Inventory Manager

Chris Hughes

December 6, 2023

1 Setup

Just run `javac InventoryManager.java` and `java InventoryManager`.

2 Storage

Currently there is no way to persistently store the books, orders, customers etc. that you create. The most rational way to do this would be to implement a database. Alternatively you could also implement `save()` and `retrieve()` functions that could save entries to the file system, perhaps in JSON format, and then retrieve them when you launch the program. This would work on a small scale, but would be difficult to maintain and inefficient at any larger scale.

3 Challenges

I initially had a lot of trouble getting the `ArrayList` to function how I wanted it to. My problem was that I did not create the sub classes correctly with the class constructor Once I figured this out it went a lot more smoothly. The other major issue I had was a lot of fiddling around with `nextInt` `nextLine`, `nextDouble` and so on. This is why I created `Input` class and, following a friends suggestion, used `nextLine` for all user inputs and then used `Integer.parseInt` and `Double.parseDouble` to get integers and doubles from the user. It also made validation and the program flow much simpler.

4 Program Structure

Class `InventoryManager`

This class is responsible for handling the overall flow of the program and creates instances of the other 3 main classes. It contains 4 functions, `main()`, which is where the `Books`, `Orders` and `Input` classes are instantiated and contains the main switch. The main switch is controlled by user input and sends users to the other 3 functions, `create()`, `read()`, `update()` and `destroy()`. I based these functions off of the "CRUD", (Create, Read, Update, Destroy) paradigm and they direct the flow of the program to functions in the relevant classes. The functions return strings, which are then printed to the console.

For example,

```
static void create(Input in, Books books, Orders orders){
    switch (in.getChar("> ")) {
    case 'b':
        System.out.println(books.input(in));
        break;
    case 'o':
        System.out.println(orders.input(in, books));
```

```
            break;
        }
    }
```

Class `Books`

The Books class contains an instance of `ArrayList` that stores the books, functions for manipulating and retrieving data from the `ArrayList`, and a Subclass `Book`, which represents a single instance of a book.

Subclass `Book`

This class contains constructor function that creates an instance of `Book` for storage in the `ArrayList`.

Function `input` returns `String`

This function takes user input for the book details, checks to make sure the inputs are valid by testing if `Input.validation` is true, adds the book to the `ArrayList` and then prints a relevant message. If the book creation fails due to a bad input it sets `Input.validation` back to true to prepare for the next input.

Function `list`

This simple functions uses a `for` loop to loop through the `ArrayList` and print the contents.

Function `getBookPrice` returns `double`

This function calls the `getBook` function to retrieve a book by its id and returns its price as a `double`

Function `updateStock` returns `Boolean`

Calls the `getBook` function to retrieve the book and then tests to see if the `quantity` parameter is equal to or less than `Book.quantity`. If so, it deducts from `Book.quantity` prints the remaining stock. Finally, if there is stock it returns `true` and if there isn't sufficient stock it returns `false`

```
if (quantity <= b.quantity){
    b.quantity -= quantity;
    isStock = true;
    System.out.println("\n**************************\n");
    System.out.println("Remaining stock: " + b.quantity);
}
```

Function `update` returns `String`

This functions gets user input for a switch to determine what element of `Book` the user would like to update and then asks for user input to update it. It returns a `String` depending on whether the update is successful or not.

Function `getBook` returns `Book`

Uses a `for` loop to iterate through `Books` to find a book based on it's id. It either returns the `Book` if it matches the id or `null` if not.

```
for (Book i: books) {
    if (id == i.book_id) {
        System.out.println(i.title);
        return i;
    }
}
return null;
```

Function `destroy` returns `String`

Calls `getBook` to retrieve a book based on the id input by the user and then prints the book details and asks for confirmation. On confirmation, it removes the `Book` from `Books`.

Class `Orders`

The orders class is very similar in structure to the books class. it has a Subclass `Order`, which is similar to the `Book` subclass. The `input` function contains some login to ensure that there is sufficient stock to order a book calling `updateStock` from the `Books` class. The `list`, `update`, and `destroy` functions are virtually the same as those in `Books`.

Function `sales` returns `String[]`

This function contains the logic for the sales report feature. in creates an `int quantity` and `double totalSales` to store the sales report data. It loops over `Orders` and increments `quantity` by the value in each `Order` and then calls `getBookPrice` from `Books` and multiplies by `quantity` to generate total sales. Finally it uses `toString` to make a `String[]` which it returns.

```
int quantity = 0;
double totalSales = 0.0;

for (Order i: orders) {
    quantity += i.quantity;
    totalSales += books.getBookPrice(i.book_id) * i.quantity;
}

String[] sales = {Integer.toString(quantity), Double.toString(totalSales)};
return sales;
```

Class `Input`

I created this class to simplify user inputs though-out the code. It contains four similar functions which check to see if `Boolean validation` is true, and then uses a try/catch statement to attempt to get the user input. if it succeeds , it returns the relevant user input and if it fails, it sets `validation` to false and prints a relevant message for the user and/or the error. This is also where I create the main instance of the `Scanner` object.