



UNIVERSITI  
TEKNOLOGI  
PETRONAS

## **OBJECT ORIENTED PROGRAMMING**

**MAY 2024 SEMESTER**

**Project Title:** Marketplace for (UTP) Students

**GROUP:** 19

**LECTURER:** Dr Nordin Zakaria

<b>NAME</b>	<b>ID NUMBER</b>	<b>COURSE</b>
Abdullah Fouzi Saleh Ahmed Naji	22012364	IT
Mohammed Fisal Hassan	22009298	COE
Mohammed Adiyani Alvi	2000161	CS

## **1. Introduction**

In today's digital age, the convenience of online marketplaces has become indispensable. Students at Universiti Teknologi PETRONAS (UTP) frequently need a platform to buy and sell items such as textbooks, electronics, furniture, and other essentials. The proposed project aims to develop a Virtual Marketplace for UTP students, leveraging a client-server architecture to provide a secure, efficient, and user-friendly platform tailored to their needs. This project will be implemented using Java as the primary programming language, with JavaFX utilized for creating an intuitive and interactive graphical user interface (GUI).

### **Project Description**

The primary objective of this project is to create a virtual marketplace platform exclusively for UTP students. This platform will facilitate the buying, selling, and trading of various items, thereby fostering a community-driven approach to student needs and reducing reliance on external marketplaces.

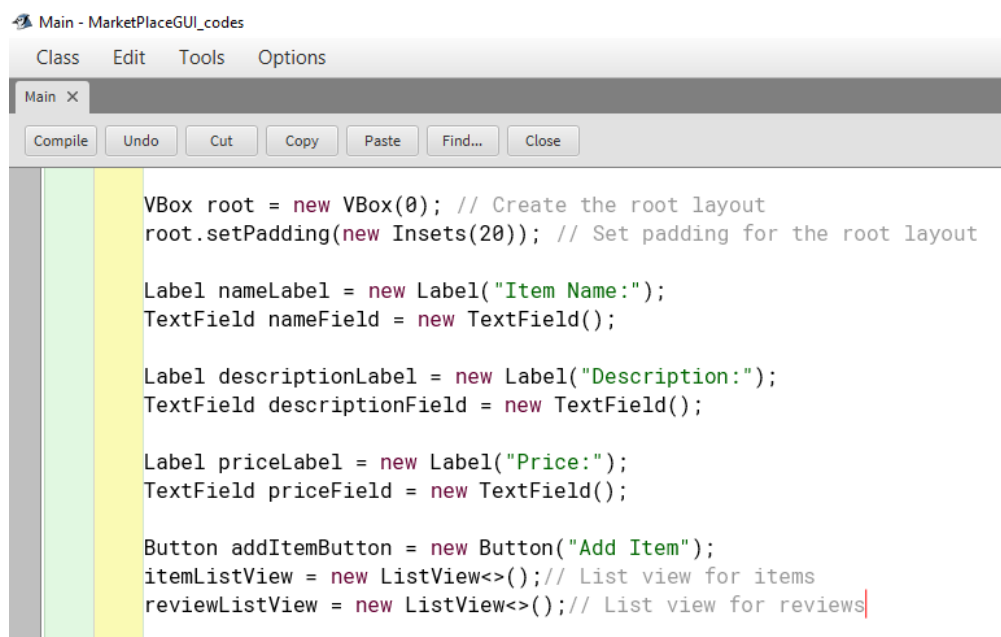
## Main Class Functionality

- **start**: Initializes the primary stage and sets up the user interface components, including forms for adding items and reviews, and list views for displaying items and reviews.
- **showAlert**: Displays alert messages for errors, warnings, and information.
- **updateItemListView**: Refreshes the list of items displayed in the ListView.
- **updateReviewListView**: Refreshes the list of reviews displayed in the ListView for a selected item.
- **findItemByName**: Searches for an item by its name in the items list.

The main class serves as the entry point of the application, orchestrating the interaction between various components and providing a graphical interface for the user to interact with the virtual marketplace.

### 1. Initialization of the Stage and UI Components:

Describe how the main stage and its user interface components are set up. Explain the purpose of each UI element and how they are arranged in the layout.



```

Main - MarketplaceGUI_codes
Class Edit Tools Options
Main X
Compile Undo Cut Copy Paste Find... Close

VBox root = new VBox(0); // Create the root layout
root.setPadding(new Insets(20)); // Set padding for the root layout

Label nameLabel = new Label("Item Name:");
TextField nameField = new TextField();

Label descriptionLabel = new Label("Description:");
TextField descriptionField = new TextField();

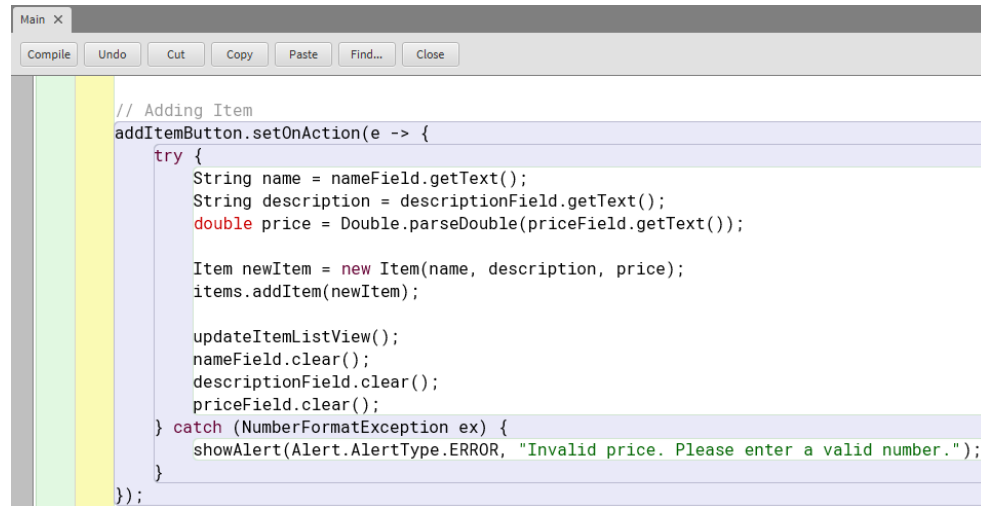
Label priceLabel = new Label("Price:");
TextField priceField = new TextField();

Button addItemButton = new Button("Add Item");
itemListView = new ListView<>(); // List view for items
reviewListView = new ListView<>(); // List view for reviews

```

## 2. Event Handling for Adding Items:

This section shows how the `addItemButton` is set up to handle the addition of items to the virtual marketplace. The event handler for the button click is defined using a lambda expression.

A screenshot of an IDE window titled 'Main'. The code editor shows a lambda expression assigned to `addItemButton.setOnAction`. The lambda body contains a `try` block that retrieves text from `nameField`, `descriptionField`, and `priceField`, creates a new `Item` object, and adds it to the `items` list. It then updates the `itemListView` and clears the input fields. A `catch` block handles `NumberFormatException` by displaying an alert with the message 'Invalid price. Please enter a valid number.'.

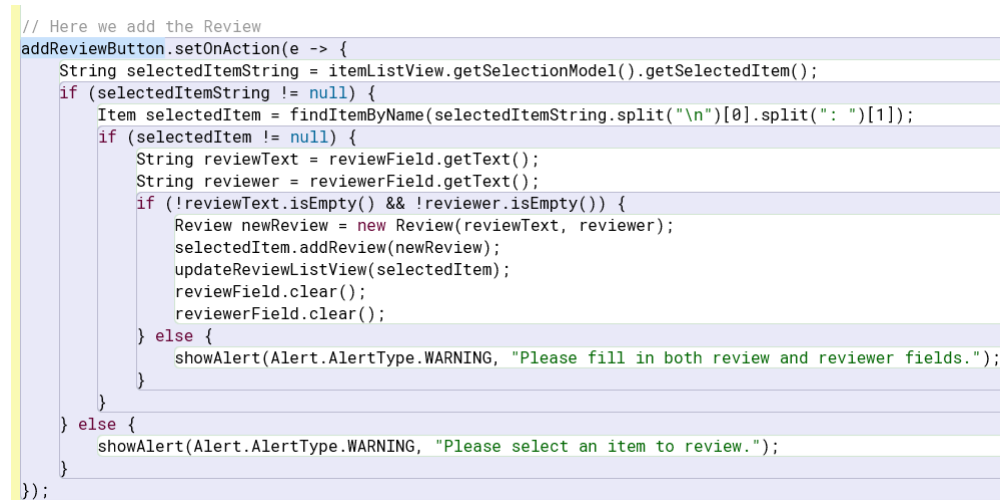
```
// Adding Item
addItemButton.setOnAction(e -> {
    try {
        String name = nameField.getText();
        String description = descriptionField.getText();
        double price = Double.parseDouble(priceField.getText());

        Item newItem = new Item(name, description, price);
        items.addItem(newItem);

        updateItemListView();
        nameField.clear();
        descriptionField.clear();
        priceField.clear();
    } catch (NumberFormatException ex) {
        showAlert(Alert.AlertType.ERROR, "Invalid price. Please enter a valid number.");
    }
});
```

## 3. Event Handling for Adding Reviews:

Show how the `addReviewButton` handles adding reviews to the selected item.

A screenshot of an IDE window showing the event handler for `addReviewButton`. The lambda expression first checks if an item is selected. If not, it shows a warning alert. If an item is selected, it retrieves the review text and reviewer from their respective fields. It then checks if both fields are non-empty. If so, it creates a new `Review` object, adds it to the selected item's reviews, updates the `reviewListView`, and clears the input fields. Otherwise, it shows a warning alert with the message 'Please fill in both review and reviewer fields.'.

```
// Here we add the Review
addReviewButton.setOnAction(e -> {
    String selectedItemString = itemListView.getSelectionModel().getSelectedItem();
    if (selectedItemString != null) {
        Item selectedItem = findItemByName(selectedItemString.split("\n")[0].split(": ")[1]);
        if (selectedItem != null) {
            String reviewText = reviewField.getText();
            String reviewer = reviewerField.getText();
            if (!reviewText.isEmpty() && !reviewer.isEmpty()) {
                Review newReview = new Review(reviewText, reviewer);
                selectedItem.addReview(newReview);
                updateReviewListView(selectedItem);
                reviewField.clear();
                reviewerField.clear();
            } else {
                showAlert(Alert.AlertType.WARNING, "Please fill in both review and reviewer fields.");
            }
        }
    } else {
        showAlert(Alert.AlertType.WARNING, "Please select an item to review.");
    }
});
```

#### 4. Creating a Secondary Stage for Sellers:

This snippet demonstrates how a secondary stage is created to display sellers in your application. A secondary stage allows you to create a new window that operates independently from the main stage, which is useful for displaying additional information such as a list of sellers.

```
// Create and show Mohammed Faisal window
Stage mohammedFaisalStage = new Stage();
mohammedFaisalStage.setTitle("Mohammed Faisal(22009298)");

VBox mohammedFaisalRoot = new VBox(10);
mohammedFaisalRoot.setPadding(new Insets(20));

ListView<String> sellerListView = new ListView<>();
for (Seller seller : sellers.getSellers()) { // Here is where the issue is
    sellerListView.getItems().add(seller.getName() + " - Rating: " + seller.getRating());
}

mohammedFaisalRoot.getChildren().addAll(
    new Label("Sellers List:"),
    sellerListView
);

Scene mohammedFaisalScene = new Scene(mohammedFaisalRoot, 300, 400);
mohammedFaisalStage.setScene(mohammedFaisalScene);
mohammedFaisalStage.show();

// Set main scene and show primary stage
Scene scene = new Scene(root, 400, 600);
primaryStage.setScene(scene);
primaryStage.show();
}
```

## 5. Updating List Views:

This section demonstrates how the list views are updated in your application. Specifically, the `updateItemListView` and `updateReviewListView` methods refresh the displayed items and reviews respectively. Screenshots of these methods can help illustrate the process.

```
private void updateItemListView() {
    itemListView.getItems().clear();
    for (Item item : items.getItemList()) {
        itemListView.getItems().add(item.toString());
    }
}

private void updateReviewListView(Item item) {
    reviewListView.getItems().clear();
    for (Review review : item.getReviews()) {
        reviewListView.getItems().add(review.toString());
    }
}

private Item findItemByName(String name) {
    for (Item item : items.getItemList()) {
        if (item.getName().equals(name)) {
            return item;
        }
    }
    return null;
}

public static void main(String[] args) {
    launch(args);
}
```

### Item Class:

The Item class represents an item in the virtual marketplace, containing attributes such as name, description, price, reviews, and ratings. This class provides methods to add reviews and ratings, retrieve information about the item, and calculate the average rating.

- Constructor:

Initializes an Item object with the specified name, description, and price.

```
public Item(String name, String description, double price) {
    this.name = name;
    this.description = description;
    this.price = price;
}
```

- Methods

- Getter Methods:

The Item class includes several getter methods that facilitate access to its private attributes, ensuring encapsulation and data protection. The getName method returns the name of the item, allowing other classes to access this private field without directly manipulating it. Similarly, the getDescription method provides access to the item's description, enabling its use in various parts of the application where the item's details need to be displayed or processed. The getPrice method retrieves the price of the item, which is crucial for display purposes and any calculations involving the item's cost. Furthermore, the getReviews method returns a list of reviews associated with the item. This method is essential for operations that require iterating through or displaying the item's reviews, thereby promoting a comprehensive understanding of user feedback. Lastly, the getRatings method provides access to a list of ratings given to the item. This method is particularly useful for calculating the average rating or performing other rating-related operations. By employing these getter methods, the Item class maintains the principles of encapsulation, ensuring that its internal state is protected while providing necessary information to other parts of the application.

```
public List<Double> getRatings() {  
    return ratings;  
}  
  
public double getAverageRating() {  
    if (ratings.isEmpty()) return 0;  
    double sum = 0;  
    for (double rating : ratings) {  
        sum += rating;  
    }  
    return sum / ratings.size();  
}
```

( Example of the getter method)

- **Add Methods:**

The addReview and addRating methods enable the addition of reviews and ratings to an item. The addReview(Review review): Adds a Review object to the reviews list. This method allows users to provide feedback on the item, which is then stored for future reference and display. Moreover, maddRating(double rating): Adds a numerical rating (a double value) to the ratings list. This method accumulates ratings from users, which can be used to compute the average rating of the item.

```
public void addReview(Review review) {  
    reviews.add(review);  
}  
  
public void addRating(double rating) {  
    ratings.add(rating);  
}
```

**(Example of the add method)**



## Items Class:

The Items class is designed to manage a collection of Item objects using a private List called itemList. This list is initialized as an ArrayList, which allows for dynamic resizing and efficient item management. The addItem method enables the addition of new Item objects to this list, effectively expanding the collection as needed. By calling itemList.add(item), the method appends the provided item to the end of the list. The getItemList method provides access to the entire list, returning it as a List<Item>. This method allows other parts of the application to view or process the current collection of items. Together, these methods and the list facilitate organized and flexible management of multiple Item objects, supporting operations such as adding new items and retrieving the full list for display or manipulation.

```
import java.util.ArrayList;
import java.util.List;

public class Items {
    private List<Item> itemList = new ArrayList<>();

    public void addItem(Item item) {
        itemList.add(item);
    }

    public List<Item> getItemList() {
        return itemList;
    }
}
```

## Customer Class

The Customer class represents a customer with basic attributes and methods to manage them. Here's how the class is structured and how it functions

- Constructors:

The default constructor initializes the name and address fields with empty strings. This allows the creation of a Customer object without specifying initial values.

- Parameterized Constructor:

The parameterized constructor allows for the creation of a Customer object with specified name and address. This provides a way to set these fields at the time of object creation.

```
// Default constructor
public Customer() {
    this.name = "";
    this.address = "";
}

// Parameterized constructor
public Customer(String name, String address) {
    this.name = name;
    this.address = address;
}
```

- Methods (Getter and Setter):

The Customer class uses getter and setter methods to manage access to its private fields name and address. The getters methods retrieve the current values of the fields. For instance, getName() returns the value of the name field, while getAddress() returns the value of the address field. They provide a way to access the fields from outside the class, allowing other parts of the application to read the current state of a Customer object. Furthermore, setters methods update the values of the fields. For example, setName(String name) assigns a new value to the name field, and setAddress(String address) assigns a new value to the address field. Setters

## Review Class

The Review class is designed to encapsulate review information with two primary attributes: text, which stores the content of the review, and reviewer, which holds the name of the person who wrote the review. It features a parameterized constructor for initializing these fields, getters getText() and getReviewer() to retrieve the review content and reviewer's name respectively, and a toString() method that provides a formatted string representation of the review as "reviewer: text". This structure ensures that review details are neatly encapsulated, easily accessible, and presented in a clear and user-friendly format.

```
public class Review {  
    private String text;  
    private String reviewer;  
  
    public Review(String text, String reviewer) {  
        this.text = text;  
        this.reviewer = reviewer;  
    }  
  
    public String getText() {  
        return text;  
    }  
  
    public String getReviewer() {  
        return reviewer;  
    }  
  
    @Override  
    public String toString() {  
        return reviewer + ": " + text;  
    }  
}
```

## Seller Class

The Seller class is designed to encapsulate information about a seller, including their name, id, rating, and review. It provides a default constructor that initializes these fields with default values (empty strings for name and id, 0.0 for rating, and an empty string for review) and a parameterized constructor to set these attributes directly. The class includes getter and setter methods for each field, allowing for controlled access and modification of the seller's details. This setup ensures that all relevant information about a seller is managed in a structured and encapsulated manner, supporting both the retrieval and updating of seller data within the application.

```
// Default constructor
public Seller() {
    this.name = "";
    this.id = "";
    this.rating = 0.0;
    this.review = "";
}

// Parameterized constructor
public Seller(String name, String id, double rating, String review) {
    this.name = name;
    this.id = id;
    this.rating = rating;
    this.review = review;
}
```

(A screenshot for the constructor in the Seller class)

```
// Getters and setters
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
```

(A screenshot of some getter and setter used in the class)

## Sellers class:

The Sellers class is designed to manage a collection of Seller objects, utilizing a `List<Seller>` to store them. It features a default constructor that initializes this list as an empty `ArrayList`. The `add(Seller seller)` method allows for adding new Seller instances to the list, while the `remove(String sellerId)` method enables the removal of a seller by their unique ID using a lambda expression to identify and delete the correct entry. The `find(String sellerId)` method searches through the list to find and return a seller with the specified ID, returning null if no match is found. Finally, the `getSellers()` method provides access to the entire list of sellers, allowing for retrieval of all managed seller objects. This class effectively encapsulates seller management, providing robust methods for maintaining and accessing seller data within the application.

```
public class Sellers {  
    private List<Seller> sellers;  
  
    // Default constructor  
    public Sellers() {  
        this.sellers = new ArrayList<>();  
    }  
  
    // Method to add a seller  
    public void add(Seller seller) {  
        sellers.add(seller);  
    }  
  
    // Method to remove a seller by ID  
    public void remove(String sellerId) {  
        sellers.removeIf(seller -> seller.getId().equals(sellerId));  
    }  
  
    // Method to find a seller by ID  
    public Seller find(String sellerId) {  
        for (Seller seller : sellers) {  
            if (seller.getId().equals(sellerId)) {  
                return seller;  
            }  
        }  
        return null;  
    }  
  
    // Method to get all sellers  
    public List<Seller> getSellers() {  
        return sellers;  
    }  
}
```

## Output

window, users can input the item's name, description, and price. Upon clicking the "Add Item" button, the item is added to the item list, which updates to reflect the new addition. Users can select an item from this list to view its details. Once an item is selected, users have the option to add reviews and ratings. The "Add Review" button allows users to enter a review text and the reviewer's name, which then appears in the review list associated with the selected item. Similarly, users can input a rating for the item, which updates its average rating and is displayed in the item list. The secondary window provides a separate view showing a list of sellers, displaying their names and ratings, allowing users to view seller information independently of the item-related interactions.

The image displays two overlapping application windows. The left window, titled "Abdullah Naji(22012364)", is a form for adding and managing items. It includes input fields for "Item Name:", "Description:", and "Price:", followed by an "Add Item" button. Below this is an "Items List" showing a single item: "Name: bacc bag", "Description: color: black, matirial: leather", "Price: \$100.0", and "Average Rating: 5.0". Further down is a "Reviews:" section with one review: "Abdullah: it is realy worth the money". At the bottom are fields for "Add Review:" (with a reviewer name field), "Add Rating:", and buttons for "Add Review" and "Add Rating". The right window, titled "Mohammed Fisal(2200...", displays a "Sellers List" with two entries: "Seller A - Rating: 4.5" and "Seller B - Rating: 3.8".

## **Conclusion**

In summary, the application effectively integrates these classes to create a functional and interactive marketplace platform. The use of JavaFX for the graphical user interface ensures a user-friendly experience, allowing students to manage items, reviews, ratings, and sellers seamlessly. This structured approach not only enhances user engagement but also supports efficient data handling and presentation within the virtual marketplace.