Applications Programming

Lab - MVC (worth 8%)

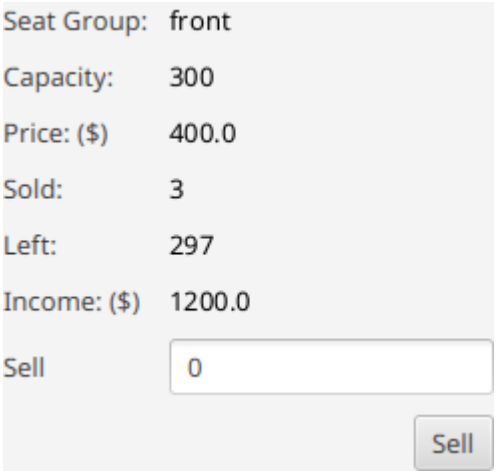
The process:

1. Create the application class.
2. Create the view in FXML.
3. Create the controller in Java.
4. Modify the model to support JavaFX properties.
5. Implement the event handlers.

Tutor demo

Main class: `StadiumApplication`

Make an application that sells seats in a stadium. The seats are divided into different groups and each seat group has a different price and capacity. In this simple application, the stadium has only one group of "front" seats. There are 300 front seats with a price of $400 per seat. The user interface is shown below:



The user enters a number into the Sell TextField and presses "Sell". The figures are updated to reflect the sale, and the Sell TextField is reset to zero.

Your tutor will code the solution.

Note: Your tutor may demonstrate bindings with code like this:

```
income.bind(sold.multiply(price));
```

Another way to create bindings is through the Bindings class:

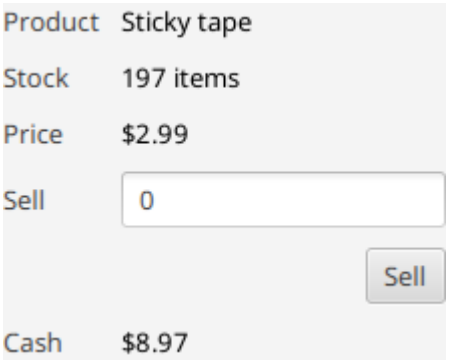
```
income.bind(Bindings.multiply(sold, price));
```

Read the documentation for the Bindings class here.

Student Specification

Main class: `StoreApplication`

Make a graphical user interface for the program you developed in the earlier classes lab: a store. The store has one product and a cash register. The product is "Sticky Tape". Initially there are 200 items of this product in stock, and they sell for $2.99 each. The user interface is as follows:



The user can input the number of items to sell. Clicking the "Sell" button will sell that number of items and reset the Sell TextField to zero. All figures are updated to reflect the sale. The stock is shown as "XYZ items" where XYZ is replaced by the stock property from the model. The price is displayed with a dollar sign to two decimal places, and so is the cash in the cash register.

Step 1. Open the lecture notes. You will need them as a reference. You may also refer to the tutor demo (downloadable from ED) and the lecture demo (downloadable from Canvas/Modules/ Subject Documents).

Step 2. Create a new JavaFX project in NetBeans IDE called "Lab9_2021AUT". If you have a preferred IDE, you may use it instead, as long as it supports Java 8 and allows projects to include non-Java files such as XML, CSS and PNG files.

NOTE: NetBeans will offer to automatically create a main class for you. Do not accept the offer since you will create the classes later by hand. But if you do accept the offer, please note:

1. The suggested class name should be changed to `StoreApplication`
2. The class should **NOT** be placed inside the suggested package, so you will need to delete the package name.

Step 3. Copy and paste the following code as a template for your StoreApplication class:

```
public class StoreApplication extends Application {
    public static void main(String[] args) { launch(args); }

    @Override
    public void start(Stage stage) throws Exception {
        FXMLLoader loader = new FXMLLoader(getClass().getResource("store.fxml"));

        // Add code here to load the root node from the FXML file
        // and show it
    }
}
```

Don't forget to import the required classes.

Step 4. Create an fxml file called "store.fxml" in the same directory as your StoreApplication class. Add code to this file to define the layout exactly as shown in the screenshot above. Note: everything in the left column is a label. The right column consists of Text nodes for Product, Stock, Price and Cash and a TextField for Sell. The Sell button is aligned right.

Step 5. Define a controller class called `StoreController` and declare the name of this controller class in your `store.fxml` file to link it to this controller. Use the following code as a template for your StoreController class:

```
public class StoreController {
    @FXML private Button sellBtn;
    @FXML private Text stockTxt;
    @FXML private Text priceTxt;
```

```
    @FXML private TextField amountTf;
    @FXML private Text cashTxt;
}
```

For each of these @FXML fields, modify the corresponding XML definition for this node in the `store.fxml` file to include an `fx:id` attribute that links it to the corresponding field above. Note that `amountTf` is the text field appearing after the Sell label.

Run your application to make sure it still works. Fix any errors reported.

Step 6. Create the model classes for Store, Product and CashRegister. You can use the solutions from last week's lecture which are included below.

Copy the following code into a new interface called `ProductObserver`

```
public interface ProductObserver {
    void handleSale(double amount);
}
```

Copy the following code into a new class called `CashRegister`

```
public class CashRegister implements ProductObserver {
    private double cash;

    public CashRegister() {
        cash = 0.0;
    }

    public void add(double money) {
        cash = cash + money;
    }

    @Override
    public void handleSale(double amount) {
        add(amount);
    }
}
```

Copy the following code into a new class called `Product`

```
public class Product {
    private LinkedList<ProductObserver> observers = new LinkedList<ProductObserver>();

    private String name;
    private int stock;
    private double price;

    public Product(String name, int stock, double price) {
        this.name = name;
        this.stock = stock;
        this.price = price;
    }

    public void sell(int n) {
        stock = stock - n;
        double money = n * price;
        for (ProductObserver observer : observers)
            observer.handleSale(money);
    }

    public void restock(int n) {
        stock = stock + n;
    }
```

```
    public boolean has(int n) {
        return stock >= n;
    }

    public void addProductObserver(ProductObserver observer) {
        observers.add(observer);
    }

    @Override
    public String toString() {
        return stock + " " + name + " at $" + price;
    }
}
```

Copy the following code into a new class called `Store`:

```
public class Store {
    private CashRegister cashRegister;
    private Product product;

    public Store() {
        cashRegister = new CashRegister();
        product = new Product("Sticky tape", 200, 2.99);
        product.addProductObserver(cashRegister);
    }
}
```

Step 7. Modify the controller and your model classes to expose JavaFX properties according to the JavaFX property patterns. Refer to the 4 patterns from the lecture:

1. Pattern #1: Immutable property
2. Pattern #2: Read Write property
3. Pattern #3: Read Only property
4. Pattern #4: Immutable property with mutable state

For example, the name of a product **never changes** so it is an immutable property. It looks like this:

```
public class Product {
    private String name;
    ...
    public final String getName() { return name; }
}
```

Step 8. Now bind each Text node in your view to the model:

1. For the Product name, use an FXML property binding expression, i.e. ${.....}.

   NOTE! Some students have reported that FXML binding expressions don't work on a Mac. If that is the case, you have permission to do the bindings in Java code rather than FXML code.

   For example, the lecture demo example (which you can download from Canvas / Modules /Subject Documents), you saw this code:

   ```
   nameTf.textProperty().bindBidirectional(customer.getAccount().nameProperty());
   ```

   However, in this case, you want to do a unidirectional binding:

   ```
   nameTf.textProperty().bind(customer.getAccount().nameProperty());
   ```

Replace `nameTf` by the node in your scene representing the product name. Because you want to refer to this from your Java code, you'll need to inject the node into your controller:

```
@FXML private Text nameTxt;
```

You should also modify `customer.getAccount().nameProperty()` to refer to the name property of the product.

2. For the Stock, do a binding in Java code so that the string " items" appears after the stock number (see the lecture notes for an example)
3. For the Price and Cash, do a binding in Java code so that the price is displayed as currency with a dollar sign and 2 decimal places.

Run your application to verify that all of the current product data is displayed.

Step 9. Set the initial contents of amountTf to just the number 0.

Step 10. In FXML, add an attribute on your Button node to specify the name of a Java method to handle the button click. Name this method `handleSell`. Write a corresponding method in your controller class that is linked to this action (see the lecture notes for an example). In this method, write code to perform the sale, but ONLY if there is enough stock. After performing the sale, set the contents of amountTf back to 0. Test your application to ensure that the button works. When you click it, the sale happens, and the data is automatically updated in the view.

Step 11. This is an assessed lab. Copy and paste the source codes and fxml codes to ED.

Step 12. Peer marking submission on Canvas. The instruction is specified here.

NOTE: After you submit the source files to ED, you will not receive your mark immediately. It is not possible for ED to mark a GUI automatically since a user is required to use a mouse to drive the program. Therefore, this lab will be manually marked by peer marking. You should submit a JAR file to Canvas by the due date to participate the peer marking. Late submission will be excluded from peer marking activities and result 50% mark deduction.

Marking Scheme

All leaf nodes are shown	10%
All nodes are laid out correctly in a grid	10%
The Sell button is correctly aligned right	5%
The product name, stock, price and cash values are shown	20%
The stock, price and cash values are formatted correctly	20%
After clicking Sell, the stock is correctly updated in the view	10%
After clicking Sell, the cash is correctly updated in the view	10%
After clicking Sell, the sell amount is reset to 0	10%
Clicking Sell does nothing if there is not enough stock	5%