Unit 7 solution

1) What do you understand by event source and event object? Explain how to register an event handler object and how to implement a handler interface?

Event Source and Event Object in JavaFX

In JavaFX, events are generated when a user interacts with UI components.

1. **Event Source:** The component (UI element) that generates an event. For example, a **Button** is an event source when it is clicked.
2. **Event Object:** An instance of the **Event** class that carries information about the event, such as the type of event and the source component.

Registering an Event Handler Object

An **event handler object** processes events for a component. It is registered using the **setOnAction()** method for action events.

Example:

```
Button btn = new Button("Click Me");

btn.setOnAction(new EventHandler<ActionEvent>() {

  @Override

  public void handle(ActionEvent event) {

    System.out.println("Button Clicked!");

  }

});
```

Here, an event handler is registered to the button using `setOnAction()`.

Implementing a Handler Interface

Instead of using an anonymous class, we can implement the **EventHandler** interface in a separate class.

Example:

```
import javafx.application.Application;

import javafx.event.ActionEvent;

import javafx.event.EventHandler;

import javafx.scene.Scene;

import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

class ButtonHandler implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Button Clicked!");
    }
}

public class EventDemo extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button btn = new Button("Click Me");
        btn.setOnAction(new ButtonHandler()); // Registering the event handler

        StackPane root = new StackPane();
        root.getChildren().add(btn);

        Scene scene = new Scene(root, 300, 200);
        primaryStage.setScene(scene);
        primaryStage.setTitle("JavaFX Event Handling");
        primaryStage.show();
    }

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

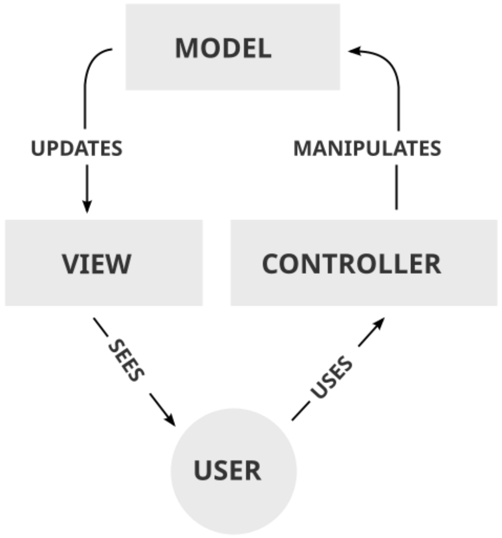
2) With a neat diagram explain the Model view controller design pattern and list out the advantages and disadvantages of using it in designing an application.

Model-View-Controller (MVC) Design Pattern in JavaFX

The **Model-View-Controller (MVC)** pattern is a software design approach that separates an application into three components:

1. **Model:** Represents the application's data and business logic.
2. **View:** Handles the UI and displays data from the model.
3. **Controller:** Manages user interactions, updates the model, and refreshes the view.

MVC Diagram:



Advantages of MVC in JavaFX:

✅ **Separation of concerns** – Easier to manage, debug, and update.
✅ **Code reusability** – Components can be reused in different parts of the application.
✅ **Scalability** – Suitable for large applications.
✅ **Parallel development** – Developers can work on Model, View, and Controller separately.

Disadvantages of MVC in JavaFX:

❌ **Complexity** – Increases the number of classes and interactions.
❌ **Learning curve** – Requires a good understanding of design patterns.
❌ **Overhead** – Can be overkill for small applications.

```
// Model

class CounterModel {
    private int count = 0;
    public int getCount() { return count; }
    public void increment() { count++; }
}

// View (UI)
class CounterView {
    Button btn = new Button("Click Me");
    Label lbl = new Label("Count: 0");
    VBox layout = new VBox(10, lbl, btn);
    Scene getScene() { return new Scene(layout, 300, 200); }
}

// Controller
class CounterController {
    private CounterModel model;
    private CounterView view;

    public CounterController(CounterModel model, CounterView view) {
        this.model = model;
        this.view = view;
        view.btn.setOnAction(e -> {
            model.increment();
            view.lbl.setText("Count: " + model.getCount());
        });
   }
}

// Main Application
public class MVCDemo extends Application {
    @Override
    public void start(Stage primaryStage) {
        CounterModel model = new CounterModel();
        CounterView view = new CounterView();
        new CounterController(model, view);

        primaryStage.setScene(view.getScene());
        primaryStage.setTitle("MVC in JavaFX");
        primaryStage.show();
    }

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

- **`view.btn.setOnAction(...)`**
    - `btn` is a **Button** inside the `view` (CounterView class).
    - `setOnAction()` is a method that registers an **event handler** for button clicks.
- **`e -> { ... }` (Lambda Expression)**
    - This is a **lambda function**, which is a shorter way to define an event handler.
    - It is an alternative to writing an **anonymous inner class**.
- **`model.increment();`**
    - Calls the `increment()` method from the `model` (CounterModel class).
    - This increases the count value by 1.
- **`view.lbl.setText("Count: " + model.getCount());`**
    - Updates the `lbl` (Label) in the `view` with the new count value.
    - `model.getCount()` retrieves the updated count and displays it.

Inner Class

```
class Outer {
    private String message = "Hello from Outer class";

    class Inner {
        void display() {
            System.out.println(message);  // Accessing outer class private data
        }
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();  // Creating an inner class object
        inner.display();
    }
}
```

Inner Class in Java

An **inner class** in Java is a class that is **declared inside another class**. It helps in **encapsulating logic**, improving **readability**, and providing **better access control**.

Types of Inner Classes

Java supports four types of inner classes:

1. **Member Inner Class**
2. **Static Nested Class**
3. **Local Inner Class**
4. **Anonymous Inner Class**

---

1. Member Inner Class

A non-static class inside another class. It has access to all members (even private) of the outer class.

Example:

```
java
CopyEdit
class Outer {
    private String message = "Hello from Outer class";

    class Inner {
        void display() {
            System.out.println(message);  // Accessing outer class private
data
        }
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();  // Creating an inner class
object
        inner.display();
    }
}
```

🔷 **Output:** `Hello from Outer class`

---

2. Static Nested Class

A `static` inner class that **does not require an instance of the outer class**.

Example:

```
java
```

```
CopyEdit
class Outer {
    static class Inner {
        void display() {
            System.out.println("Inside Static Inner Class");
        }
    }

    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner();  // No need to create an
Outer class object
        inner.display();
    }
}
```

◆ **Output:** `Inside Static Inner Class`

---

3. Local Inner Class

A class defined **inside a method**. It can only be accessed within that method.

Example:

```
java
CopyEdit
class Outer {
    void outerMethod() {
        class Inner {
            void display() {
                System.out.println("Inside Local Inner Class");
            }
        }
        Inner inner = new Inner();
        inner.display();
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.outerMethod();
    }
}
```

◆ **Output:** `Inside Local Inner Class`

---

4. Anonymous Inner Class

A class **without a name** that is used for **one-time use**, usually for implementing interfaces or abstract classes.

Example: (Using an interface)

```
java
CopyEdit
interface Greeting {
    void sayHello();
}

public class AnonymousExample {
    public static void main(String[] args) {
        Greeting obj = new Greeting() {  // Anonymous class implementing
Greeting
            public void sayHello() {
                System.out.println("Hello from Anonymous Inner Class!");
            }
        };
        obj.sayHello();
    }
}
```

🔷 **Output:** `Hello from Anonymous Inner Class!`

Advantages of Inner Classes

✅ **Encapsulation:** Keeps related classes together.
✅ **Improved Readability:** Reduces unnecessary class files.
✅ **Access to Outer Class Members:** Can access `private` members of the outer class.

Disadvantages of Inner Classes

❌ **Increases Complexity:** Code might be harder to understand.
❌ **Less Reusable:** Inner classes are tightly coupled with the outer class.