**1**

Сегодня мы напишем чат. Набор программ с помощью которого можно будет обмениваться текстовыми сообщения.  
Набор будет состоять из одного сервера и нескольких клиентов, по одному для каждого участника чата.

Начнем с сервера. Нам понадобятся такие классы:  
1) **Server** - основной класс сервера.  
2) **MessageType** - **enum**, который отвечает за тип сообщений пересылаемых между клиентом и сервером.  
3) **Message** - класс, отвечающий за пересылаемые сообщения.  
4) **Connection** - класс соединения между клиентом и сервером.  
5) **ConsoleHelper** - вспомогательный класс, для чтения или записи в консоль.

Создай эти классы и добавь метод main в класс **Server**.

**Требования:**

* •

Класс Server должен быть публичным и создан в отдельном файле.

* •

Перечисление MessageType должно быть публичным и создано в отдельном файле.

* •

Класс Message должен быть публичным и создан в отдельном файле.

* •

Класс Connection должен быть публичным и создан в отдельном файле.

* •

Класс ConsoleHelper должен быть публичным и создан в отдельном файле.

* •

В классе Server должен быть создан метод public static void main(String[] args).

**2**

Первым делом, для удобства работы с консолью реализуем класс **ConsoleHelper**. В дальнейшем, вся работа с консолью должна происходить через этот класс.

**Добавь в него:**  
1. Статическое поле типа BufferedReader, инициализированное с помощью System.in.  
2. Статический метод writeMessage(String **message**), который должен выводить сообщение message в консоль.  
3. Статический метод String readString(), который должен считывать строку с консоли.  
Если во время чтения произошло исключение, вывести пользователю сообщение "  
***Произошла ошибка при попытке ввода текста. Попробуйте еще раз.***"  
И повторить ввод. Метод не должен пробрасывать исключения IOException наружу.  
Другие исключения не должны обрабатываться.  
4. Статический метод int readInt(). Он должен возвращать введенное число и использовать метод readString().  
Внутри метода обработать исключение **NumberFormatException**.  
Если оно произошло вывести сообщение  
"***Произошла ошибка при попытке ввода числа. Попробуйте еще раз."***  
И повторить ввод числа.

В этой задаче и далее, если не указано дополнительно другого, то все поля класса должны быть приватными, а методы публичными.

**Требования:**

* •

В классе ConsoleHelper должно быть создано и инициализировано приватное, не финальное, статическое поле типа BufferedReader.

* •

В классе ConsoleHelper должен быть реализован статический метод writeMessage(String message), выводящий сообщение на консоль.

* •

В классе ConsoleHelper должен быть реализован статический метод readString, возвращающий строку считанную с консоли.

* •

В классе ConsoleHelper должен быть реализован статический метод readInt, возвращающий число считанное с консоли.

* •

Метод readInt должен использовать метод readString для чтения с консоли.

* •

Метод readString должен перехватывать IOException, выводить сообщение о некорректном вводе и повторять считывание с консоли.

* •

Метод readInt должен перехватывать NumberFormatException, выводить сообщение о некорректном вводе и повторять считывание с консоли.

**3**

Прежде, чем двигаться дальше, нужно разработать протокол общения клиента и сервера.

**Сформулируем основные моменты протокола:**  
- Когда новый клиент хочет подсоединиться к серверу, сервер должен запросить имя клиента.  
- Когда клиент получает запрос имени от сервера он должен отправить свое имя серверу.  
- Когда сервер получает имя клиента он должен принять это имя или запросить новое.  
- Когда новый клиент добавился к чату, сервер должен сообщить остальным участникам о новом клиенте.  
- Когда клиент покидает чат, сервер должен сообщить остальным участникам об этом.  
- Когда сервер получает текстовое сообщение от клиента, он должен переслать его всем остальным участникам чата.

Добавь для каждого пункта вышеописанного протокола соответствующее значение в **enum**  
**MessageType:**  
1) NAME\_REQUEST - запрос имени.  
2) USER\_NAME - имя пользователя.  
3) NAME\_ACCEPTED - имя принято.  
4) TEXT - текстовое сообщение.  
5) USER\_ADDED - пользователь добавлен.  
6) USER\_REMOVED - пользователь удален.

**Требования:**

* •

В перечислении MessageType должен присутствовать элемент NAME\_REQUEST.

* •

В перечислении MessageType должен присутствовать элемент USER\_NAME.

* •

В перечислении MessageType должен присутствовать элемент NAME\_ACCEPTED.

* •

В перечислении MessageType должен присутствовать элемент TEXT.

* •

В перечислении MessageType должен присутствовать элемент USER\_ADDED.

* •

В перечислении MessageType должен присутствовать элемент USER\_REMOVED.

**4**

Сообщение **Message** - это данные, которые одна сторона отправляет, а вторая принимает.  
Каждое сообщение должно иметь тип **MessageType**, а некоторые и дополнительные данные, например, текстовое сообщение должно содержать текст.  
Т.к. сообщения будут создаваться в одной программе, а читаться в другой, удобно воспользоваться механизмом сериализации для перевода класса в последовательность битов и наоборот.

**Добавь в класс Message:**  
1) Поддержку интерфейса **Serializable**.  
2) final поле типа **MessageType type**, которое будет содержать тип сообщения.  
3) final поле типа **String data**, которое будет содержать данные сообщения.  
4) **Геттеры** для этих полей.  
5) Конструктор, принимающий только **MessageType**, он должен проинициализировать поле type переданным параметром, а поле data оставить равным **null**.  
6) Конструктор, принимающий **MessageType** **type** и **String data**.  
Он должен также инициализировать все поля класса.

**Требования:**

* •

Класс Message должен поддерживать интерфейс Serializable.

* •

Приватное поле type в классе Message должно быть объявлено с модификатором final и быть типа MessageType.

* •

Приватное поле data в классе Message должно быть объявлено с модификатором final и быть типа String.

* •

В классе Message должны быть созданы корректные геттеры для полей data и type.

* •

Конструктор класса Message с одним параметром типа MessageType должен инициализировать поле type и оставлять поле data равным null.

* •

Конструктор класса Message с двумя параметрами (MessageType и String) должен инициализировать поля type и data.

**5**

Клиент и сервер будут общаться через сокетное соединение.  
Одна сторона будет записывать данные в сокет, а другая читать.  
Их общение представляет собой обмен сообщениями **Message**.  
Класс **Connection** будет выполнять роль обертки над классом **java.net.Socket**,  
которая должна будет уметь сериализовать и десериализовать объекты типа **Message** в сокет.  
Методы этого класса должны быть готовы к вызову из разных потоков.

**Добавь в класс Connection:**  
1) Final поля:  
а) **Socket** socket  
б) **ObjectOutputStream** out  
в) **ObjectInputStream** in

2) Конструктор, который должен принимать **Socket** в качестве параметра и инициализировать поля класса.  
Для инициализации полей in и out используй соответствующие потоки сокета.  
Конструктор может бросать исключение IOException.  
Создать объект класса **ObjectOutputStream** нужно до того, как будет создаваться объект класса ObjectInputStream,  
иначе может возникнуть взаимная блокировка потоков, которые хотят установить соединение через класс **Connection**.  
Более подробно об этом ты можешь прочитать в спецификации класса **ObjectInputStream**.

3) Метод void send(Message **message**) throws IOException.  
Он должен записывать (сериализовать) сообщение message в ObjectOutputStream.  
Этот метод будет вызываться из нескольких потоков.  
Позаботься, чтобы запись в объект ObjectOutputStream была возможна только одним потоком в определенный момент времени,  
остальные желающие ждали завершения записи.  
При этом другие методы класса Connection не должны быть заблокированы.

4) Метод Message receive() throws IOException, ClassNotFoundException.  
Он должен читать (десериализовать) данные из ObjectInputStream.  
Сделай так, чтобы операция чтения не могла быть одновременно вызвана несколькими потоками,  
при этом вызов других методы класса **Connection** не блокировать.

5) Метод SocketAddress getRemoteSocketAddress(), возвращающий удаленный адрес сокетного соединения.

6) Метод void close() throws IOException, который должен закрывать все ресурсы класса.

Класс **Connection** должен поддерживать интерфейс **Closeable**.

**Требования:**

* •

Класс Connection должен поддерживать интерфейс Closeable.

* •

В классе Connection должно быть создано private final поле socket типа Socket.

* •

В классе Connection должно быть создано private final поле out типа ObjectOutputStream.

* •

В классе Connection должно быть создано private final поле in типа ObjectInputStream.

* •

В классе Connection должен быть создан конструктор с одним параметром типа Socket инициализирующий поля класса в соответствии с условием задачи.

* •

В классе Connection должен быть корректно реализован метод send c одним параметром типа Message.

* •

В классе Connection должен быть корректно реализован метод receive без параметров.

* •

Метод getRemoteSocketAddress класса Connection должен возвращать удаленный адрес сокетного соединения.

* •

Метод close класса Connection должен закрывать потоки чтения, записи и сокетное соединение.

**6**

Приступим к самому важному - написанию класса **Server**.  
Сервер должен поддерживать множество соединений с разными клиентами одновременно.  
Это можно реализовать с помощью следующего алгоритма:

- Сервер создает серверное сокетное соединение.  
- В цикле ожидает, когда какой-то клиент подключится к сокету.  
- Создает новый поток обработчик Handler, в котором будет происходить обмен сообщениями с клиентом.  
- Ожидает следующее соединение.

Добавь:

1) В класс **Server** приватный статический вложенный класс **Handler**, унаследованный от Thread.  
2) В класс **Handler** поле socket типа Socket.  
3) В класс **Handler** конструктор, принимающий в качестве параметра Socket и инициализирующий им соответствующее поле класса.  
4) Метод main класса **Server**, должен:

а) Запрашивать порт сервера, используя ConsoleHelper.  
б) Создавать серверный сокет java.net.ServerSocket, используя порт из предыдущего пункта.  
в) Выводить сообщение, что сервер запущен.  
г) В бесконечном цикле слушать и принимать входящие сокетные соединения только что созданного серверного сокета.  
д) Создавать и запускать новый поток Handler, передавая в конструктор сокет из предыдущего пункта.  
е) После создания потока обработчика Handler переходить на новый шаг цикла.  
ж) Предусмотреть закрытие серверного сокета в случае возникновения исключения.  
з) Если исключение Exception все же произошло, поймать его и вывести сообщение об ошибке.

**Требования:**

* •

В классе Server должен быть создан приватный статический класс Handler, унаследованный от класса Thread.

* •

В классе Handler должно быть создано поле socket типа Socket.

* •

Конструктор класса Handler должен принимать один параметр типа Socket и инициализировать поле socket.

* •

Метод main должен считывать с клавиатуры порт сервера используя метод readInt класса ConsoleHelper.

* •

Метод main должен корректно реализовывать бесконечный цикл описанный в условии задачи.

**7**

Т.к. сервер может одновременно работать с несколькими клиентами, нам понадобится метод для отправки сообщения сразу всем.

**Добавь в класс Server:**

1. Статическое поле **Map**<String, Connection> **connectionMap**, где ключом будет имя клиента, а значением - соединение с ним.  
2. Инициализацию поля из **п.7.1** с помощью подходящего **Map** из библиотеки **java.util.concurrent**, т.к. работа с этим полем будет происходить из разных потоков и нужно обеспечить потокобезопасность.  
3. Статический метод void sendBroadcastMessage(Message **message**), который должен отправлять сообщение message всем соединениям из **connectionMap**.  
Если при отправке сообщение произойдет исключение **IOException**, нужно отловить его и сообщить пользователю, что не смогли отправить сообщение.

**Требования:**

* •

В классе Server должно существовать статическое приватное поле connectionMap типа Map.

* •

Поле connectionMap должно быть инициализировано потокобезопасной реализаций интерфейса Map из пакета java.util.concurrent.

* •

В классе Server должен быть корректно реализован статический метод sendBroadcastMessage(Message message), отправляющий сообщение всем соединениям из connectionMap.

**8**

Класс **Handler** должен реализовывать протокол общения с клиентом.  
Выделим из протокола отдельные этапы и реализуем их с помощью отдельных методов:

**Этап первый** - это этап рукопожатия (знакомства сервера с клиентом).  
Реализуем его с помощью приватного метода String serverHandshake(Connection connection) throws IOException, **ClassNotFoundException**.  
Метод в качестве параметра принимает соединение connection, а возвращает имя нового клиента.

**Реализация метода должна:**  
1) **Сформировать** и **отправить** команду запроса имени пользователя  
2) **Получить** **ответ** клиента  
3) **Проверить**, что получена команда с именем пользователя  
4) **Достать** из ответа **имя**, **проверить**, что оно не пустое и пользователь с таким именем еще не подключен (используй connectionMap)  
5) **Добавить** **нового** **пользователя** и **соединение** с ним в connectionMap  
6) **Отправить** **клиенту** команду информирующую, что его имя принято  
7) Если какая-то **проверка** не прошла, **заново** запросить имя клиента  
8) **Вернуть** принятое **имя** в качестве возвращаемого значения

**Требования:**

* •

В классе Handler должен присутствовать метод private String serverHandshake(Connection connection).

* •

Метод serverHandshake должен отправлять запрос имени используя метод send класса Connection.

* •

До тех пор, пока тип сообщения полученного в ответ не будет равен MessageType.USER\_NAME, запрос имени должен быть выполнен снова.

* •

В случае, если в ответ пришло пустое имя, запрос имени должен быть выполнен снова.

* •

В случае, если имя уже содержится в connectionMap, запрос имени должен быть выполнен снова.

* •

После успешного проведения всех проверок, метод serverHandshake должен добавлять новую пару (имя, соединение) в connectionMap и отправлять сообщение о том, что имя было принято.

* •

Метод serverHandshake должен возвращать имя нового клиента с которым было установлено соединение.

**9**

**Этап второй**, но не менее важный - отправка клиенту (новому участнику) информации об остальных клиентах (участниках) чата.

Для этого добавь приватный метод void notifyUsers(Connection connection, String userName) throws IOException, где connection - соединение с участником, которому будем слать информацию, а userName - его имя.  
**Метод должен:**  
1) Пройтись по connectionMap.  
2) У каждого элемента из п.1 получить имя клиента, сформировать команду с типом USER\_ADDED и полученным именем.  
3) Отправить сформированную команду через connection.  
4) Команду с типом USER\_ADDED и именем равным **userName** отправлять не нужно, пользователь и так имеет информацию о себе.

**Требования:**

* •

В классе Handler должен быть создан метод private void notifyUsers(Connection connection, String userName).

* •

Метод notifyUsers() должен отправлять через connection сообщение о том, что был добавлен пользователь с именем name для каждого имени из connectionMap.

* •

Метод notifyUsers() НЕ должен отправлять сообщение о добавлении пользователя, если имя пользователя совпадает со вторым параметром метода (userName).

**10**

**Этап третий** - главный цикл обработки сообщений сервером.

Добавь приватный метод void serverMainLoop(Connection **connection**, String **userName**) throws IOException, ClassNotFoundException, где значение параметров такое же, как и у метода notifyUsers().

**Он должен:**  
1. Принимать сообщение клиента  
2. Если принятое сообщение - это текст (тип TEXT), то формировать новое текстовое сообщение путем конкатенации: имени клиента, двоеточия, пробела и текста сообщения.  
Например, если мы получили сообщение с текстом "***привет чат***" от пользователя "***Боб***", то нужно сформировать сообщение "***Боб: привет чат***".  
3. Отправлять сформированное сообщение всем клиентам с помощью метода sendBroadcastMessage().  
4. Если принятое сообщение не является текстом, вывести сообщение об ошибке  
5. Организовать бесконечный цикл, внутрь которого перенести функционал пунктов ***10.1-10.4***.

**Требования:**

* •

В классе Handler должен быть создан метод private void serverMainLoop(Connection connection, String userName).

* •

Метод serverMainLoop() должен в бесконечном цикле получать сообщения от клиента (используя метод receive() класса Connection).

* •

Если сообщение, полученное методом serverMainLoop(), имеет тип MessageType.TEXT, то должно быть отправлено новое сообщение всем участникам чата используя метод sendBroadcastMessage() (форматирование сообщения описано в условии).

* •

Если сообщение, полученное методом serverMainLoop(), имеет тип отличный от MessageType.TEXT, метод sendBroadcastMessage() не должен быть вызван, и должно быть выведено сообщение об ошибке.

**11**

Пришло время написать главный метод класса **Handler**, который будет вызывать все вспомогательные методы, написанные ранее.  
Реализуем метод void run() в классе **Handler**.

**Он должен:**  
1. Выводить сообщение, что установлено новое соединение с удаленным адресом, который можно получить с помощью метода getRemoteSocketAddress().  
2. Создавать Connection, используя поле socket.  
3. Вызывать метод, реализующий рукопожатие с клиентом, сохраняя имя нового клиента.  
4. Рассылать всем участникам чата информацию об имени присоединившегося участника (сообщение с типом USER\_ADDED).  
Подумай, какой метод подойдет для этого лучше всего.  
5. Сообщать новому участнику о существующих участниках.  
6. Запускать главный цикл обработки сообщений сервером.  
7. Обеспечить закрытие соединения при возникновении исключения.  
8. Отловить все исключения типа **IOException** и **ClassNotFoundException**, вывести в консоль информацию, что произошла ошибка при обмене данными с удаленным адресом.  
9. После того как все исключения обработаны, если **п.11.3** отработал и возвратил нам имя, мы должны удалить запись для этого имени из connectionMap и разослать всем остальным участникам сообщение с типом USER\_REMOVED и сохраненным именем.  
10. Последнее, что нужно сделать в методе run() - вывести сообщение, информирующее что соединение с удаленным адресом закрыто.

Наш сервер полностью готов. Попробуй его запустить.

**Требования:**

* •

Метод run() должен выводить на экран сообщение о том, что было установлено соединение с удаленным адресом (используя метод getRemoteSocketAddress()).

* •

Метод run() должен создавать новое соединение (connection) используя в качестве параметра поле socket.

* •

Метод run() должен вызывать метод serverHandshake() используя в качестве параметра только что созданное соединение; результатом будет имя пользователя (userName).

* •

Метод run() должен вызывать метод sendBroadcastMessage() используя в качестве параметра новое сообщение (MessageType.USER\_ADDED, userName).

* •

Метод run() должен вызывать метод notifyUsers() используя в качестве параметров connection и userName.

* •

Метод run() должен вызывать метод serverMainLoop используя в качестве параметров connection и userName.

* •

Прежде чем завершиться, метод run() должен удалять из connectionMap запись соответствующую userName, и отправлять всем участникам чата сообщение о том, что текущий пользователь был удален.

* •

Метод run() должен корректно обрабатывать исключения IOException и ClassNotFoundException.

**12**

Приступим к ***написанию клиента***.

Клиент, в начале своей работы, должен запросить у пользователя адрес и порт сервера, подсоединиться к указанному адресу, получить запрос имени от сервера, спросить имя у пользователя, отправить имя пользователя серверу, дождаться принятия имени сервером.  
После этого клиент может обмениваться текстовыми сообщениями с сервером.  
Обмен сообщениями будет происходить в двух параллельно работающих потоках.  
Один будет заниматься чтением из консоли и отправкой прочитанного серверу, а второй поток будет получать данные от сервера и выводить их в консоль.

**Начнем реализацию клиента:**  
1) Создай пакет **client**. В дальнейшем все классы, отвечающие за реализацию клиентов, создавай в этом пакете.  
2) Создай класс **Client**.  
3) Создай внутренний класс **SocketThread** унаследованный от **Thread** в классе **Client**.  
Он будет отвечать за поток, устанавливающий сокетное соединение и читающий сообщения сервера.  
Класс должен иметь публичный модификатор доступа.  
4) Создай поле Connection connection в классе **Client**.  
Используй модификатор доступа, который позволит обращаться к этому полю из класса потомков, но запретит обращение из других классов вне пакета.  
5) Добавь приватное поле-флаг boolean clientConnected в класс **Client**.  
Проинициализируй его значением **false**.  
В дальнейшем оно будет устанавливаться в **true**, если клиент подсоединен к серверу или в false в противном случае.  
При объявлении этого поля используй ключевое слово, которое позволит гарантировать что каждый поток, использующий поле clientConnected, работает с актуальным, а не кэшированным его значением.

**Требования:**

* •

Класс Client должен быть создан в пакете client.

* •

В классе Client должен быть создан публичный класс SocketThread унаследованный от Thread.

* •

В классе Client должно быть создано protected поле connection типа Connection.

* •

В классе Client должно быть создано private volatile поле clientConnected типа boolean.

**13**

Продолжаем реализацию вспомогательных методов класса **Client**.

Добавь в класс методы, которые будут доступны классам потомкам, но не доступны из других классов вне пакета:  
1. String getServerAddress() - должен запросить ввод адреса сервера у пользователя и вернуть введенное значение.  
Адрес может быть строкой, содержащей **ip**, если клиент и сервер запущен на разных машинах или '***localhost***', если клиент и сервер работают на одной машине.  
2. int getServerPort() - должен запрашивать ввод порта сервера и возвращать его.  
3. String getUserName() - должен запрашивать и возвращать имя пользователя.  
4. boolean shouldSendTextFromConsole() - в данной реализации клиента всегда должен возвращать **true** (***мы всегда отправляем текст введенный в консоль***).  
Этот метод может быть переопределен, если мы будем писать какой-нибудь другой клиент, унаследованный от нашего, который не должен отправлять введенный в консоль текст.  
5. SocketThread getSocketThread() - должен создавать и возвращать новый объект класса **SocketThread**.  
6. void sendTextMessage(String text) - создает новое текстовое сообщение, используя переданный текст и отправляет его серверу через соединение connection.  
Если во время отправки произошло исключение **IOException**, то необходимо вывести информацию об этом пользователю и присвоить **false** полю clientConnected.

**Требования:**

* •

Метод getServerAddress() должен возвращать строку (адрес сервера), считанную с консоли.

* •

Метод getServerPort() должен возвращать число (порт сервера), считанное с консоли.

* •

Метод getUserName() должен возвращать строку (имя пользователя), считанную с консоли.

* •

Метод shouldSendTextFromConsole() должен возвращать true.

* •

Метод sendTextMessage() должен создавать и отправлять новое текстовое сообщение используя connection и устанавливать флаг clientConnected в false, если во время отправки или создания сообщения возникло исключение IOException.

* •

Метод getSocketThread() должен возвращать новый объект типа SocketThread.

**14**

Приступим к написанию главного функционала класса **Client**.

1. Добавь метод public void run().  
Он должен создавать вспомогательный поток **SocketThread**, ожидать пока тот установит соединение с сервером, а после этого в цикле считывать сообщения с консоли и отправлять их серверу.  
Условием выхода из цикла будет отключение клиента или ввод пользователем команды '***exit***'.  
Для информирования главного потока, что соединение установлено во вспомогательном потоке, используй методы wait() и notify() объекта класса **Client**.

Реализация метода run должна:  
а) Создавать новый сокетный поток с помощью метода getSocketThread().  
б) Помечать созданный поток как **daemon**, это нужно для того, чтобы при выходе из программы вспомогательный поток прервался автоматически.  
в) Запустить вспомогательный поток.  
г) Заставить текущий поток ожидать, пока он не получит нотификацию из другого потока.  
**Подсказка:** используй wait() и синхронизацию на уровне объекта.  
Если во время ожидания возникнет исключение, сообщи об этом пользователю и выйди из программы.  
д) После того, как поток дождался нотификации, проверь значение clientConnected.  
Если оно **true** - выведи "***Соединение установлено.  
Для выхода наберите команду 'exit'.***".  
Если оно **false** - выведи "***Произошла ошибка во время работы клиента."***.  
е) Считывай сообщения с консоли пока клиент подключен.  
Если будет введена команда '***exit***', то выйди из цикла.  
ж) После каждого считывания, если метод shouldSendTextFromConsole() возвращает **true**, отправь считанный текст с помощью метода sendTextMessage().

2. Добавь метод public static void main(String[] args).  
Он должен создавать новый объект типа **Client** и вызывать у него метод run().

**Требования:**

* •

Метод main() должен создавать новый объект типа Client и вызывать у него метод run().

* •

Метод run() должен создавать и запускать новый поток, полученный с помощью метода getSocketThread().

* •

Поток созданный с помощью метода getSocketThread() должен быть отмечен как демон (setDaemon(true)).

* •

После запуска нового socketThread метод run() должен ожидать до тех пор, пока не будет пробужден.

* •

До тех пор, пока флаг clientConnected равен true,с консоли должны считываться сообщения с помощью метода ConsoleHelper.readString().

* •

Если была введена команда "exit" цикл считывания сообщений должен быть прерван.

* •

Если метод shouldSentTextFromConsole() возвращает true, должен быть вызван метод sendTextMessage() со считанным текстом в качестве параметра.

**15**

Напишем реализацию класса **SocketThread**.  
Начнем с простых вспомогательных методов.

Добавь методы, которые будут доступны классам потомкам и не доступны остальным классам вне пакета:  
1) void processIncomingMessage(String **message**) - должен выводить текст message в консоль.  
2) void informAboutAddingNewUser(String **userName**) - должен выводить в консоль информацию о том, что участник с именем userName присоединился к чату.  
3) void informAboutDeletingNewUser(String **userName**) - должен выводить в консоль, что участник с именем userName покинул чат.  
4) void notifyConnectionStatusChanged(boolean **clientConnected**) - этот метод должен:  
а) Устанавливать значение поля clientConnected внешнего объекта Client в соответствии с переданным параметром.  
б) Оповещать (***пробуждать ожидающий***) основной поток класса **Client**.

**Подсказка:** *используй синхронизацию на уровне текущего объекта внешнего класса и метод*notify()*.  
Для класса***SocketThread***внешним классом является класс***Client***.*

**Требования:**

* •

Метод processIncomingMessage() должен выводить на экран сообщение полученное в качестве параметра.

* •

Метод informAboutAddingNewUser() должен выводить на экран сообщение о том что пользователь подключился к чату.

* •

Метод informAboutDeletingNewUser() должен выводить на экран сообщение о том что пользователь покинул чат.

* •

Метод notifyConnectionStatusChanged() должен устанавливать значение поля clientConnected внешнего объекта Client равным полученному параметру.

* •

Метод notifyConnectionStatusChanged() должен вызвать метод notify() на внешнем объекте Client.

**16**

Теперь все готово, чтобы дописать необходимые методы класса **SocketThread**.

1) Добавь **protected** метод clientHandshake() throws IOException, **ClassNotFoundException**.  
Этот метод будет представлять клиента серверу.

**Он должен:**  
а) В цикле получать сообщения, используя соединение connection.  
б) Если тип полученного сообщения NAME\_REQUEST (***сервер запросил имя***), запросить ввод имени пользователя с помощью метода getUserName(), создать новое сообщение с типом MessageType.USER\_NAME и введенным именем, отправить сообщение серверу.  
в) Если тип полученного сообщения MessageType.NAME\_ACCEPTED (***сервер принял имя***), значит сервер принял имя клиента, нужно об этом сообщить главному потоку, он этого очень ждет.  
Сделай это с помощью метода notifyConnectionStatusChanged(), передав в него **true**.  
После этого выйди из метода.  
г) Если пришло сообщение с каким-либо другим типом, кинь исключение IOException("*Unexpected MessageType*").

2) Добавь **protected** метод void clientMainLoop() throws IOException, **ClassNotFoundException**.

Этот метод будет реализовывать главный цикл обработки сообщений сервера. Внутри метода:  
а) Получи сообщение от сервера, используя соединение connection.  
б) Если это текстовое сообщение (***тип*** MessageType.TEXT), обработай его с помощью метода processIncomingMessage().  
в) Если это сообщение с типом **MessageType.USER\_ADDED**, обработай его с помощью метода informAboutAddingNewUser().  
г) Если это сообщение с типом MessageType.USER\_REMOVED, обработай его с помощью метода informAboutDeletingNewUser().  
д) Если клиент получил сообщение какого-либо другого типа, брось исключение IOException("Unexpected **MessageType**").  
е) Размести код из предыдущих пунктов внутри бесконечного цикла.  
Цикл будет завершен автоматически если произойдет ошибка (***будет брошено исключение***) или поток, в котором работает цикл, будет прерван.

**Требования:**

* •

Метод clientHandshake() должен отправлять новое сообщение (MessageType.USER\_NAME, getUserName()) используя connection, если тип принятого сообщения равен MessageType.NAME\_REQUEST.

* •

Метод clientHandshake() должен вызывать метод notifyConnectionStatusChanged(true) и завершаться, если тип принятого сообщения равен MessageType.NAME\_ACCEPTED.

* •

Метод clientHandshake() должен бросать исключение IOException, если тип принятого сообщения не MessageType.NAME\_ACCEPTED или не MessageType.NAME\_REQUEST.

* •

Метод clientHandshake() должен принимать сообщения от сервера до тех пор, пока тип сообщения равен MessageType.NAME\_REQUEST.

* •

Метод clientMainLoop() должен принимать сообщения от сервера до тех пор, пока тип сообщения равен MessageType.TEXT, MessageType.USER\_REMOVED или MessageType.USER\_ADDED.

* •

Метод clientMainLoop() должен обрабатывать полученное сообщение с помощью метода processIncomingMessage(), если тип принятого сообщения равен MessageType.TEXT.

* •

Метод clientMainLoop() должен обрабатывать полученное сообщение с помощью метода informAboutAddingNewUser(), если тип принятого сообщения равен MessageType.USER\_ADDED.

* •

Метод clientMainLoop() должен обрабатывать полученное сообщение с помощью метода informAboutDeletingNewUser(), если тип принятого сообщения равен MessageType.USER\_REMOVED.

* •

Метод clientMainLoop() должен бросать исключение IOException, если тип принятого сообщения не MessageType.TEXT, MessageType.USER\_REMOVED или не MessageType.USER\_ADDED.

**17**

Последний, но самый главный метод класса **SocketThread** - это метод void run().  
Добавь его. Его реализация с учетом уже созданных методов выглядит очень просто.

**Давай напишем ее:**  
1) Запроси адрес и порт сервера с помощью методов getServerAddress() и getServerPort().  
2) Создай новый объект класса***java.net.Socket***, используя данные, полученные в предыдущем пункте.  
3) Создай объект класса **Connection**, используя сокет из **п.17.2**.  
4) Вызови метод, реализующий "***рукопожатие***" клиента с сервером (clientHandshake()).  
5) Вызови метод, реализующий основной цикл обработки сообщений сервера.  
6) При возникновении исключений **IOException** или **ClassNotFoundException** сообщи главному потоку о проблеме, используя notifyConnectionStatusChanged() и **false** в качестве параметра.

Клиент готов, можешь запустить сервер, несколько клиентов и проверить как все работает.

**Требования:**

* •

В методе run() должно быть установлено и сохранено в поле connection соединение с сервером (для получения адреса сервера и порта используй методы getServerAddress() и getServerPort()).

* •

В методе run() должен быть вызван метод clientHandshake().

* •

В методе run() должен быть вызван метод clientMainLoop().

* •

При возникновении исключений IOException или ClassNotFoundException в процессе работы метода run(), должен быть вызван метод notifyConnectionStatusChanged() с параметром false.

* •

Заголовок метода run() должен соответствовать условию задачи.

**18 – пишем БОТ**

Иногда бывают моменты, что не находится достойного собеседника. Не общаться же с самим собой :)  
Давай напишем бота, который будет представлять собой клиента, который автоматически будет отвечать на некоторые команды.  
Проще всего реализовать бота, который сможет отправлять текущее время или дату, когда его кто-то об этом попросит.

**С него и начнем:**  
1) Создай новый класс **BotClient** в пакете client.  
Он должен быть унаследован от **Client**.  
2) В классе **BotClient** создай внутренний класс **BotSocketThread** унаследованный от **SocketThread**.  
Класс **BotSocketThread** должен быть публичным.  
3) **Переопредели методы:**  
а) getSocketThread(). Он должен создавать и возвращать объект класса **BotSocketThread**.  
б) shouldSendTextFromConsole(). Он должен всегда возвращать **false**.  
Мы не хотим, чтобы бот отправлял текст введенный в консоль.  
в) getUserName(), метод должен генерировать новое имя бота, например: **date\_bot\_X**, где X - любое число от **0** до **99**.  
Для генерации X используй метод Math.random().  
4) Добавь метод main(). Он должен создавать новый объект **BotClient** и вызывать у него метод run().

**Требования:**

* •

В пакете client должен быть создан класс BotClient являющийся потомком класса Client.

* •

В классе BotClient должен быть создан внутренний публичный класс BotSocketThread являющийся потомком класса SocketThread.

* •

В классе BotClient должен быть переопределен метод getSocketThread() возвращающий новый объект класса BotSocketThread.

* •

В классе BotClient должен быть переопределен метод shouldSendTextFromConsole() всегда возвращающий false.

* •

В классе BotClient должен быть переопределен метод getUserName() возвращающий имя бота по формату указанному в условии задачи.

* •

В методе main() класса BotClient должен быть создан новый объект класса BotClient и у него должен быть вызван метод run().

**19**

Сейчас будем реализовывать класс **BotSocketThread**, вернее переопределять некоторые его методы, весь основной функционал он уже унаследовал от **SocketThread**.

1. Переопредели метод clientMainLoop():  
а) С помощью метода sendTextMessage() отправь сообщение с текстом "***Привет чатику. Я бот. Понимаю команды: дата, день, месяц, год, время, час, минуты, секунды.***"  
б) Вызови реализацию clientMainLoop() родительского класса.

2. Переопредели метод processIncomingMessage(String **message**).  
Он должен следующим образом обрабатывать входящие сообщения:  
а) Вывести в консоль текст полученного сообщения message.  
б) Получить из message имя отправителя и текст сообщения. Они разделены "***:***".  
в) Отправить ответ в зависимости от текста принятого сообщения.  
**Если текст сообщения:**  
"***дата***" - отправить сообщение содержащее текущую дату в формате "***d.MM.YYYY***";  
"***день***" - в формате "***d***";  
"***месяц***" - "***MMMM***";  
"***год***" - "***YYYY***";  
"***время***" - "***H:mm:ss***";  
"***час***" - "***H***";  
"***минуты***" - "***m***";  
"***секунды***" - "***s***".  
Указанный выше формат используй для создания объекта SimpleDateFormat. Для получения текущей даты необходимо использовать класс **Calendar** и метод getTime().  
Ответ должен содержать имя клиента, который прислал запрос и ожидает ответ, например, если Боб отправил запрос "***время***", мы должны отправить ответ "***Информация для Боб: 12:30:47***".

Наш бот готов. Запусти сервер, запусти бота, обычного клиента и убедись, что все работает правильно.  
Помни, что message бывают разных типов и не всегда содержат "***:***"

**Требования:**

* •

Метод clientMainLoop()класса BotSocketThread должен вызывать метод sendTextMessage() у внешнего объекта BotClient c приветственным сообщением указанном в условии задачи.

* •

Метод clientMainLoop() класса BotSocketThread должен вызывать clientMainLoop() у объекта родительского класса (super).

* •

Метод processIncomingMessage() должен выводить на экран полученное сообщение message.

* •

Метод processIncomingMessage() должен отправлять ответ с помощью метода sendTextMessage() (форматирование согласно условию задачи).

* •

Метод processIncomingMessage() не должен вызывать метод sendTextMessage() для некорректных запросов.

**20**

Консольный клиент мы уже реализовали, чат бота тоже сделали, почему бы не сделать клиента с графическим интерфейсом? Он будет так же работать с нашим сервером, но иметь графическое окно, кнопки и т.д.

Итак, приступим. При написании графического клиента будет очень уместно воспользоваться паттерном **MVC** (*Model-View-Controller*).  
Ты уже должен был с ним сталкиваться, если необходимо, освежи свои знания про **MVC** с помощью Интернет.  
В нашей задаче самая простая реализация будет у класса, отвечающего за модель (*Model*).

**Давай напишем его:**  
1) Создай класс **ClientGuiModel** в пакете client.  
Все классы клиента должны быть созданы в этом пакете.  
2) Добавь в него множество(**set**) строк в качестве final поля allUserNames.  
В нем будет храниться список всех участников чата. Проинициализируй его.  
3) Добавь поле String newMessage, в котором будет храниться новое сообщение, которое получил клиент.  
4) Добавь **геттер** для allUserNames, запретив модифицировать возвращенное множество.  
Разберись, как это можно сделать с помощью метода класса **Collections**.  
5) Добавь **сеттер** и **геттер** для поля newMessage.  
6) Добавь метод void addUser(String **newUserName**), который должен добавлять имя участника во множество, хранящее всех участников.  
7) Добавь метод void deleteUser(String **userName**), который будет удалять имя участника из множества.

**Требования:**

* •

Класс ClientGuiModel должен быть создан в пакете client.

* •

Множество строк allUserNames должно быть инициализировано и объявлено с модификаторами private final.

* •

Приватное поле newMessage должно быть типа String.

* •

В классе ClientGuiModel должен быть создан корректный геттер для поля allUserNames.

* •

В классе ClientGuiModel должны быть созданы корректные геттер и сеттер для поля newMessage.

* •

Метод addUser() должен добавлять новое имя пользователя в множество allUserNames.

* •

Метод deleteUser() должен удалять полученное имя пользователя из множества allUserNames.

**21 - SWING**

У меня есть отличнейшая новость для тебя. Компонент представление (***View***) уже готов.  
Я добавил класс **ClientGuiView**.  
Он использует библиотеку ***javax.swing***.  
Ты должен как следует разобраться в каждой строчке этого класса.  
Если тебе все понятно - это замечательно, если нет - обязательно найди ответы на свои вопросы с помощью дебага, документации или поиска в Интернет.

Осталось написать компонент контроллер (Controller):  
1. Создай класс **ClientGuiController** унаследованный от **Client**.  
2. Создай и инициализируй поле, отвечающее за модель ClientGuiModel model.  
3. Создай и инициализируй поле, отвечающее за представление ClientGuiView view.  
Подумай, что нужно передать в конструктор при инициализации объекта.  
4. Добавь внутренний класс **GuiSocketThread** унаследованный от **SocketThread**.  
Класс **GuiSocketThread** должен быть публичным.  
В нем переопредели следующие методы:  
а) void processIncomingMessage(String **message**) - должен устанавливать новое сообщение у модели и вызывать обновление вывода сообщений у представления.  
б) void informAboutAddingNewUser(String **userName**) - должен добавлять нового пользователя в модель и вызывать обновление вывода пользователей у отображения.  
в) void informAboutDeletingNewUser(String **userName**) - должен удалять пользователя из модели и вызывать обновление вывода пользователей у отображения.  
г) void notifyConnectionStatusChanged(boolean **clientConnected**) - должен вызывать аналогичный метод у представления.  
5. Переопредели методы в классе **ClientGuiController**:  
а) SocketThread getSocketThread() - должен создавать и возвращать объект типа **GuiSocketThread**.  
б) void run() - должен получать объект SocketThread через метод getSocketThread() и вызывать у него метод run().  
Разберись, почему нет необходимости вызывать метод run() в отдельном потоке, как мы это делали для консольного клиента.  
в) getServerAddress(), getServerPort(), getUserName().  
Они должны вызывать одноименные методы из представления (***view***).  
6. Реализуй метод ClientGuiModel getModel(), который должен возвращать модель.  
7. Реализуй метод main(), который должен создавать новый объект ClientGuiController и вызывать у него метод run().  
Запусти клиента с графическим окном, нескольких консольных клиентов и убедись, что все работает корректно.

**Требования:**

* •

В пакете client должен быть корректно объявлен класс ClientGuiController и вложенный в него класс GuiSocketThread.

* •

В классе ClientGuiController должны быть корректно объявлены и инициализированы поля перечисленные в условии задачи.

* •

Метод processIncomingMessage() должен быть реализован в классе GuiSocketThread в соответствии с условием задачи.

* •

Метод informAboutAddingNewUser() должен быть реализован в классе GuiSocketThread в соответствии с условием задачи.

* •

Метод informAboutDeletingNewUser() должен быть реализован в классе GuiSocketThread в соответствии с условием задачи.

* •

Метод notifyConnectionStatusChanged() должен быть реализован в классе GuiSocketThread в соответствии с условием задачи.

* •

Метод getSocketThread() в классе ClientGuiController должен возвращать новый объект типа GuiSocketThread.

* •

Метод run() в классе ClientGuiController должен получать объект SocketThread через метод getSocketThread() и вызывать у него метод run().

* •

Методы getServerAddress(), getServerPort(), getUserName() в классе ClientGuiController должны вызывать соответствующие методы у объекта view.

* •

Метод getModel() в классе ClientGuiController должен возвращать значение поля model.

* •

Метод main() в классе ClientGuiController должен создавать новый объект типа ClientGuiController и вызывать у него метод run().

* •

Класс ClientGuiController должен быть потомком класса Client.

**22**

**Итак, подведем итог:**  
• Ты написал сервер для обмена текстовыми сообщениями.  
• Ты написал консольный клиент, который умеет подключаться к серверу и обмениваться сообщениями с другими участниками.  
• Ты написал бот клиента, который может принимать запросы и отправлять данные о текущей дате и времени.  
• Ты написал клиента для чата с графическим интерфейсом.

**Что можно добавить или улучшить:**  
• Можно добавить поддержку приватных сообщений (*когда сообщение отправляется не всем, а какому-то конкретному участнику*).  
• Можно расширить возможности бота, попробовать научить его отвечать на простейшие вопросы или время от времени отправлять шутки.  
• Добавить возможность пересылки файлов между пользователями.  
• Добавить контекстное меню в графический клиент, например, для отправки приватного сообщения кому-то из списка участников.  
• Добавить раскраску сообщений в графическом клиенте в зависимости от отправителя.  
• Добавить блокировку сервером участников за что-либо, например, ненормативную лексику в сообщениях.  
• Добавить еще миллион фич и полезностей!

**Ты научился:**  
• Работать с сокетами.  
• Пользоваться сериализацией и десериализацией.  
• Создавать многопоточные приложения, синхронизировать их, применять модификатор volatile, пользоваться классами из библиотеки ***java.util.concurrent***.  
• Применять паттерн **MVC**.  
• Использовать внутренние и вложенные классы.  
• Работать с библиотекой **Swing**.  
• Применять классы **Calendar** и **SimpleDateFormat**.

**Так держать!**

**Требования:**

* •

Поздравляю, чат готов!