file by name, read and populate objects with data, and sort these objects using algorithms. It will also feature a user-friendly menu, log activities, search data, and serialize objects for storage. By the end, you'll have hands-on experience with data structures, algorithms, and file I/O in Java while building a functional digital library system.

Scenario

The digital library system you'll develop manages a collection of books, with all book data stored in a single file named "*books.txt*". Each line in this file represents a book, including its title, author, and publication year; for example:

- The Great Gatsby,F. Scott Fitzgerald,1925
- To Kill a Mockingbird,Harper Lee,1960
- 1984,George Orwell,1949

Your search engine program will allow users to search for specific books within this file and view their details. Users should be able to sort the list of books based on various criteria, such as title, author, or publication date. And to provide a seamless user experience, you'll create a text-based menu-driven interface that facilitates easy navigation through these options.

Additionally, to ensure that user data and interactions are preserved across sessions, you will implement serialization to save the current state of the library to a file named "*library.ser*." This will allow the program to restore the library's state the next time it is started. User interactions, such as searches and sorts, will also be logged to a file named “*user_interactions.log*” for auditing and debugging purposes.

Requirements

- Accurately read and parse book data from a file, create *Book* objects, and add them to the books list.
- Apply sorting algorithms (e.g. bubble sort) to organize the books list based on criteria such as title, author, or publication year.
- Implement a search function in the *Library* class to locate a book by a keyword (e.g. title, author, or year).
- Design and implement a text-based menu to allow users to interact with the application for searching, viewing, and sorting data.
- Track user activities, such as searches and sorts, by implementing a logging mechanism for monitoring and debugging.
- Serialize the data structure to save the library state to a file, and deserialize it to restore the state when the program restarts.

Task 1 Review library file and read data from a file

Begin by familiarizing yourself with the structure of the project and the library files, focusing on how the data is organized within the *books.txt* file.

The search engine library files are located in the src/resources folder inside the src folder in the project root directory. This structure organizes the required none code project files into a clear and meaningful hierarchy:

- */project-root*:
    - The root directory of your project. This is the top-level folder containing all the project files and subdirectories.
- *src/resources*:

    - A directory dedicated to storing non-source code files that your application needs, such as data files and logs. This keeps these files separate from your Java source code, making the project more organized.
- *src/resources/data*:
    - *books.txt*: A file containing the initial data for your application. Each line represents a book with attributes: title, author, and publication year.
    - *library.ser*: A serialized file where the application's state (i.e. the list of books) is saved. This file is created or updated by the application and allows the program to resume from where it left off.
    - *user_interactions.log*: A log file that records user actions (such as searches and sorts) during the application's runtime. It helps in tracking what actions users have taken.
- *src/resources/test*:
    - *test_books.txt*: The file test_books.txt is an exact copy of books.txt and isn't used in the tutorial. Learners can use it for practice if they want, or simply ignore it.

The *loadBooks* method in the *Library* class is currently incomplete. Complete this method first to ensure the essential functionality for managing and displaying the book collection within the application.

🖥️ It's time to get coding!

- TODO 1: parse each line from the books.txt file to extract the book details (title, author, and publication year).
- TODO 2: create a *Book* object using the extracted details.
- TODO 3: add the *Book* object to the books list.

```java
public void loadBooks(String fileName) {

        try (BufferedReader br = new BufferedReader(new
FileReader(fileName))) {

            String line;

            while ((line = br.readLine()) != null) {


// TODO missing code


            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }
```

- TODO 4: update the main method in the *Main* class to load the books and display them.

```java
public class Main {

    public static void main(String[] args) {

        Library library = new Library();

        library.loadBooks("src/resources/data/books.txt");

        library.viewAllBooks();

    }

}
```

Pause and check

Great start! Before you continue, ensure everything is working as expected. Run your program by executing the *Main* class, which will load the books from the *books.txt* file and display them on the console.

Expected output

The desired output confirms that the *Library* class has successfully loaded the data from the *books.txt* file, created *Book* objects, and populated the books list accordingly.

```
Book{title='The Great Gatsby', author='F. Scott Fitzgerald',
publicationYear=1925}

Book{title='To Kill a Mockingbird', author='Harper Lee',
publicationYear=1960}

Book{title='1984', author='George Orwell', publicationYear=1949}
```

Task 2 : Complete and test the bubbleSort method

The *SortUtil* class provides various sorting algorithms for organizing a list of books by different criteria. While the insertion sort and quicksort algorithms have already been implemented, the next task is to complete the implementation of the bubble sort algorithm to sort a list of *Book* objects by different attributes.

To do this, you'll implement the bubble sort algorithm in the *SortUtil* class to sort the list of *Book* objects by a specified attribute, such as title, author, or publication year. (The method signature for the bubble sort is included in the source code.)

🖥️ It's time to get coding!

- TODO 5: calculate the list size. Inside the method, start by identifying the size of the books list.
- TODO 6: implement the nested loops. Use two nested loops to iterate over the list:
    1. the outer loop will manage the overall passes through the list.
    2. the inner loop will compare and potentially swap adjacent elements.

- TODO 7: compare and swap adjacent elements. Inside the inner loop, use the comparator to compare adjacent *Book* objects.
    1. if the current *Book* object should come after the next one according to the comparator, swap them using *Collections.swap*.

```
public static void bubbleSort(List<Book> books, Comparator<Book>
comparator)

{



}
```

- TODO 8: update the code of the main method in the *Main* class that loads the books, applies the sorting algorithm, and then displays the sorted list.

```
public class Main {

    public static void main(String[] args) {

        Library library = new Library();

        library.loadBooks("src/resources/data/books.txt");



        System.out.println("Before sorting:");

        library.viewAllBooks();



SortUtil.bubbleSort(library.getBooks(),
Comparator.comparing(Book::getTitle));



        System.out.println("After sorting by title:");
        library.viewAllBooks();

    }

}
```

Pause and check

To be sure of your work so far, run your program by executing the *Main* class. This class loads the books, displays them in their original order, sorts them by title, and then displays them again in their sorted order—amazing!

Expected output

The output should confirm that the *bubbleSort* method is functioning correctly, sorting the list of books by the specified attribute (in this case, the title).

```
Before sorting:

Book{title='The Great Gatsby', author='F. Scott Fitzgerald',
publicationYear=1925}

Book{title='To Kill a Mockingbird', author='Harper Lee',
publicationYear=1960}

Book{title='1984', author='George Orwell', publicationYear=1949}



After sorting by title:

Book{title='1984', author='George Orwell', publicationYear=1949}

Book{title='The Great Gatsby', author='F. Scott Fitzgerald',
publicationYear=1925}

Book{title='To Kill a Mockingbird', author='Harper Lee',
publicationYear=1960}
```

`Task : Implement a search function`

To enhance the user experience, your next task is to develop a search function within the *Library* class that lets users quickly locate a specific book by its title, author, or publication year. The search should be case-insensitive and return the details of the matching book if found, or an appropriate message if the book is not in the library.

This functionality will significantly improve the application's usability. The method signature has already been created for you in the Library class. It takes a String parameter and returns the matching Book object if found or null if not.

🖥️ It's time to get coding!

- TODO 9: ensure that the search is case-insensitive and convert the keyword to lowercase. This makes comparing the keyword with the title and author easier without worrying about case differences. You can use "*keyword = keyword.toLowerCase();*")
- TODO 10: use a *for-each* loop to iterate over the books list. This allows you to examine each *Book* object in the library one by one. As you iterate over each *Book* object, check if any of the following conditions are true and convert them if needed:
    - The lowercase title of the book contains the lowercase keyword.
    - The lowercase author name contains the lowercase keyword.
    - The publication year, converted to a string, exactly matches the keyword.
- TODO 11: check each condition within the loop.
    - If any of the conditions above are *true*, return the current *Book* object. (This immediately ends the method and provides the user with the first matching book.)
    - If the loop completes and no match is found, return *null* to indicate that there is no matching book in the library.

```
public Book searchBookByKeyword(String keyword) {

return null;
}
```

- TODO 12: update the *main* method to prompt the user for input and display the search results to test *searchBookByKeyword*.

```
public class Main {
    public static void main(String[] args) {
        Library library = new Library();
        library.loadBooks("src/resources/data/books.txt");

   Scanner scanner = new Scanner(System.in);
   System.out.print("Enter a keyword to search (title, author, or year): 
");
        String keyword = scanner.nextLine();

        Book foundBook = library.searchBookByKeyword(keyword);

        if (foundBook != null) {
            System.out.println("Book found: " + foundBook);
        } else {
            System.out.println("Book not found.");
        }
    }
}
```

Pause and check

Run your program by executing the *Main* class. This class will load the books from books.txt and prompt you to enter a keyword for searching. You can enter different keywords to search for books by title, author, or publication year. Be sure to test both successful and unsuccessful searches to ensure the search functionality is comprehensive.

Expected output

The output should confirm that the *searchBookByKeyword* method is functioning correctly, identifying matches based on the title, author, or year provided by the user.

```
Enter a keyword to search (title, author, or year): 1984
Book found: Book{title='1984', author='George Orwell', publicationYear=1949}


Enter a keyword to search (title, author, or year): J.K. Rowling
Book not found.
```

Task : Implement a text-based menu

You'll now design and implement a text-based menu that allows users to interact with the library. The users will appreciate the menu because it will enable them to perform various actions, such as viewing all books, sorting books by different criteria, searching for books, and exiting the program.

To achieve this, implement the *displayMenu* method in the *LibraryMenu* class to provide an intuitive interface for interacting with the library. This method presents the user with a series of options and handles their input accordingly.

🖥️ It's time to get coding!

- TODO 13: use a *while (true)* loop to keep the menu active until the user chooses to exit; the *displayMenu* method should continuously display a list of options to the user.
- TODO 14: present the menu options for the following:
  - Viewing all books.
  - Sorting books by title.
  - Sorting books by author.
  - Sorting books by publication year.
  - Searching for a book by keyword.
  - Exiting the program.
- TODO 15: use a *Scanner* to read the user's choice from the console and a switch statement to handle each menu option based on their input:

Option 1	Call the *viewAllBooks()* method to display all books.
Option 2	Call the *bubbleSort* method to sort books by keyword, then display the sorted list.
Option 3	Call the *insertionSort* method to sort books by author, then display the sorted list.
Option 4	Call the *quickSort* method to sort books by publication year, then display the sorted list.
Option 5	Call the *searchBookByKeyWord* method to search for a book by its title, author name or year of publication.
Option 6	Exit the menu and terminate the program.
Invalid Input	Print a mes

```
public void displayMenu() {
    Scanner scanner = new Scanner(System.in);
    while (true) {

    }
}
```

- TODO 16: update the code of the *main* method in the *Main* class to initialize the library, load the books, and invoke the *displayMenu* method to test it.

```
public class Main {
    public static void main(String[] args) {

        Library library = new Library();

        library.loadBooks("src/resources/data/books.txt");

        LibraryMenu menu = new LibraryMenu(library);

        menu.displayMenu();
    }
}
```

Pause and check

How's it going so far? To find out, run the program by executing the *Main* class. The program will load the books and display the main menu, allowing you to interact with various options. Use the menu to view all books, sort them by title, author, or year, search for specific books by keyword, and exit the program.

```
=== Main Menu ===


1. View All Books
2. Sort Books by Title
3. Sort Books by Author
4. Sort Books by Year
5. Search for a Book by keyword
6. Exit
```

Task 5 : Implement the log method

To track user activity, all interactions with the library – such as viewing books, sorting by different criteria, searching for books, and exiting the program – need to be logged and saved. This logging is crucial for auditing or debugging purposes.

To accomplish this, complete the log method in the *UserInteractionLogger* class. This method will record each user action, including the time and type of interaction (e.g. sorting, searching). The method signature is already in place. The log method will save these interactions to the *user_interactions.log* file with a timestamp, enabling you to track user activity for debugging or auditing.

🖥️ It's time to get coding!

- TODO 17: open the *user_interactions.log* file in append mode using a *FileWriter* to add new log entries without overwriting existing content.
- TODO 18: construct the log entry by combining the current timestamp from *LocalDateTime.now()* with the message passed to the log method.
- TODO 19: handle exceptions by surrounding the file writing operations with a try-catch block to catch any *IOException* that may occur.

```
 public void log(String message) {

}
```

- TODO 20: integrate logging into various actions in the *LibraryMenu* class to test it. The main method should invoke the *displayMenu* method, where each interaction with the menu will be logged.

```
public class Main {
    public static void main(String[] args) {

        Library library = new Library();
        library.loadBooks("src/resources/data/books.txt");

        LibraryMenu menu = new LibraryMenu(library);
        menu.displayMenu();

        UserInteractionLogger logger = new UserInteractionLogger();
        logger.log("Program started and menu displayed.");

        library.viewAllBooks();
        logger.log("Viewed all books.");

        SortUtil.bubbleSort(library.getBooks(),
Comparator.comparing(Book::getTitle));
        logger.log("Sorted books by title.");
    }
}
```

Pause and check

Great progress! Before continuing, make sure everything is working so far. Run your program and try the following:

1. Execute the *Main* class. The program should display the menu and simulate some actions that are logged.
2. Choose different options to view books, sort them, search for specific books, and exit the program.
3. After running the program, open the *user_interactions.log* file located in the src/resources/data directory. Verify that each interaction is logged with accurate timestamps and messages.

Expected output

After interacting with the program, the *user_interactions.log* file might contain entries like what follows, which confirms the log method is functioning correctly:

```
2023-08-21T10:35:21.123 - Program started and menu displayed.
2023-08-21T10:35:35.456 - Viewed all books.
2023-08-21T10:36:05.789 - Sorted books by title.
```

This confirms that the log method is functioning correctly, capturing user interactions with the library system.

Task : Serialize and deserialize objects

ou’re almost finished! In this final task, you’ll implement the saveLibrary and l*oadLibrary* methods in the *LibrarySerializer* class to enable saving and restoring the library's state across sessions. This ensures that any changes made by the user, such as sorting books in a specific order, are retained even after the program is closed.

Note
Serialization saves the current state to a file, while deserialization reloads it when the program restarts, providing a seamless user experience with data persistence and continuity.

🖥️ It’s time to get coding!

- TODO 21: implement the *saveLibrary* method by completing the missing code. Inside the *try* block, use the *writeObject* method to serialize the list of books and write it to the specified file.

```
public void saveLibrary(List<Book> books, String fileName) {
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(fileName))) {

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

- TODO 22: implement the *loadLibrary* method by completing the missing code. Inside the *try* block, use *ObjectInputStream* to read the list of books from the specified file. Ensure the deserialized object is cast to *List<Book>* before returning it.

```
public List<Book> loadLibrary(String fileName) {

        File file = new File(fileName);

        if (!file.exists() || file.length() == 0) {

            return null;

        }

        try

                (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(fileName))) {

        }

        catch (EOFException e) {

          System.err.println("The file is empty or corrupted: " +
fileName);

            return null;

        } catch (IOException | ClassNotFoundException e) {

            e.printStackTrace();

        }
```

```
            return null;

        }

    }
```

- TODO 23: add calls to the *saveLibrary* and *loadLibrary* methods in the appropriate places within the main method of the *Main* class. Specifically, load the library data when the program starts and save the library data when the program exits.

```
public class Main {

    public static void main(String[] args) {

        Library library = new Library();

        LibrarySerializer serializer = new LibrarySerializer();

        List<Book> books =
serializer.loadLibrary("src/resources/data/library.ser");

        if (books != null) {

            library.setBooks(books);

            System.out.println("Library loaded successfully from
src/resources/data/library.ser");

        } else {

           System.out.println("Loading data from books.txt...");

            library.loadBooks("src/resources/data/books.txt");
        }

        LibraryMenu menu = new LibraryMenu(library);

        menu.displayMenu();

        serializer.saveLibrary(library.getBooks(),
"src/resources/data/library.ser");

        System.out.println("Library saved successfully to
src/resources/data/library.ser");

    }

}
```

Pause and checkFinally, it's time to test the whole program! So far, you've implemented various functionalities like loading and saving the library, sorting books, and searching within the library. Now, test your efforts by following these steps:

- Execute the *Main* class. The program will either load the library from the serialized file or from the *books.txt* file if the serialized file doesn't exist.
- Use the menu to view books, sort them, and perform searches. Ensure all functionality works as expected.
- After interacting with the menu, exit the program. The library's state should be saved automatically.
- After the program exits, verify that the *library.ser* file is created or updated in the src/resources/data directory.
- Run the program again to ensure it loads the library state from the *library.ser* file.

Expected console output

As you test the program, the console output will help you verify that the program is functioning correctly. Below are examples of what you should see depending on how the library is loaded and saved.

If the library is loaded from the serialized file:

```
Library loaded successfully from src/resources/data/library.ser
```

If the library is loaded from *books.txt*:

When the library state is saved upon exiting:

```
Library saved successfully to src/resources/data/library.ser
```