2020

DSR ASSIGNMENT ONE

DOCUMENTATION AND TESTING

ETHAN FRASER | Z5260786

UML DIAGRAMS

Book

`- bookTitle: String`
`- bookAuthor: String`

+ Book(title: String, author: String): void + setTitle(title: String): void + setAuthor(author: String): void + getTitle(): String + getAuthor(): String + equals(obj: Object): Boolean + toString(): String

BooksCatalogue

`- bookCat: List<Book>`
`- readingHistory: List<Book>`

+ addBook(b: Book): void + removeBook(b: Book): void + removeByAuthor(author: String): void + searchByAuthor(author: String): List<Book> + getNumberOfBooks(author: String): int + searchByTitle(title: String): List<Book> + getNumberOfBooksTitle(title: String): int + addReadingBooks(b: Book): void + getReadingHistory(): List<Book> + sortByAuthorAscending(): List<Book> + sortByAuthorDescending(): List<Book> + sortByTitleAscending(): List<Book> + sortByTitleDescending(): List<Book>

BookWithISBN extends Book
`- ISBN: int`
+ Book(title: String, author: String, ISBN int): void + getISBN(): int + equals(obj: Object): Boolean

TitleSortA
+ compare(bookA: Book, bookB: Book)

TitleSortD
+ compare(bookA: Book, bookB: Book)

BooksCatalogueISBN
`- books: Map<Integer, BookWithISBN>`
+ addBook(book: BookWithISBN): void + removeBook(book: BookWithISBN): void + gteBook(ISBN: int): BookWithISBN

AuthorSortA
+ compare(bookA: Book, bookB: Book)

AuthorSortD
+ compare(bookA: Book, bookB: Book)

TASK 2

DESIGN CONSIDERATIONS

When designing and creating the BooksCatalogue class, very few issues arose and new significant design decisions had to be changed. By using a List<> object the program can easily add, remove and search within the object for the items. When testing this class and method, it was important to test if it would accept the same book twice and then a different book with the same details. It was important to make sure that the removal methods were prepared for a scenario where the book they were trying to remove did not exist. These were simple factors to consider when designing the methods and the results of this is displayed in the testing segment below.

TASK 3

DESIGN CONSIDERATIONS

When designing the searching methods for this task it was important that it centred around two important factors. The method had to return with a list containing the books relevant to the search and that the search is case insensitive and can retrieve partial matches. This meant that, for example, when searching the method must return the same when searching with the first two letters of the author name in lower case as searching with the full author name in capital letters. Once this consideration was implemented the only other part of the search to implement was adding the successful searches to the List object for the function to return.

By using this process, it also simplified the process of returning the count of the authors or titles returned in the search. This is due to the ability to call on methods dynamically within a class as to return the count of authors, the counting method simply runs the search method and returns the size of that list. This allows the program to have less redundant code and simplify the reading and understanding of the code for other developers.

TASK 4

DESIGN CONSIDERATIONS

For this task it was important to consider different options of storing objects and decide which option was the most effective for this task. I chose to use a `List<Book>` as previously used for the book catalogue. The reason I chose to use this option was because as it is the object used for the book catalogue, I felt it would be simpler as a developer to maintain the consistency. Another reason I used it was because it meant I had the capability to add books to the very start of the list but remove them from anywhere inside the list. This allowed me to effectively meet the task criteria which required any new book added to be added to the start of the book and for that book to be removed if it already exists.

TASK 5

DESIGN CONSIDERATIONS

When approaching this task, I felt I had two options, both using a similar process but with different efficiency. The first option was using a bubble sort process that utilises the `compare()` method when iterating through the catalogue and checking the alphabetic difference between books, moving them forward and keeping them in the same spot based on the result from the `compare()` method. The second option, which is the option I elected to use, was to create a comparator that would do the sorting inside of the inbuilt `sort()` method.

I chose the comparator process because it was a cleaner option to the numerous for loops that would be required for each method in the sorting process and because it utilised an inbuilt method in the Java Collections classes. This was a much more efficient process of sorting the lists and simply required new comparators to be made which could be attached to any of the classes, as show in the Book.JAVA file.

TASK 6

DESIGN CONSIDERATIONS

When designing the classes and methods for this task it was useful to remember and use the other classes that are already designed. For the `BookWithISBN` class, after extending the Book class all that it needed was to add in the ISBN variable that the Book class doesn't have. The only other significant changes to the `BookWithISBN` class that the Book class does not have is the ability to get a book ISBN and a new `equals()` method that will be able to check the value of the ISBN as well.

Making a book catalogue with an ISBN meant that we now had a more effective method of differentiating whether books are already in the catalogue or not because the ISBN is used as the key in the HashMap object for each book. This is reflected in the process of checking the existence of a book in each of the catalogue methods.

TASK 7

REFLECTION

I found that designing the UML diagrams and more importantly, the tests for each of the methods, was an excellent first step for designing the code. With the tests as the first step in the design process, I had an outline for each method requirements and limitations that helped to know what I needed to code. The tests allowed me to dynamically update my code before I was finished the assignment because I was able to see where my problems may appear.

Through testing and adjusting, it became clear just how integral effective testing is to successful programming as with only a small portion of test cases, you may not be able to identify all possible faults in a program. When testing and adjusting a bubble sort method for sorting authors, my code would not successfully sort a catalogue of more than two books. Through an extensive series of tests this was identified and corrected. This is a fault that, in a large-scale program for a library or bookshop would have been a critical fault in their system.

One of the most significant constraints I found with designing and testing was that as there is no requirement for user input or an interface there is total control of input. For an interface or any user input testing I would also do type testing to establish the type of data that is entered. To try do this testing straight from the JAVA classes the compiler will run an error before it can run. This means that the code cannot run in anyway and cannot be tested for these scenarios as the classes cannot compile if the function has the wrong data type entered.

TESTING

TASK 1 TEST

SERIAL	DESCRIPTION	INPUT	EXPECTED OUTPUT	ACTUAL OUTPUT
1	Empty title	`“b = new Book(“”, “a”);`	“Error: Invalid input - Provide a book title”	“Error: Invalid input - Provide a book title”
2	Null title	“b = new Book(null, “a”);		
3	Empty author	“b = new Book(“title”, “”);”	“Error: Invalid input, author must not be empty and only contain characters - a-z and A-Z characters only”	“Error: Invalid input, author must not be empty and only contain characters - a-z and A-Z characters only”
4	Null author	“b = new Book(“title”, null);”		
5	Digit input author	“b = new Book(“title”, “tree5”);”		
6	Punctuation input author	“b = new Book(“title”, “tree!”);”		
7	Standard input book	“b = new Book(“Title”, “Author”);”	“Successfully added – ‘Title’ by Author”	“Successfully added – ‘Title’ by Author”
8	Retrieve author	“System.out.println(b.getAuthor());”	“Author”	“Author”
9	Retrieve title	“System.out.println(b.getTitle());”	“Title”	“Title”
10	Printing string of book	“System.out.println(b.toString());”	“‘Title’ by Author”	“‘Title’ by Author”

TASK 2 TEST

Objects used:

```
BooksCatalogue books = new BooksCatalogue();
Book b1 = new Book(“Title A”, “Author A”);
Book b2 = new Book(“Title B”, “Author B”);
Book b3 = new Book(“Title C”, “Author A”);
Book b4 = new Book(“Title A”, “Author A”);
```

SERIAL	DESCRIPTION	INPUT	EXPECTED OUTPUT	ACTUAL OUTPUT
1	Add in book	`books.add(b1);`	“Added item – [BOOK DETAILS]”	“Error: Invalid input - Provide a book title”
2	Add in different book	`“books.add(b2);”`		
3	Add in same book	`“books.add(b1);”`	"ERROR: Book already exists in catalogue."	"ERROR: Book already exists in catalogue."
4	Add in different	`“books.add(b4);”`		

	book, same details			
5	Remove book	`“books.remove(b1);”`	“Removed item – [BOOK DETAIL]	“Removed item – [BOOK DETAIL]
6	Remove same book	`“books.remove(b1);”`	"ERROR: Book already exists in catalogue."	"ERROR: Book already exists in catalogue."
7	Remove by Author (incorrect author)	`“books.removeByAuthor(“Steven”)`	“No books by Steven found”	“No books by Steven found”
8	Remove by author	`“books.removeByAuthor(“Author A”)`	“Removed 2 books by Author A”	“Removed 2 books by Author A”

TASK 3 TEST

Objects used:

```
Book b1 = new Book(“Title A”, “Author A”);
Book b2 = new Book(“Title B”, “Author B”);
Book b3 = new Book(“Title C”, “Author A”);
BooksCatalogue books = new BooksCatalogue();
Books.add(b1); books.add(b2); books.add(b3);
```

SERIAL	**DESCRIPTION**	**INPUT**	**EXPECTED OUTPUT**	**ACTUAL OUTPUT**
1	Search by author (Incorrect)	`“books.searchByAuthor(“Steven”);”`	“[]”	[]
2	Search by author (All caps)	`“books.searchByAuthor(“AUTHOR A”);”`	“[“Title A” by Author A, “Title C” by Author A]”	“[“Title A” by Author A, “Title C” by Author A]”
3	Search by author (Partial)	`“books.searchByAuthor(“Auth”);”`	“[“Title A” by Author A, “Title B” by Author B, “Title C” by Author A]”	“[“Title A” by Author A, “Title B” by Author B, “Title C” by Author A]”
4	Get number of books by author	`“books.getNumberOfBooks(“Auth”);”`	“3”	“3”
5	Search by title	`“books.searchByTitle(“Title B”);”`	“[“Title B” by Author B]”	“[“Title B” by Author B]”

6	Get number of books by title	`"books.getNumberOfBooksTitle("Title B");"`	"1"	"1"

TASK 4 TEST

Objects used:

```
Book b1 = new Book("Title A", "Author A");
Book b2 = new Book("Title B", "Author B");
Book b3 = new Book("Title C", "Author A");
BooksCatalogue books = new BooksCatalogue();
Books.add(b1); books.add(b2); books.add(b3);
```

SERIAL	**DESCRIPTION**	**INPUT**	**EXPECTED OUTPUT**	**ACTUAL OUTPUT**
1	Add reading history	`"books.addReadingHistory(b1);"`	"Added "Title A" by Author A to reading history"	"Added "Title A" by Author A to reading history"
2	Adding book to history again (after adding b2)	`"books.addReadingHistory(b1);"`	"Added "Title A" by Author A to reading history"	"Added "Title A" by Author A to reading history"
3	Getting reading history after changes	`"books.getReadingHistory();"`	"["Title A" by Author A, "Title B" by Author B]"	"["Title A" by Author A, "Title B" by Author B]"

TASK 5 TEST

Objects used:

```
Book b1 = new Book("Title A", "Author C");
Book b2 = new Book("Title B", "Author B");
Book b3 = new Book("Title C", "Author A");
BooksCatalogue books = new BooksCatalogue();
Books.add(b1); books.add(b2); books.add(b3);
```

SERIAL	**DESCRIPTION**	**INPUT**	**EXPECTED OUTPUT**	**ACTUAL OUTPUT**
1	Sort author ascending	`"books.sortByAuthorAscending();"`	"["Title C" by Author A, "Title B" by Author B,	"["Title C" by Author A, "Title B" by Author B,

			"Title A" by Author C]"	"Title A" by Author C]"
2	Sort by author descending	`"books.sortByAuthorDescending();"`	"["Title A" by Author C, "Title B" by Author B, "["Title C" by Author A]"	"["Title A" by Author C, "Title B" by Author B, "["Title C" by Author A]"
3	Sort by title ascending	`"books.sortByTitleAscending();"`	"["Title A" by Author C, "Title B" by Author B, "["Title C" by Author A]"	"["Title A" by Author C, "Title B" by Author B, "["Title C" by Author A]"
4	Sort by title descending	`"books.sortByTitleDescending();"`	"["Title C" by Author A, "Title B" by Author B, "Title A" by Author C]"	"["Title C" by Author A, "Title B" by Author B, "Title A" by Author C]"

TASK 6 TEST

Objects used:

```
BooksCatalogueISBN books = new BooksCatalogueISBN();
```

SERIAL	DESCRIPTION	INPUT	EXPECTED OUTPUT	ACTUAL OUTPUT
1	Null ISBN	`"BookWithISBN b1 = new BookWithISBN("Title", "Author", null);"`	"ERROR: ISBN cannot be null"	"ERROR: ISBN cannot be null"
2	Valid ISBN	`"BookWithISBN b1 = new BookWithISBN("Title", "Author", 1);"`	"Added book – "Title" by Author"	"Added book – "Title" by Author"
3	Get ISBN	`"System.out.println(b1.getISBN());"`	"1"	"1"
4	Add book	`"books.add(b1);"`	"Added item – [BOOK DETAILS]"	"Added item – [BOOK DETAILS]"
5	Add book that already exists	`"books.add(b1);"`	"ERROR: Book already exists in catalogue."	"ERROR: Book already exists in catalogue."
6	Get book	`"books.getBook(1);"`	""Title" by Author"	""Title" by Author"

7	Get book not in catalogue	`“books.getBook(4);”`	"ERROR: This book does not exist"	"ERROR: This book does not exist"
8	Remove book	`“books.remove(b1);”`	“Removed item – [BOOK DETAILS]”	“Removed item – [BOOK DETAILS]”
9	Remove book not in catalogue	`“books.remove(b1);”`	“ERROR: Book does not exist in catalogue.”	“ERROR: Book does not exist in catalogue.”