NAME:SOWAD RAHMAN
ID:24341284
section:CSE470-13
Assignment 2

Question 1—-> Answer:

Description:

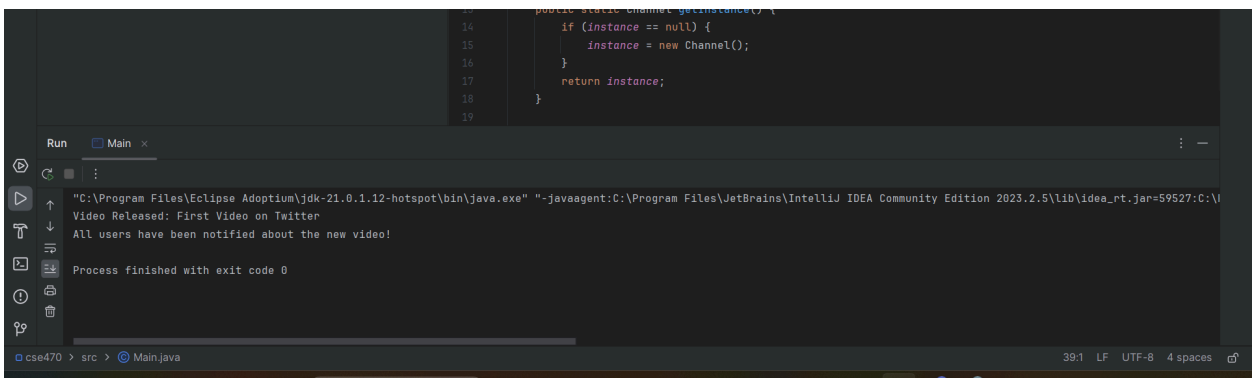
Pattern Used: Singleton Pattern
Why: The Singleton Pattern is ideal for situations where only one instance of a class should exist, as required for creating the single "Twitter" channel in this scenario. This pattern ensures that no additional instances of the channel can be created, thus enforcing that the only channel available is "Twitter." Additionally, it provides a global point of access, making it easier to manage the channel and notify users when a video is released.

Code:

```
class Channel {
  private static Channel instance;
  private String channelName;


  private Channel() {
     channelName = "Twitter";
  }


  public static Channel getInstance() {
     if (instance == null) {
        instance = new Channel();
     }
     return instance;
  }

  // Notify users when a video is released
  public void releaseVideo(String videoTitle) {
     System.out.println("Video Released: " + videoTitle);
     notifyUsers();
```

```
    }

    // Placeholder method to notify users
    private void notifyUsers() {
        System.out.println("All users have been notified about the new video!");
    }
}

// Driver code
public class Main {
    public static void main(String[] args) {
        Channel twitterChannel = Channel.getInstance();
        twitterChannel.releaseVideo("First Video on Twitter");
    }
}
```

OUTPUT:



Question 2—-> Answer

Description:

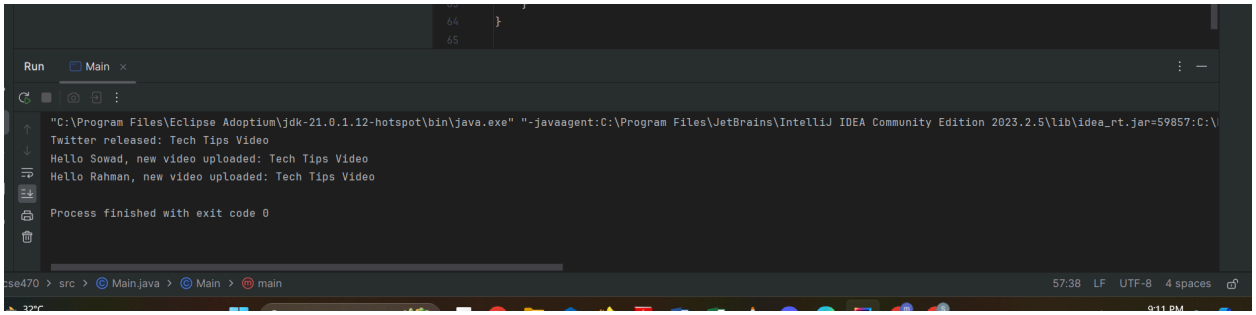
Pattern Used: Observer Pattern
Why: The Observer Pattern is perfect for scenarios where multiple objects (in this case, users) need to be notified of changes made by a subject (the channels). In this case, each channel can have its own observers (subscribed users), and whenever a new video is uploaded to a channel, all subscribed users are notified. This makes the Observer Pattern well-suited for this problem, as it facilitates event-driven notification systems.

Code:

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


interface Observer {
   void update(String videoTitle);
}


class Channel {
   private String name;
   private List<Observer> observers = new ArrayList<>();

   public Channel(String name) {
      this.name = name;
   }

   public void subscribe(Observer observer) {
      observers.add(observer);
   }

   public void unsubscribe(Observer observer) {
      observers.remove(observer);
   }

   public void releaseVideo(String videoTitle) {
      System.out.println(name + " released: " + videoTitle);
      notifyUsers(videoTitle);
   }

   private void notifyUsers(String videoTitle) {
      for (Observer observer : observers) {
         observer.update(videoTitle);
      }
   }
}


class User implements Observer {
   private String name;
```

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

    @Override
    public void update(String videoTitle) {
        System.out.println("Hello " + name + ", new video uploaded: " + videoTitle);
    }
}

// Driver code
public class Main {
    public static void main(String[] args) {
        Channel twitterChannel = new Channel("Twitter");
        User user1 = new User("Sowad");
        User user2 = new User("Rahman");

        twitterChannel.subscribe(user1);
        twitterChannel.subscribe(user2);

        twitterChannel.releaseVideo("Tech Tips Video");
    }
}
```

OUTPUT:



QUESTION 3—->ANSWER

Description:

Patterns Used: Singleton Pattern and Structural Pattern

Why:

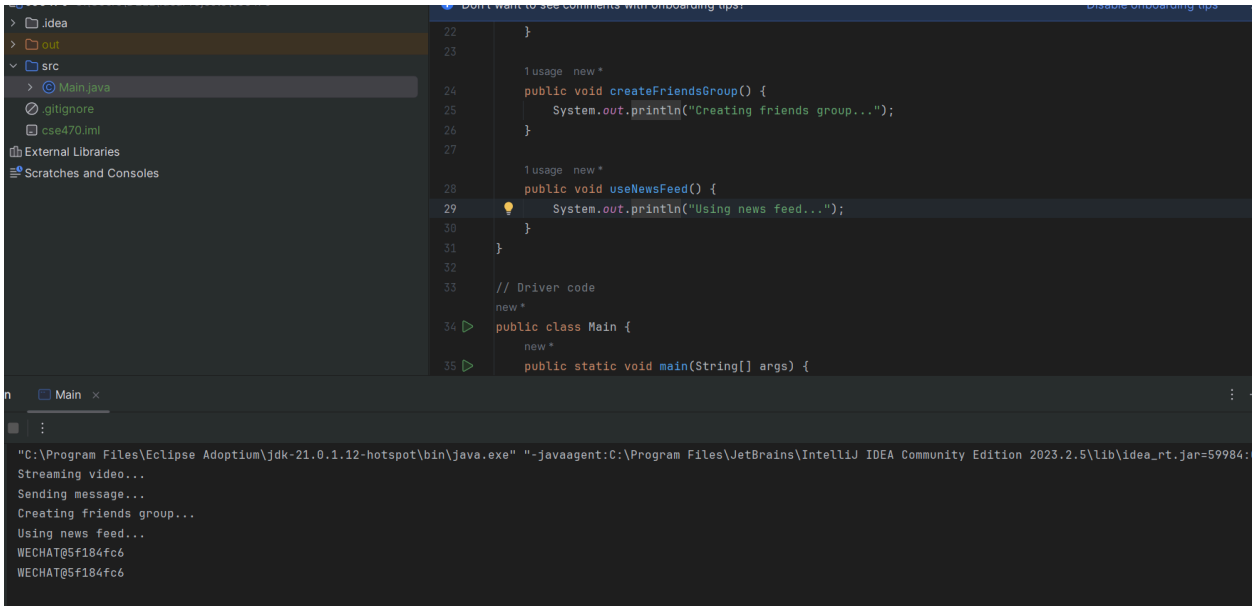
> **Singleton Pattern ensures that only one instance of the WECHAT application exists, which is essential for maintaining a unified platform that integrates features from Messenger, Facebook, and Twitch. This avoids creating multiple instances of the app and ensures consistent functionality.**
>
> **Structural Pattern is used to provide a simplified interface for interacting with complex subsystems (such as video streaming, messaging, and newsfeed management). This pattern hides the complexity of the subsystems and presents a unified interface for using different features of the app, making it easier to extend and maintain.**

CODE:

```
class WECHAT {
  private static WECHAT instance;

  private WECHAT() {

  }

  public static WECHAT getInstance() {
     if (instance == null) {
        instance = new WECHAT();
     }
     return instance;
  }

  public void streamingVideo() {
     System.out.println("Streaming video...");
```

```
    }

    public void sendMessage() {
        System.out.println("Sending message...");
    }

    public void createFriendsGroup() {
        System.out.println("Creating friends group...");
    }

    public void useNewsFeed() {
        System.out.println("Using news feed...");
    }
}

// Driver code
public class Main {
    public static void main(String[] args) {
        WECHAT weChat = WECHAT.getInstance();
        weChat.streamingVideo();
        WECHAT weChat2 = WECHAT.getInstance();
        weChat.sendMessage();
        weChat.createFriendsGroup();
        weChat2.useNewsFeed();

        System.out.println(weChat);
        System.out.println(weChat2);
    }
}
```

OUTPUT:



QUESTION 4—-> ANSWER

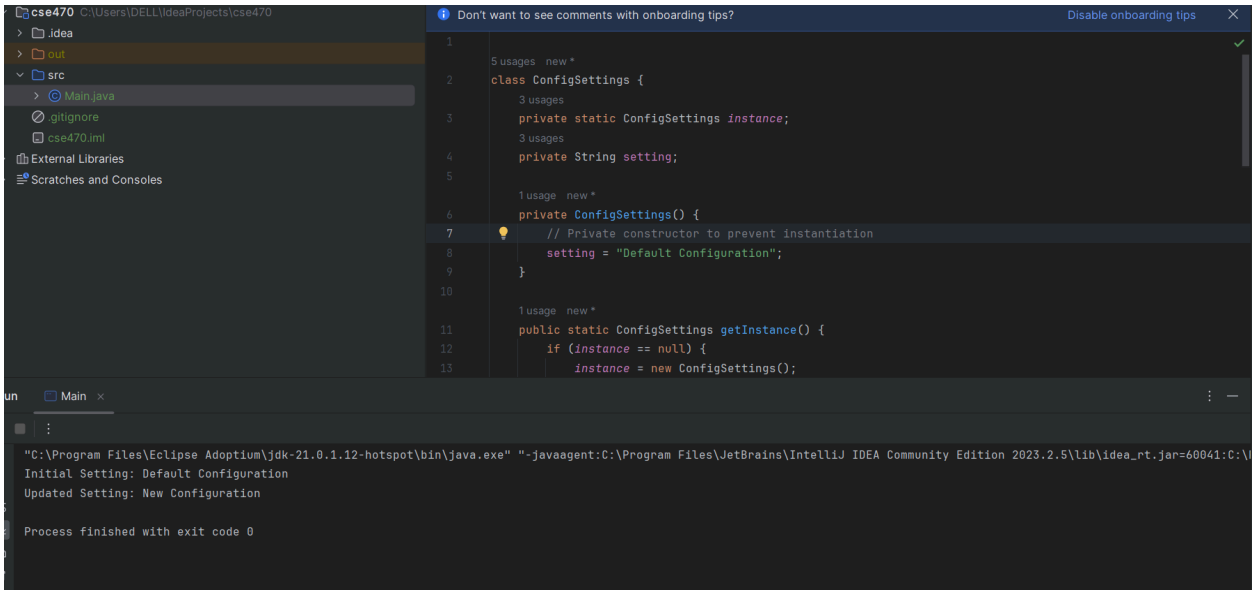
Description:

Pattern Used: Singleton Pattern
Why: The Singleton Pattern is appropriate for ensuring that only one instance of the configuration settings class exists. Like how Mr. Shaheed ensures the consistency of his cakes by being the sole baker, the Singleton Pattern ensures that the settings remain stable and consistent across the entire system. By having only one instance of the settings, it prevents any conflicting changes or instances, ensuring control and stability throughout the application's lifecycle.

CODE:

```
class ConfigSettings {
   private static ConfigSettings instance;
   private String setting;

   private ConfigSettings() {
      // Private constructor to prevent instantiation
      setting = "Default Configuration";
   }

   public static ConfigSettings getInstance() {
      if (instance == null) {
         instance = new ConfigSettings();
      }
      return instance;
   }

   public String getSetting() {
      return setting;
   }

   public void updateSetting(String newSetting) {
      setting = newSetting;
   }
}

// Driver code
public class Main {
   public static void main(String[] args) {
      ConfigSettings settings = ConfigSettings.getInstance();
      System.out.println("Initial Setting: " + settings.getSetting());

      settings.updateSetting("New Configuration");
      System.out.println("Updated Setting: " + settings.getSetting());
   }
}
```

OUTPUT:


cse470 C:\Users\DELL\IdeaProjects\cse470
.idea
out
src
Main.java
.gitignore
cse470.iml
External Libraries
Scratches and Consoles
Don't want to see comments with onboarding tips?
Disable onboarding tips
5 usages new *
class ConfigSettings {
3 usages
private static ConfigSettings instance;
3 usages
private String setting;
1 usage new *
private ConfigSettings() {
// Private constructor to prevent instantiation
setting = "Default Configuration";
}
1 usage new *
public static ConfigSettings getInstance() {
if (instance == null) {
instance = new ConfigSettings();
Main
"C:\Program Files\Eclipse Adoptium\jdk-21.0.1.12-hotspot\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2023.2.5\lib\idea_rt.jar=60041:C:\
Initial Setting: Default Configuration
Updated Setting: New Configuration
Process finished with exit code 0