10/11/2024

Ayesha Arshad

**Sp24-BSe-019**

**Object Oriented Programming**

**(Java )**

**Project: Messaging app**

In the messaging application, three main classes are used: Main, Message, and AppManager. Each class plays a specific role in ensuring that the messaging system works efficiently.

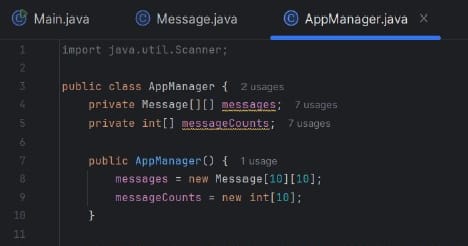
**1. Classes and their Explanations**

**Class 1: Main.java**  
This class is responsible for managing the main interaction with the user. It contains the main() method which runs the program's menu and takes user inputs to trigger different actions.

**Class 2: Message.java**  
This class holds the information about each message. It defines attributes such as the sender, receiver, content, and time details. It also has methods to manipulate the message, like marking it as seen and updating its content. So basically this class works on data.

**Class 3: AppManager.java**  
This class of my code manages the operations of sending, receiving, and updating messages. It interacts with the Message class to maintain the functionality of the messaging system.

**2D array:** In the following code 2D array is used, there is 1 sender but many receivers, the array is set for no of receivers and messages per receiver and the message count is set for the actual limit for messaging data. Thus, there are 10 receivers and eachcan receive upto 10 messages.



**2. Class Attributes**

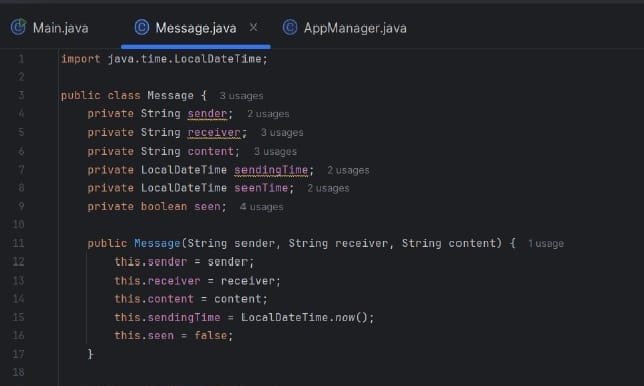
The main class is purely functional

**Attributes of Message Class:**

* String sender: Stores the name of the message sender.
* String receiver: Stores the name of the message receiver.
* String content: Stores the content of the message.
* LocalDateTime sendingTime: Records the date and time when the message was sent.
* LocalDateTime seenTime: Records the date and time when the message was marked as seen.
* boolean seen: Tracks whether the message has been seen or not.

**Attributes of AppManager Class:**

* **Message[][] messages:** A 2D array to store messages for multiple receivers. Each index of the array corresponds to a particular receiver, and each receiver can store multiple messages.
* **String[] contacts:** This array holds the names of all receivers. In this particular program, receivers are identified by their index (0-9), but this array could store actual contact names in a more advanced version.



**Iterative model and programming approach for this code:**

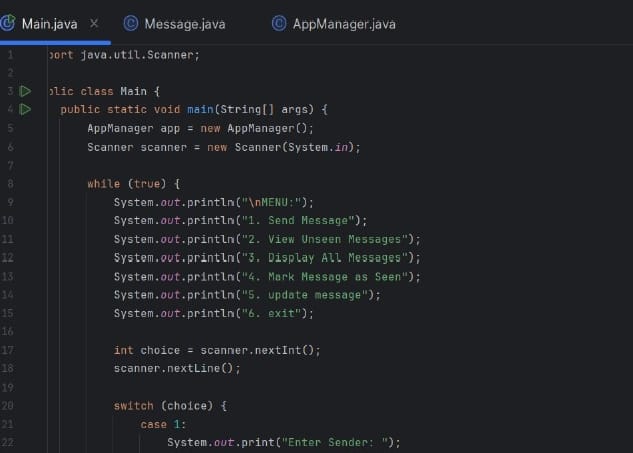
The development of this messaging app followed an iterative approach. Initially, basic features like sending and viewing messages are implemented and then advanced features like unseen messages and marking messages as seen were added moreever Updates to messages were introduced

**Menu Screens**

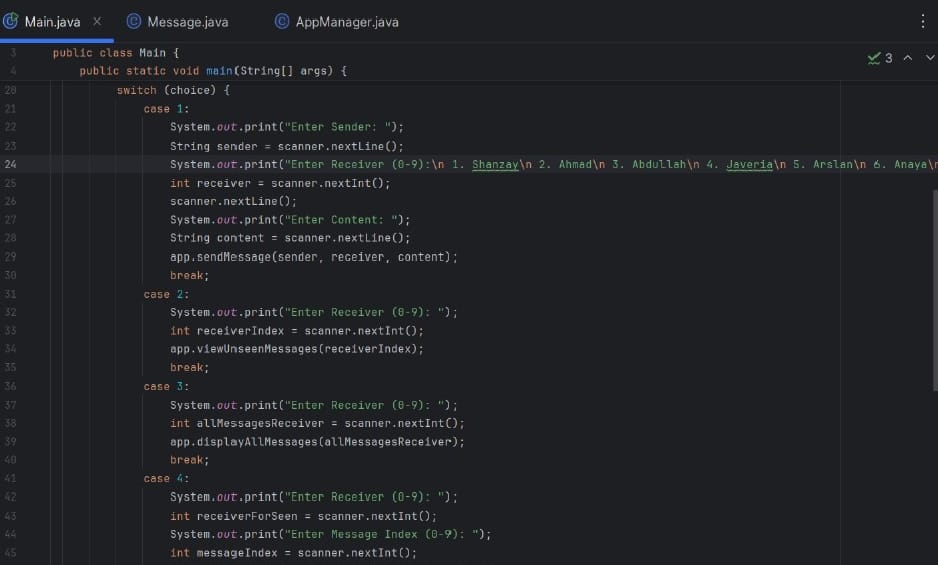
The user interacts with the program via a menu displayed in the console. Here is the full layout of the menu, as it appears during execution:

**Explanation of Each Option:**

* **Option 1 (Send Message):** The user is prompted to enter the sender's name, the receiver's index (0-9), and the message content. This message is stored in the 2D array and marked as unseen.
* **Option 2 (View Unseen Messages):** The user selects a receiver, and the program displays all unseen messages for that receiver.
* **Option 3 (Display All Messages):** The program shows all messages (seen and unseen) for the chosen receiver.
* **Option 4 (Mark Message as Seen):** The user marks a specific message for a receiver as "seen" updating its seen status and timestamp.
* **Option 5 (Update Message):** The user can edit the content of an existing message for a particular receiver.
* **Option 6 (Exit):** This ends the program execution.

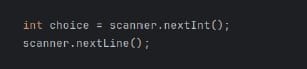
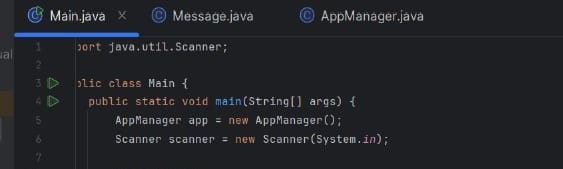


This part of the code is responsible for handling the process of sending a message within the messaging application. It collects the necessary details (sender, receiver, and message content) and then calls the sendMessage method from the AppManager class to store the message.



I added such user interface interaction for every function display thus to specify the initial details of a particular message.

* **User Input for Sender:** This allows the user to specify who is sending the message.
* **Receiver Selection:** Instead of typing the name of the receiver, the user selects a number from a predefined list of people (Shanzay, Ahmad, Abdullah, etc.). This makes it easier to identify receivers.
* **Message Content:** The user enters the content of the message.
* **Storing the Message:** Once all inputs are provided, the message is sent and stored for the chosen receiver, using the sendMessage method.

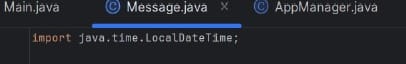
And the **Scanner** class in Java is a built-in utility found in the java.util package that takes user input as choice

**LocalDateTime Class in Java**

The LocalDateTime class, part of the java.time package, is a built-in Java utility that represents both the date and the time without time zone information. It is useful for recording timestamps in applications, such as when messages are sent or seen. And the .now() function provides current time as a built in function that improves continuity. The LocalDateTime class is beneficial in situations where we need to store both date and time information, making the program more efficient by automatically capturing the current system time.

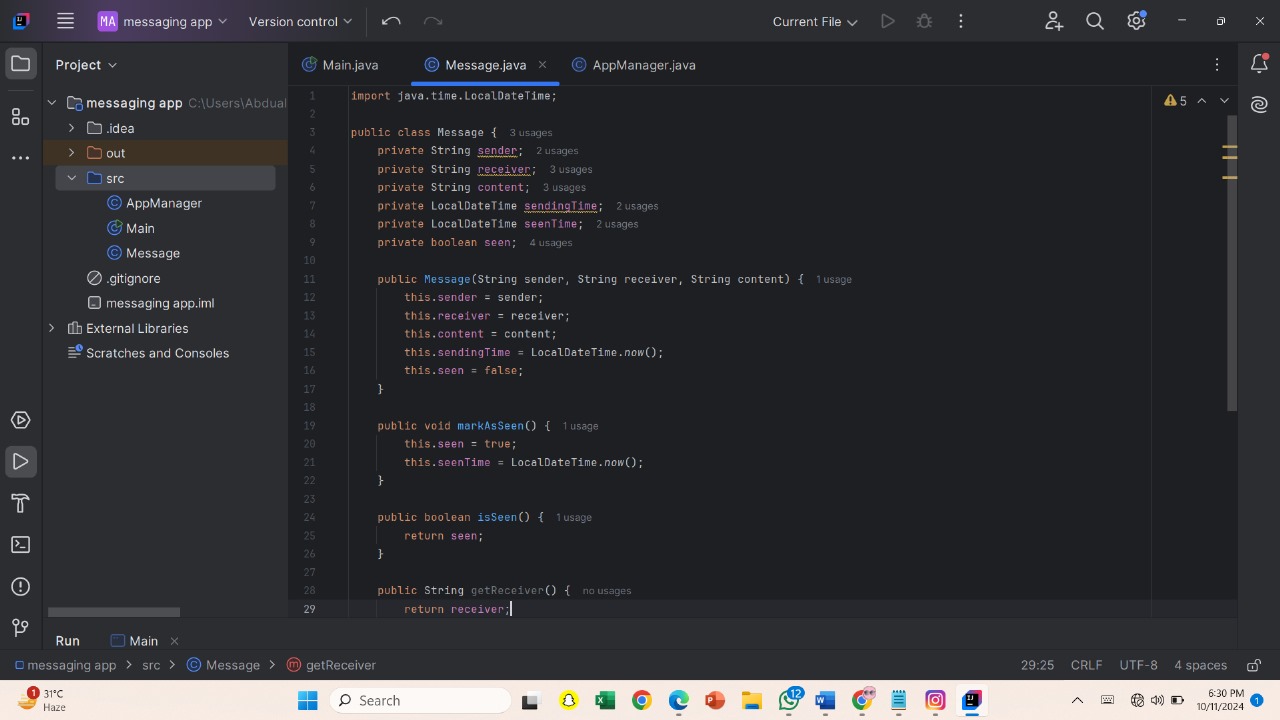
In the code:

* **Sending Time**: The time when the message was sent is recorded using sendingTime = LocalDateTime.now();.
* **Seen Time**: When the message is marked as seen, seenTime records the time it was read.



**Class: Message.java**

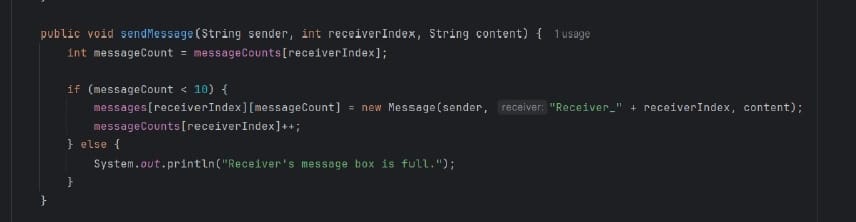
This class of my code includes attributes , constructor to initialize, their getter setters thus encapsulation method , toString and composition that improves code structure.



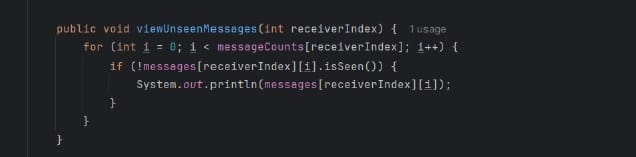
**AppManager Class**

The AppManager class is responsible for managing the core functionality of the messaging system. It provides various methods to send messages, view unseen messages, mark messages as seen, display all messages, and update message content. The class operates using a 2D array of Message objects to store messages and an integer array to keep track of the number of messages for each receiver.

* 1. sendMessage

. This method allows a sender to send a message to a specific receiver. It first checks whether the receiver's message box has space (fewer than 10 messages). If space is available, it creates a new Message object and stores it in the messages array. The method then increments the message count for that receiver

1. **viewUnseenMessages**



This method displays all the unseen messages for a specific receiver. It loops through the receiver's messages and prints those that haven't been marked as seen.

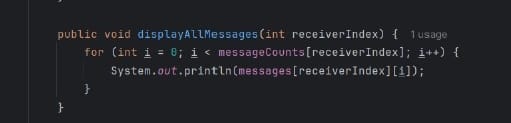
1. **markMessageAsSeen**

This method marks a specific message as "seen." It first checks if the message and receiver indexes are valid, then marks the message as seen using the markAsSeen() method of the Message class.



1. **displayAllMessages**

This method prints all the messages (both seen and unseen) for a specific receiver. It loops through the array of messages for the receiver and displays each one.



1. **updateMessage**

This method updates the content of a specific message for a receiver. If the provided indexes for the receiver and message are valid, it updates the message content using the setContent() method of the Message class.



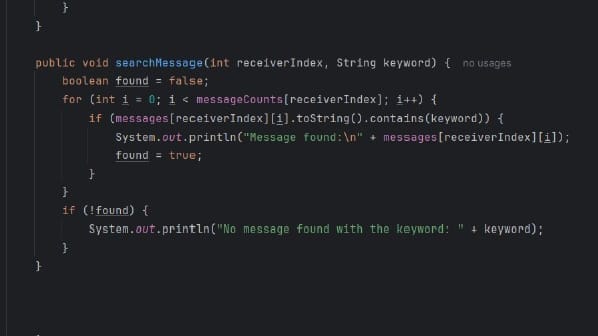
1. **Search message**

 A boolean variable found is initialized to false. This variable will help track whether any matching messages are found during the search.

 The function uses a for loop to iterate through the messages of the specified receiver ,receiverIndex. The loop runs from 0 to the number of messages currently stored for that receiver (messageCounts[receiverIndex]).

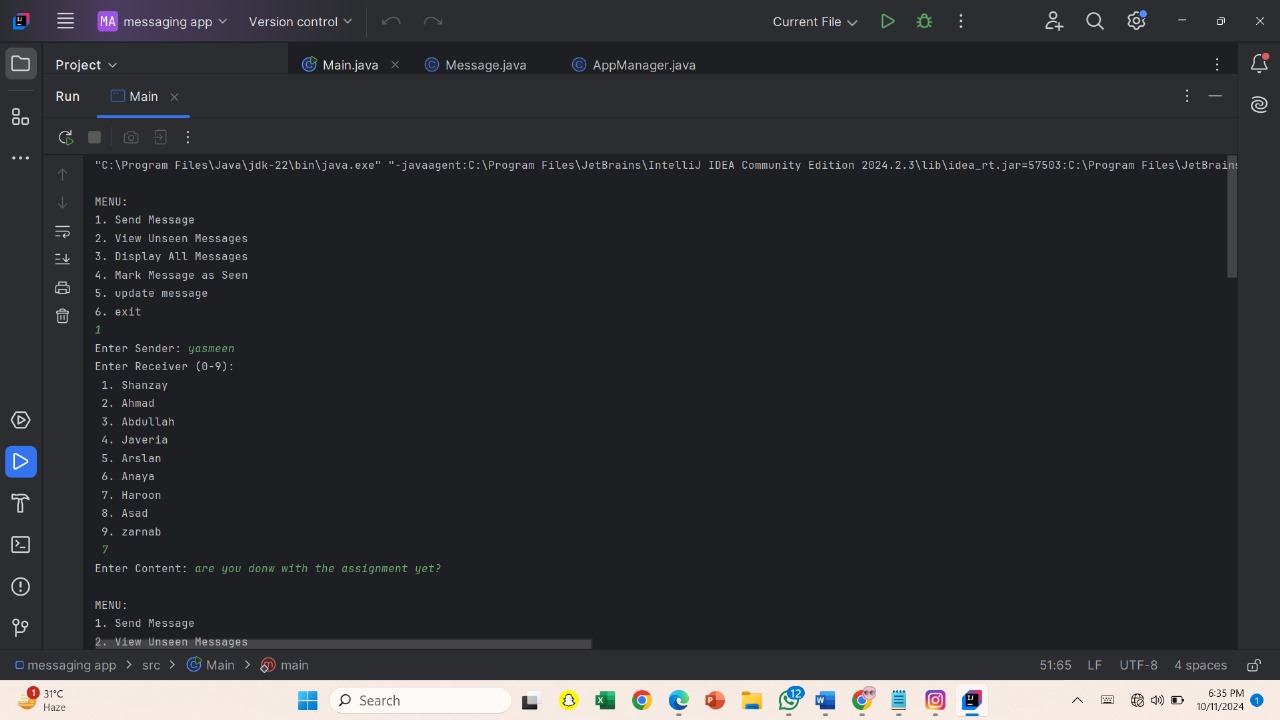
 Inside the loop, the toString() method of the Message class is called for each message. The method converts the message object into a readable string format, which is then checked to see if it contains the keyword.

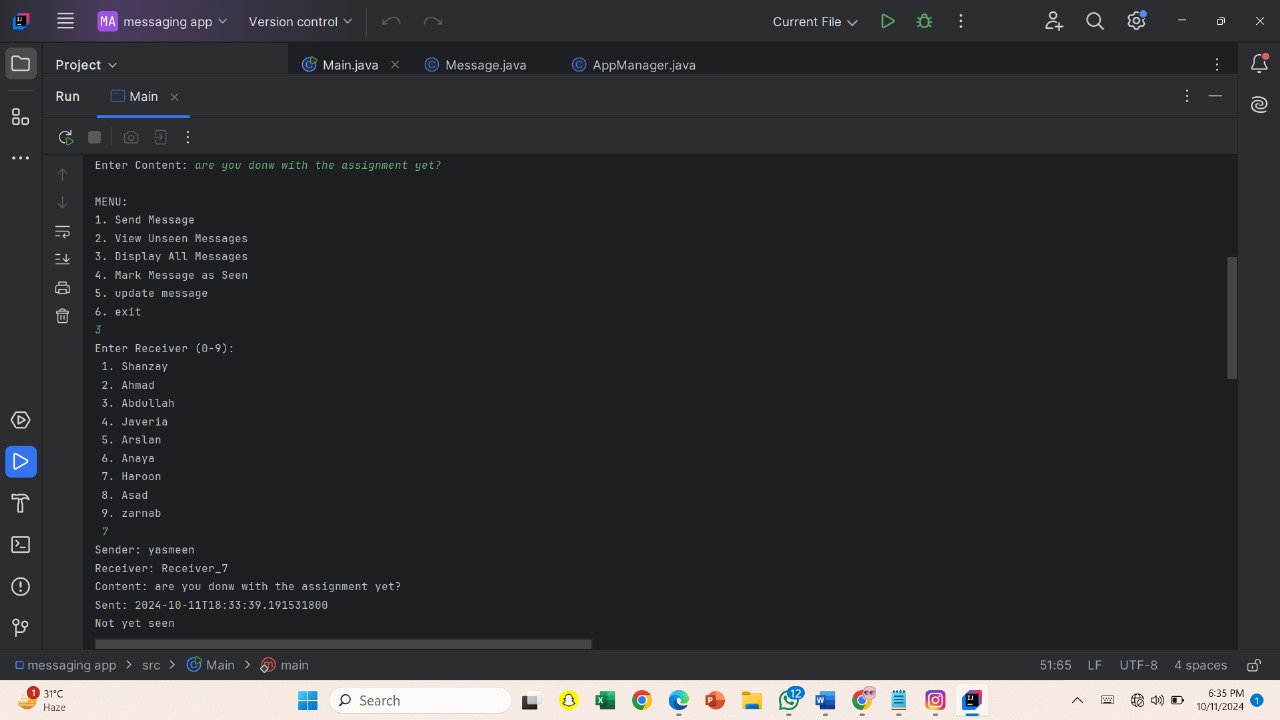
 If a matching message is found, it prints the message and sets found to true

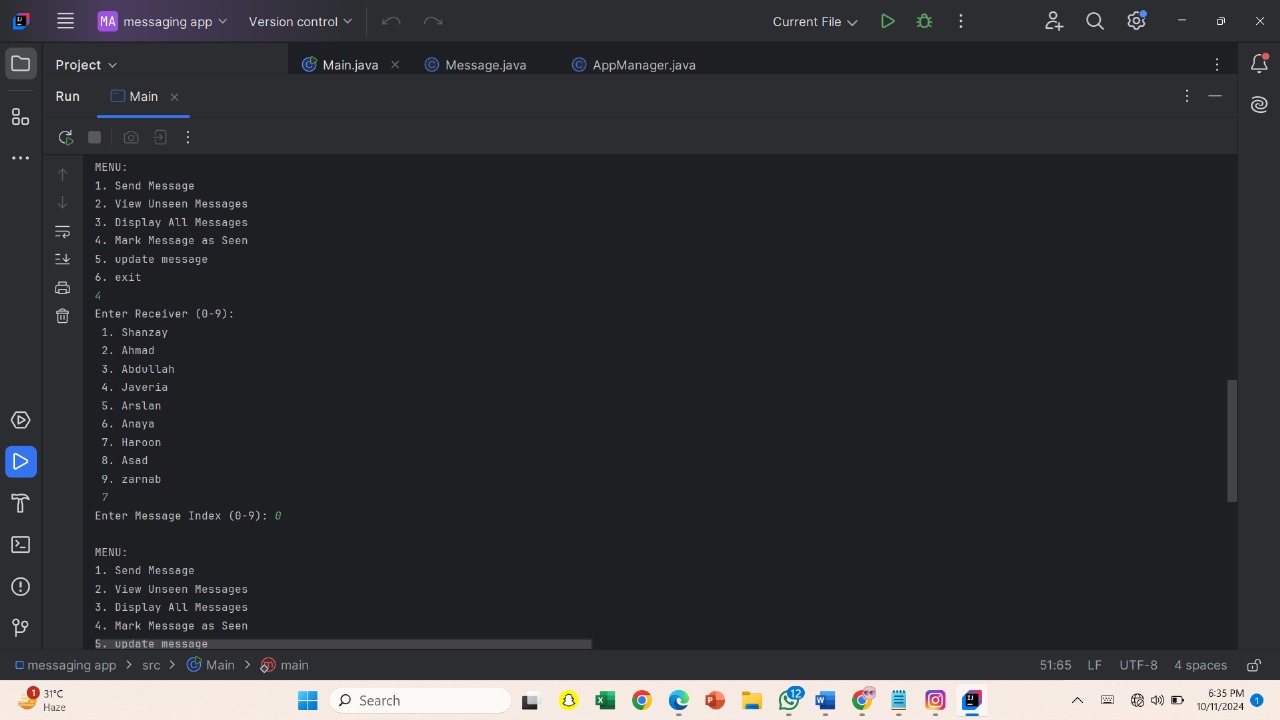


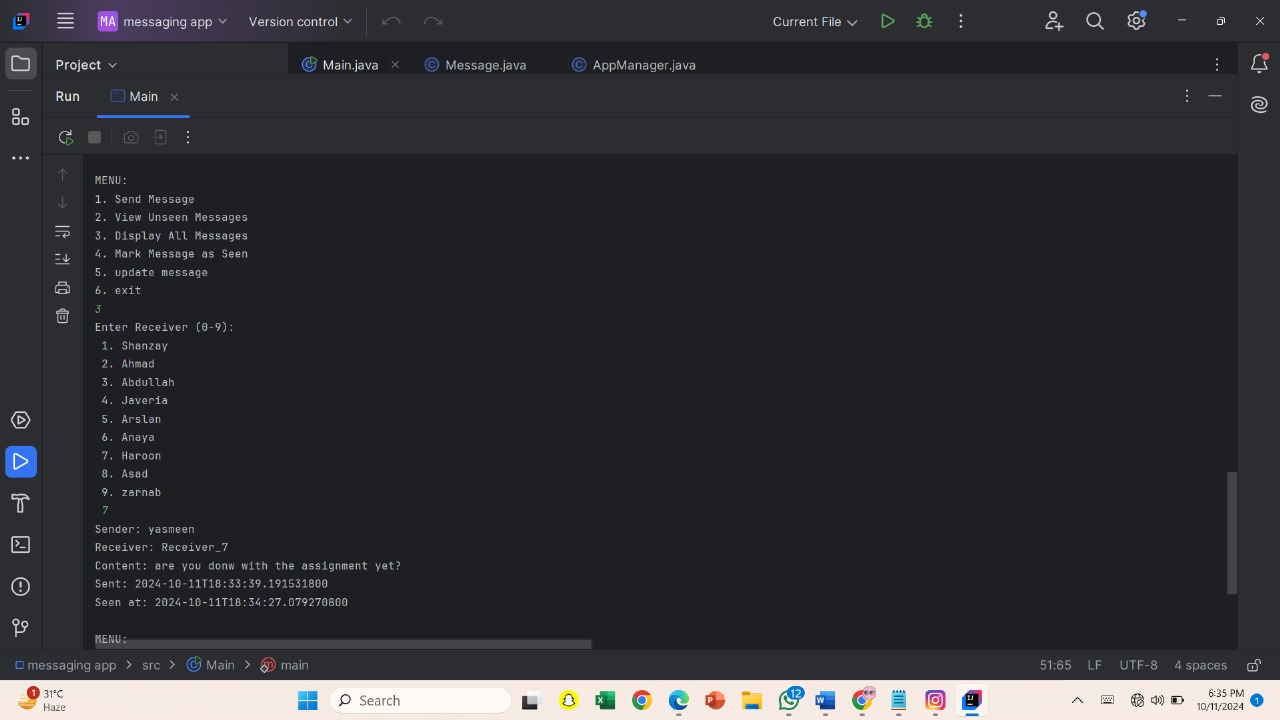
**User Interface Console Explanation**

The **console-based user interface** in this messaging app project provides a simple yet effective way for users to interact with the application via text commands. This interface presents a **menu-driven system**, guiding users to choose specific operations by typing a number corresponding to the desired action.









**UML Diagram:**

