Телеграм-бот - это виртуальный пользователь Telegram, с которым можно “общаться” как с живым человеком. Перед нами – очень интересная задача. Мы должны создать “умного” бота, который умеет:

вести диалог

практиковаться в свиданиях

генерировать адекватные и прикольные описания для сайтов знакомств

раздавать советы при переписке на сайте знакомств.

Чтобы бот этому всему научился и выглядел умным, мы прикрутим к нему ChatGPT.

Предварительная подготовка к работе

Прежде, чем начинать программировать, нам нужно подготовить рабочее место. Для этого нам понадобится:

Сама Java (Java Development Kit)

Приложение для написания и запуска программ, IntelliJ IDEA Community Edition

Шаблон нашего проекта

Telegram, желательно для удобства загрузить десктнопную версию, по крайней мере для первой лекции.

Установка IntelliJ IDEA и загрузка проекта

Загрузить бесплатную версию IntelliJ IDEA Community Edition можно с сайта разработчика для , или . Шаблон проекта желательно разархивировать и запомнить, где лежит папка. Если уже загружали, найдите, где именно она хранится. Откройте IntelliJ IDEA. Если это первый запуск, скорее всего, вы увидите окно с предложением создать проект, открыть его либо загрузить из системы контроля версий. Если вы уже работали в IntelliJ IDEA и у вас открыт другой проект, жмите на Windows/Linux File - Open, MacOS Intellij IDEA - Open Далее в обоих случаях отыскиваем папку TinderBolt там, где вы её сохранили. И снова жмем Open. Перед вами откроется шаблон проекта. Слева вы увидите дерево проекта, справа же будет показан активный файл. Дерево проекта состоит из различных директорий и файлов. Нужные нам файлы с Java-кодом лежат по пути TinderBolt—> src → main → Java → com → javarush → telegram. Если вы никогда не программировали, вероятно, структура проекта да и сам код в файлах покажется вам чем-то непонятным и слишком сложным. На самом деле понимание всего этого просто требует практики. Пока что мы только собираемся учиться на Java Junior, а “джуны” обычно не создают структуру проекта, а лишь пишут отдельные его части. Так что в течение следующих занятий мы с вами сымитируем работу начинающего программиста на проекте, шаблон которого создал какой-то senior или архитектор.

Немного теории перед первой лекцией

Java-программа или Java-проект В самом общем смысле программа – это последовательность команд, которые понимает и выполняет компьютер. Команды эти пишет программист, в нашем случае – на языке Java. Чтобы программистам было проще работать, у Java-проекта есть определенная структура. На физическом уровне это файлы и папки, расположенные в определенной иерархии, как вы могли видеть в открытом проекте TinderBolt. Основное количество файлов в типичном Java-проекте – это файлы с Java-кодом, но также есть файлы с ресурсами (например, текстовыми файлами или графическими аватарками), конфигурационные файлы и так далее. В простейшем случае программа может состоять из одного файла с Java-кодом. Java, установленная на компьютер, сможет перевести этот код на понятный компьютеру язык (откомпилировать), а затем и запустить программу. Пример такой простейшей программы:

public class Main {

public static void main(String[] args) {

System.out.println("Это моя программа");

}

}

Вывод в консоль

Это моя программа

Здесь может быть много непонятных для новичка слов. Некоторые из них мы пока что проигнорируем и остановимся на самом важном. Такая незамысловатая программа выводит на экран (в консоль) сообщение “Это моя программа”. Делает она это с помощью команды

System.out.println("Это моя программа");

Каждая команда в Java заканчивается точкой с запятой ;. А логический блок кода ограничен фигурными скобками { }.

int age = 35;

if (age < 40) { // начало логического блока.

System.out.println("Ты еще молод"); // команда

} // конец логического блока.

/\*

В данном примере код проверяет,

является ли человек, чей возраст хранится в переменной age, моложе 40 лет.

Если да, то код выводит сообщение "Ты еще молод" в консоль.

\*/

В коде выше после символа // начинается комментарий длиной в одну строку. А многострочный комментарий можно написать /\* между наклонными слешами со звездочкой, вот так, как мы сейчас это делаем \*/. Комментарии можно оставлять для коллег, которые будут читать ваш код или же для вас, чтобы не забыть чего. Суть в том, что в программе комментарии не выполняются, они здесь только чтобы что-то пояснить и в них можно писать что угодно. Но желательно, чтобы они несли какой-то смысл. Что ещё? Оператор = это не равенство, а присвоение.

String namePhone = "Mi 11";

значит, что переменной namePhone присвоили значение Mi 11. За равенство в Java отвечает оператор ==. Не обязательно к установке Когда много скобок и других подобных знаков можно легко запутаться. И кто-то уже придумал плагин Rainbow Brackets. Он выделяет разным цветом скобки и другие знаки, облегчая читабельность кода. 1. Зайдите в раздел "Настройки", Windows/Linux File - Settings, MacOS Intellij IDEA - Preferences 2. В появившемся окне в боковом меню выберите раздел Plugins и откройте вкладку Marketplace. И в строке поиска напиши  Rainbow Brackets 3. После установки нажать Restar IDE

**Дополнение и краткий конспект к занятию №1**

Любой Java-проект состоит из каких-то файлов и папок. Это могут быть файлы с Java-кодом, файлы с конфигурациями, ресурсы для работы нашей программы, например, картинки, которые она будет загружать или текстовые файлы, информацию из которых она будет тем или иным образом считывать. Давайте поговорим о структуре нашего проекта, точнее о самых главных его файлах. Они, Java-классы, лежат в папке проекта по пути src—>main—> java —> com —>javarush—>telegram. В папке resources лежат картинки, текстовые сообщения и промпты для chatGPT. Тут есть еще конфигурационный файл pom.xml, какой-то .gitignore и много всего непонятного новичку. Давайте от них абстрагируемся и вернемся к нашим Java-файлам. Самый важный из них, в котором будет вестись основная работа — TinderBoltApp. Практически весь код мы будем писать в этом классе, в методе основной функциональности бота onUpdateEventReceived(Update update). Остальные классы:

UserInfo — для хранения профилей людей в программе для знакомств

ChatGPTService — класс для упрощения работы с искусственным интеллектом.

DialogMode — содержит режимы работы с ботом (например «тренировка в переписке» или «создание профайла» и так далее).

MultiSessionTelegramBot — для упрощения работы нашей программы с API-серверами Telegram.

API — это аббревиатура от "Application Programming Interface", что переводится как "интерфейс программирования приложений". Это набор правил и протоколов для создания и интеграции программного обеспечения. То есть API позволяет программам общаться между собой, а программистам – использовать возможности других программ в своих.

Регистрация бота и получение логина, токена и ссылки на бот

В нашем классе TinderBoltApp есть переменные:

TELEGRAM\_BOT\_NAME

TELEGRAM\_BOT\_TOKEN

OPEN\_AI\_TOCKEN

Первые два токена – это логин и пароль, а о третьем поговорим на следующем уроке. Чтобы нам получить значение логина и токена(пароля), нам нужно зарегистрировать нашего бота на сервере Telegram:

Заходим в Telegram

В поиске ищем бота по имени BotFather (этакий папа всех ботов).

Пишем команду /start

Жмем на /newbot

Выбираем имя для нашего бота

Теперь выбираем имя пользователя (username) для своего бота. Имя должно быть уникальным и заканчиваться на «bot».

Если все ок, бот нам напишет приветственное сообщение, в котором будет наше имя пользователя, а в качестве токена можно использовать выданный ботом токен для HTTP API, а также ссылка на нашего бота.

Теперь возвращаемся в IntelliJ IDEA, в класс TinderBoltApp и именуем переменные

TELEGRAM\_BOT\_NAME = «your name»

TELEGRAM\_BOT\_TOKEN = «your HTTP API token»

Наш бот пока ничего не умеет, но он уже работает. Чтобы запустить программу в IntelliJ IDEA нужно щелкнуть по зеленой стрелочке справа сверху окна ▶ (при этом в самом окне желательно чтобы был открыт TinderBoltApp. Либо же кликните по имени класса TinderBoltApp в дереве проектов правой кнопкой мыши и найдите там такую же стрелочку с надписью Run TinderBoltApp.main(). Так запускается любая программа на java. По сути мы запустили метод main класса TinderBolt. Теперь, когда программа запущена, мы можем возвращаться в телеграмм: наш бот работает. Кликаем по ссылке, полученной в приветственном сообщении и мы у бота в гостях. Мы можем нажать на кнопку Start или написать /start (это кнопка начала общения с ботом). Правда, говорить он еще не умеет, поэтому ничего не произойдет. Поэтому просто остановим программу и перейдем к написанию кода. Как мы общаемся с ботом или живым человеком? Мы получаем его сообщения и пишем свои. Чаще всего это текстовые сообщения, но также это могут быть картинки, а в случае бота — еще и команды. Как мы уже писали выше, для общения программ между собой есть специальные пакеты API. Разумеется, программисты уже написали API для телеграмм-ботов на Java, и наш ментор их использовал, чтобы создать методы для общения именно с нашим ботом. В принципе, это могли сделать и мы, но тогда наша работа усложнилась бы в разы, а мы только начинаем учиться, нам и так сложно. Сделать так, чтобы бот нам что-то написал несложно. Давайте вызовем этот метод в главном методе onUpdateEventReceived нашего основного класса TinderBoltAPI. Для этого в этом методе нужно написать следующую команду:

sendTextMessage("Привет!");

Теперь если мы запустим бот, он будет реагировать «приветом» на любое посланное нами сообщение.

Сообщения с кнопками

Чтобы отправить сообщение с кнопкой, можно использовать метод sendTextButtonMessage

sendTextButtonMessage(

"Выберите режим работы", // Текст перед кнопкой

"Старт", "btn\_start", // Текст и команда кнопки "Старт"

"Стоп", "btn\_stop" // Текст и команда кнопки "Стоп"

);

Теперь если мы запустим бота у нас отобразится две кнопки Наконец, отправлять в чат картинки можно с помощью метода sendPhotoMessage

sendPhotoMessage("image\_name");

Ему можно отправлять картинки по ключу, а сами картинки лежат в папке resources —> pictures

Работа с командами Telegram

Команды в Telegram начинаются с / косой черты. Собственно, с одной из них мы уже работали, это команда /start. Давайте попробуем заставить нашу программу отреагировать на команду /start. Например, если пользователь прислал команду старт, то ему в ответ прислать картинку. А в других случаях бот картинку слать не должен. Для таких вот условий в командах в Java используется оператор if.

if (условие) {

Какие-то команды;

}

В нашем случае код условия будет выглядеть так:

// Проверка, начинается ли сообщение пользователя с команды "/start"

if (message.startsWith("/start")) {

// Загрузка сообщения с названием "main"

String text = loadMessage("main");

// Отправка фотографии с именем "main"

sendPhotoMessage("main");

// Отправка текстового сообщения с текстом из переменной "text"

sendTextMessage(text);

// Завершение обработки команды "/start"

return;

}

Здесь написано буквально следующее: если сообщение пользователя начинается с команды /start, тогда мы отсылаем картинку с именем main и текст с именем main. И текст, и картинка расположены в соответствующих папках проекта в resources. Return обычно возвращает значение метода, а потом завершает работу метода. Если же мы применим return без значения не на выходе из метода, а на выходе из какого-либо блока кода, например, условного оператора, как здесь, он просто закончит обработку нашего сообщения и следующие команды не будут исполнятся.

**Что нужно сделать перед и во время первой лекции**

1. Установить все необходимое для работы. 2. Послушать лекцию и повторить все за ментором По итогу у нас должен быть бот, который

Имеет имя, название и ссылку на себя

Запускается

Считывает сообщение пользователя

Отсылает картинку и описание своей работы, если пользователь послал ему команду /start

Отсылает простое текстовое сообщение

Не все может быть понятно, но важно не запутаться и все сделать, повторить за ментором. Некоторые моменты станут понятны позднее.

Подсказки к выполнению задания №1

Проверка корректности кода и обновление изменений в проекте

Чтобы увидеть изменения, которые ты вносишь непосредственно в работе Телеграм-бота, не забудь после написания своего кода применить все изменения (перезапустить проект) в IntelliJ IDEA. Для этого нужно в нижней горизонтальной панели нажать на зеленую круглую стрелку:

Важные принципы форматирования кода

Помни о том, что твои названия переменных, методов и классов должны быть читабельными: по названию должно быть понятно, что они делают или для чего предназначены.

Форматирование текста

Чтобы выделить текст курсивом, нужно в начале и в конце необходимого фрагмента добавить нижние подчеркивания: \_текст\_

Чтобы выделить текст жирным, нужно в начале и в конце необходимого фрагмента добавить звездочки: \*текст\*

**Дополнение и краткий конспект к занятию №2**

В этой лекции наш бот станет умнее, потому что мы прикрутим к нему искусственный интеллект, а именно — chatGPT. А работать с нашей программой всеми любимый чатик будет с помощью… правильно, специального API. То есть наш бот будет переключаться в режим работы с чатом и все последующие сообщения пользователя уже будут обрабатываться именно им.

Что для этого нужно

Для начала нам понадобится токен chatGPT, это ключ, с помощью которого наш бот сможет общаться с сервером chatGPT. Его можно купить или достать каким-то окольным путем, но вам повезло, поскольку перед этой лекцией вам его уже прислали в бот. Первое наше действие с кодом в этом уроке — скопировать значение присланного токена и вставить его в соответствующее поле.

public static final String OPEN\_AI\_TOKEN = «вставьте сюда ваш ключ";

В списке команд чата есть следующая: /gpt. Согласно описанию в боте, эта команда должна задать вопрос нашему чатику. Теперь нам нужно создать кусок кода, который будет обрабатывать эту команду, Обратите внимание на нюанс, если вы нажали на какую-то команду( или ввели ее), это уже есть сообщение для бота. А вопрос чату уже будет следующей командой. Поэтому нужно обработать два сообщения. По аналогии с обработкой команды /start пишем код, который означает буквально следующее: если сообщение пользователя начинается со строки /gpt, наш бот переключается в режим диалога с чатом. Можно предложить пользователю задать вопрос ИИ, а еще — для наглядности прислать картинку и закончит работу с сообщением.

if (message.startsWith("/gpt")) {

currentMode = DialogMode.GPT;

sendPhotoMessage("gpt");

sendTextMessage("Напишите любое сообщение и посмотрим, что вам ответит \*ChatGPT\*:");

return;

}

Что такое Dialog.Mode?

Это варианты работы бота, те самые, что указаны все они описаны в одноименном классе и потом выводятся в приветственном сообщении. Заходите в него и там вы обнаружите что-то необычное.

public enum DialogMode {

MAIN,

PROFILE,

OPENER,

MESSAGE,

DATE,

GPT,

}

Вместо class – enum. На самом деле перед нами довольно простая штука и называется она перечислением. Это такой специальный класс, который преставляет группу констант (неизменяемых переменных). Их удобно использовать для представления фиксированного числа связанных значений, например, дней недели или уровней доступа. Тут у нас идет перечисление режимов. В режимы при общении с телеграм-ботами мы входим выполняя команды, которые начинаются со слеша. Мы уже реализовали обработку команды /start. По сути наша главная задача – реализовать код, который бы работал со всеми шестью режимами. Режим /profile, /opener, /message, /date, /gpt. В нашем ENUM как раз есть для них заготовки. То есть когда пользователь входит в определенный режим, наш бот должен понимать, что мы в него вошли и реагировать соответствующе. В этой же лекции наша задача – подготовить блок кода, в котором после переключения в режим работы с чатом GPT (/gpt) он мог отвечать на вопросы пользователя. Для этого нужно написать блок кода с условным оператором if. А чтобы общение было более осмысленным и интересным, мы использовали заранее подготовленные промпты. Они лежат в папочке resources\prompts. Промпты – это хорошо сформулированные запросы к ИИ. У нас они лежат в папке с ресурсами. Также в этой лекции мы рассмотрели метод добавления меню и списка команд. Для этого есть метод showMainMenu(). В нашем случае, если мы включаем только команды под уже реализованные функции код будет каким-то таким:

1. /start — главное меню бота

2. /profile — генерация Tinder-профля 😎

3. /opener — сообщение для знакомства 🥰

4. /message — переписка от вашего имени 😈

5. /date — переписка со звездами 🔥

6. /gpt - задать вопрос чату GPT 🧠

Все смайлики в программе отображаются с помощью специальных последовательностей символов. Таким образом к концу этой лекции мы можем общаться с чатом тремя способами: нажать на команду в приветственном сообщении ввести команду в формате /команда кликнуть по меню и выбрать соответствующую команду. Итого на этой лекции мы:

прикрутили ChatGPT

добавили обработку

добавили переключение в режим диалога

разобрались как работать с промптами

изображение меню

**Дополнение и краткий конспект к занятию №3**

Подключаем режим дейтинга /date

Наша задача заключается в том, чтобы пользователь после переключения в режим дейтинга после команды /date смог открыть меню и выбрать там соответствующую звезду. Затем нам нужно научить бота отвечать от имени конкретной звезды, то есть скормить ему промпт, в котором объяснить, с какой звездой он будет общаться, а еще – прописать команды так, чтобы он реагировал на все сообщения соответствующим образом, а не только на одно из них. То есть мы должны добавить запросы к чату с сохранением истории. На первом занятии мы уже разбирали, как создавать кнопки для меню с текстовыми сообщениями. Чтобы создать меню с такими кнопками мы можем использовать метод sendTextButtonsMessage После этого на лекции мы начали писать обработку для каждой кнопки. Сначала мы научились высылать фото выбранной звезды, а потом создали сам тренажер для приглашения на свидания. Для того, чтобы создать тренажер мы сделали несколько действий:

Передали программе промпт, который описывает общение с соответствующей звездой (все такие промпты лежат в папке resources).

Перевели chatGPT из режима вопрос-ответ в режим диалога.

Установка промпта:

String prompt = loadPrompt(query);

chatGPT.setPrompt(prompt);

Обработка команды /message

В этом пункте мы добавили функциональность, которая позволяет chatGPT генерировать удачные ответы на нашу переписку. Для накопления сообщений мы использовали переменную типа ArrayList() или список. ArrayList в Java — это как специальный список или блокнот, где ты можешь записывать вещи и всегда добавлять новые записи в конец или стирать старые. Ты можешь легко посмотреть, что находится на любой странице списка, просто зная номер этой страницы. Если тебе нужно что-то добавить и в списке нет места, он сам сделает себя больше, чтобы все поместилось. Ты также можешь поставить несколько одинаковых вещей подряд или даже оставить пустую страницу (это как null в Java), и это будет нормально для этого списка. Единственное условие – записи в таком списке должны быть однородные. То есть один список может содержать только строки или только числа или только собачек из класса Dog. Представим, что у вас есть список любимых мультфильмов, и вы хотите его организовать. Вот как вы можешь это сделать с помощью ArrayList:

import java.util.ArrayList; // Подключаем возможность использовать ArrayList

public class Main {

public static void main(String[] args) {

// Создаем новый список для мультфильмов

ArrayList<String> favoriteCartoons = new ArrayList<>();

// Добавляем мультфильмы в список

favoriteCartoons.add("Том и Джерри");

favoriteCartoons.add("Спанч Боб");

favoriteCartoons.add("Гравити Фолз");

// Теперь предположим, что ты хочешь добавить еще один мультфильм

favoriteCartoons.add("Аватар: Последний маг воздуха");

// И можешь даже удалить мультфильм, который тебе больше не нравится

favoriteCartoons.remove("Спанч Боб"); // Удаляем "Спанч Боб" из списка

}

}

}

Точно так мы используем в нашем чатботе список для хранения сообщений И по аналогии с предыдущим пунктом мы написали обработчик нажатия на кнопки.

Метод join()

Метод String.join() в Java позволяет объединить (склеить) несколько строк в одну, используя разделитель. Очень удобен, когда нужно соединить элементы списка или массива в одну строку с заданным разделителем. Синтаксис:

String.join(delimiter, elements);

delimiter — строка, которая будет использоваться в качестве разделителя между элементами.

elements — элементы, которые нужно объединить. Могут быть переданы как массив или как последовательность строк.

Пример:

String result = String.join("-", "2024", "01", "01");

System.out.println(result); // Выводит: 2024-01-01

Обновление сообщений

У вас бывало, что вы делали ошибку в сообщении и потом ее исправляли? Это может делать и наш бот с помощью метода updateTextMessage(message, text). В нашем случае мы заменяем сообщение перед отправкой сообщения GPT, а потом меняем это сообщение на ответ самого чата. Например:

// Отправляем сообщение в чат с текстом "ChatGPT думает над вариантами ответа..."

var sm = sendTextMessage("ChatGPT думает над вариантами ответа...");

// Отправляем запрос в ChatGPT, передавая ему prompt и историю чата с пользователем

String answer = chatGPT.sendMessage(prompt, userChatHistory);

// Обновляем ранее отправленное сообщение (sm) текстом ответа от ChatGPT

updateTextMessage(sm, answer);

**Дополнение и краткий конспект к занятию №4**

На этой лекции мы реализуем две функции:

profile - позволяет генерировать профиль для своего тиндера

opener - генерация первого сообщения незнакомому человеку

Режим profile. Создание профиля пользователя

Как и для всех предыдущих команд мы реализуем ее через два условных оператора if. И как всегда нам первым делом нужно проверить, не вошел ли пользователь в соответствующий режим, в нашем случае – в режим профиля с помощью команды /profile. Реакцией на ввод пользователем этой команды должен быть перевод работы бота в этот диалог с помощью такой строки:

currentMode = DialogMode.PROFILE;

После него – как обычно демонстрация открывающей картинки и какое-то текстовое сообщение, типичное для режима. В данном случае пользователю предлагают написать информацию о себе. Второй блок с оператором if как раз проверяет, находится ли пользователь в режиме DialogMode.PROFILE, и если это так, то отправленные пользователем сообщение (где он написал что-то о себе) отсылаются в ChatGPT вместе с просьбой сгененировать на основе этой информации профиль в Tinder. Мы могли бы просто попросить пользователя описать себя, но чтобы информация была точнее, и профиль получился более интересным мы будем использовать класс UserInfo и счетчик вопросов. По ходу написания этого блока мы создали объект класса UserInfo (это один из написанных ментором классов, который можно видеть в дереве структуры класса) и нам понадобилась возможность работать с объектами как со строками. В этом блоке мы столкнулись с новым для нас оператором switch. Он используется для выполнения различных действий на основе разных условий. Это хорошая альтернатива цепочке if-else при проверке одной и той же переменной на равенство с множеством значений.

int temperature = 36;

switch(temperature)

{

case 36:

System.out.println("Низкая");

case 37:

System.out.println("Нормальная");

case 38:

System.out.println("Высокая");

}

При значении температуры 36, программа перейдет внутри switch на первый блок кода — первую строку, и после ее выполнения просто будут выполняться остальные блоки кода. Но вернемся к нашей теме. В результате мы создали блок кода, в котором пользователь переходит в режим профиля, и ему задают несколько ключевых вопросов. Ответы на эти вопросы все вместе отправляются чату GPT, который получил промпт о создании профиля на основе ответов пользователя. Собственно, это ChatGPT и делает, создает классный профайл на основе ваших данных. Также, как и в прошлый раз мы используем связку sendMessage + UpdateMessage: сначала шлем предварительное сообщение, потом ChatGPT обрабатывает наш запрос, и когда его ответ готов, предварительное сообщение меняется на его ответ.

// Отправка сообщения пользователю с информацией о том, что ChatGPT работает над ответом

Message message = sendTextMessage("ChatGPT думает... Подождите пару секунд");

// Отправка запроса в ChatGPT с использованием заданного prompt и информации о пользователе

String answer = chatGPT.sendMessage(prompt, userInfo);

// Обновление ранее отправленного сообщения ответом от ChatGPT

updateTextMessage(message, answer);

Усложняем условие проверки нахождения в том или ином режиме

Если мы по ошибке зашли не в тот режим, а программа уже начала работу в нем, например, как в случае Opener или Profiler, она начала собирать ответы на вопросы, то последующее нажатие другой команды будет воспринято как ответ на вопрос. Как это исправить? Нужно немного изменить проверку, добавив в нее дополнительное условие. Это условие заключается в том, что следующее сообщение не является командой. Это можно сделать так:

if (currentMode == DialogMode.GPT && !isMessageCommand()) {

//код

}

Давайте разберемся, что тут написано. Сначала, как и раньше, мы проверяем, не находимся ли мы в каком-то режиме (в данном случае – в режиме GPT). Длее два символа &&. Это оператор “и”. Он объединяет два условия в одно, должно выполняться и условие слева от &&, и условие справа от &&. Оператор ! означает отрицание того, что идет за ним. В данном случае оператор отрицает то, что возвращает метод isMessageCommand(). Этот метод из нашего класса проверяет, является ли сообщение командой (/start, /GPT и т д). Короче говоря наше условие проверяет, вошли ли мы в режим GPT и при этом сообщение пользователя не является командой и только в таком случае мы выполняем код, который идет дальше. После такой правки если пользователь введет команду, пользователя переключат в другой режим. Теперь мы реализовали все.