**Introduction**

This document serves as a comprehensive guide to the Java messaging application. The application allows users to send and manage messages between different users in a simple console-based interface. The application consists of three main classes: Main, MessageApp, and Message. This documentation will explain the purpose of each class, its attributes, key methods, and the overall logic behind the application.

**Class Descriptions**

**1. Main Class**

The Main class is the starting point of the application. It interacts with users by displaying options and processing their inputs.

**Key Features:**

* **Menu Display**: The class presents a welcome message and a menu with various options (such as sending a message, displaying messages, and managing contacts).
* **User Interaction**: The application remains in a loop, allowing users to select options repeatedly until they choose to exit by selecting option 0.
* **Method Calls**: Based on the user's choice, the class invokes methods from the MessageApp class to perform different functionalities.

**Example Workflow:**

1. The user starts the application and sees the menu.
2. The user selects an option (e.g., to send a message).
3. The program processes the selection and displays the relevant prompt or information.

**Loop Explanation:**

* **Main Loop**: The primary loop in the Main class continuously displays the menu until the user chooses to exit. It checks the user's input against the available options.
  + **Check for Valid Input**: If the user inputs an option that does not correspond to any of the available menu choices, the loop prompts them to enter a valid option again.

**2. MessageApp Class**

The MessageApp class is responsible for managing the core functionalities of the messaging system, including sending messages, storing them, and managing contacts.

**Attributes:**

* **Message Array**: This array holds up to 1000 messages. Each message is an instance of the Message class.
* **Contact Array**: This array holds up to 500 contact numbers.
* **Message Counter**: A counter keeps track of the total number of messages sent, ensuring unique message IDs.
* **Contact Counter**: A counter tracks the number of contacts in the contact list.

**Key Methods:**

* **sendMessage()**: This method allows the user to send a message by entering the sender's number, receiver's number, and the message content.
  + **Logic**:
    - The method first checks if the sender and receiver numbers are in the contact list.
    - If a number is not present, it prompts the user to add it to the contact list.
    - The message is then stored in the message array, and the Message class is used to create a new message object with a unique ID.
* **addContact()**: This method allows users to add a new contact to their list.
  + **Logic**:
    - It checks if the contact already exists using the contactExists() helper method.
    - If the contact is new, it is added to the contact array, ensuring no duplicates are allowed.

**Loop Explanation:**

* **Message Display Loop**: In methods like displayAllMessages() and displaySpecificMessages(), loops iterate through the message array to retrieve and display messages.
  + **Check for Non-empty Array**: The loop checks if there are any messages stored in the array. If there are no messages, it notifies the user that no messages are available.
  + **Sorting Loop**: The bubble sort algorithm iterates through the messages to arrange them in chronological order, checking adjacent messages and swapping them if necessary.
* **Delete Message Loop**: The deleteMessage() method includes a loop that searches for a message by its unique ID.
  + **Check for Existence**: The loop continues until it finds the specified message ID or reaches the end of the array, ensuring that users cannot delete messages that do not exist.
* **Sent Messages Loop**: The displayNotSent() method includes a loop that checks each message's status.
  + **Check for Status**: It prints only the messages marked as not sent, helping users identify which messages failed to send.

**3. Message Class**

The Message class represents a single message in the messaging application, encapsulating all relevant details about the message.

**Attributes:**

* **sender**: The phone number of the person sending the message.
* **receiver**: The phone number of the person receiving the message.
* **content**: The actual text content of the message.
* **messageId**: A unique identifier for each message, generated sequentially.
* **status**: A boolean value indicating whether the message was sent successfully.
* **timestamp**: The date and time when the message was created, recorded using LocalDateTime.
* **seen**: A boolean flag indicating whether the message has been seen by the receiver.

**Key Methods:**

* **Getters and Setters**: These methods provide access to the private attributes of the Message class. They allow other classes to retrieve and modify message details easily.
* **toString()**: This method returns a string representation of the message object, which is useful for displaying the message details in a human-readable format.

**Logic Overview**

The application is structured to provide a clear and efficient way to send and manage messages. Here are some key logical components:

* **Contact Management**: Before sending a message, the application checks whether the sender and receiver are in the contact list. This ensures that all participants in a conversation are recognized by the system, preventing unnecessary duplicate entries.
* **Message Storage**: Messages are stored in a fixed-size array that can hold up to 1000 messages. This limit helps manage memory efficiently, allowing users to send a considerable number of messages without overwhelming the system.
* **Sorting Mechanism**: The application uses a bubble sort algorithm to sort messages by their timestamps before displaying them. This ensures users can view their messages in chronological order, making it easier to track conversations.
* **Error Handling**: The application includes checks to handle scenarios where users attempt to send messages when storage is full or delete messages that do not exist. This helps maintain the application’s stability and usability.

**Conclusion**

The Java messaging application is a straightforward yet functional tool for sending and managing messages. Its modular design makes it easy to extend and enhance with additional features, such as editing messages or implementing group chats in the future.

**Possible Future Improvements:**

* **Message Editing**: Allow users to edit previously sent messages.
* **Group Messaging**: Introduce functionality for sending messages to multiple recipients simultaneously.
* **Message Persistence**: Save messages to a file or database for long-term storage and retrieval.
* **User Authentication**: Implement a login system to ensure that only authorized users can access certain features.