| Кулагин Максим Вадимович  mvkulagin@etu.ru |
| --- |
| Общие требования к практическим работам:Выполнять можно одному или в бригаде до 2 чел.Код должен быть в вашем репозитории GithubМожно писать на любом объектно-ориентированном языке (Kotlin, Java, Python, C#, …)Требуется соблюдение [принципов ООП](https://javarush.ru/groups/posts/1966-principih-obhhektno-orientirovannogo-programmirovanija)Соблюдение [code style](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82_%D0%BE%D1%84%D0%BE%D1%80%D0%BC%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%BA%D0%BE%D0%B4%D0%B0) ([документация Kotlin](https://kotlinlang.org/docs/coding-conventions.html), [документация Java](https://google.github.io/styleguide/javaguide.html)) |
| Практическое задание 1.  Консольная программа для поиска в Википедии.  **Дедлайн - 6 октября**  Описание задания  Напишите программу, которая с консоли считывает поисковый запрос пользователя, и выводит результат поиска по Википедии. После выбора нужной статьи программа должна открывать ее в браузере. Программа должна реагировать корректно на любой пользовательский ввод.  Задача разбивается на 5 этапов:   1. Считать введенные пользователем данные 2. Сделать запрос к серверу 3. Распарсить ответ 4. Вывести результат поиска 5. Открыть нужную страницу в браузере   Использовать готовые библиотеки для работы с Википедией нельзя.  Генерация запроса  API предоставляет возможность делать поисковые запросы без ключей авторизации. Вот таким образом:  https://ru.wikipedia.org/w/api.php?action=query&list=search&utf8=&format=json&srsearch="ЛЭТИ"  Вы можете открыть эту ссылку в браузере, и посмотреть на результат запроса.  Однако, чтобы запрос прошел удачно, следует убрать из ссылки недопустимые символы, то есть сделать [Percent-encoding](https://en.wikipedia.org/wiki/Percent-encoding), он же URL Encoding.  Для этого в Kotlin можно воспользоваться статическим методом *encode* в классе [URLEncoder](https://docs.oracle.com/javase/10/docs/api/java/net/URLEncoder.html), вот так:  street = URLEncoder.encode(street, "UTF-8")  Подготовка к обработке ответа  Сервер возвращает данные в формате [JSON](https://en.wikipedia.org/wiki/JSON).  Но нам его не надо парсить вручную, для этого есть библиотека [Gson](https://github.com/google/gson) от Google.  Примеры есть тут:  <https://github.com/google/gson>  Открытие статьи в браузере  После получения результатов поиска из Википедии нам будет известен pageid каждой статьи. По нему можно получить ссылку на статью следующим образом:  https://ru.wikipedia.org/w/index.php?curid=11099  Для открытия страницы в браузере можно воспользоваться, например, Java-классом [Desktop](https://docs.oracle.com/javase/9/docs/api/java/awt/Desktop.html). |
| Практическое задание 2.  Консольная программа для работы с CSV и XML файлами.  **Дедлайн - 10 ноября**  Описание задания  Даны 2 файла-справочника городов ([файлы во вложении](https://drive.google.com/drive/u/0/folders/1WAj235ic5dGXNsZ3g-Zp5pQXkFGjU0TY)). Один файл в формате xml, другой в формате csv.  Необходимо разработать консольное приложение для работы с ними.    address.xml    address.csv  После запуска приложение ожидает ввода пути до файла-справочника либо команды на завершение работы (какая-то комбинация клавиш).  По команде завершения работы приложение завершает свою работу.  После ввода пути до файла-справочника приложение формирует сводную статистику:  1) Отображает дублирующиеся записи с количеством повторений.  2) Отображает, сколько в каждом городе: 1, 2, 3, 4 и 5 этажных зданий.  3) Показывает время обработки файла.  После вывода статистики приложение снова ожидает ввода пути до файла-справочника либо команды на завершение работы.  В процессе работы приложение падать не должно, выход только по команде на завершение работы.  Критерии оценки  1) Качество и читаемость кода  2) Скорость работы приложения  Возможные варианты реализации  Работа с [hashmap](https://kotlins.org/maps#runtime) будет намного быстрее, чем с массивами.  Различные способы работы с CSV есть [тут](https://www.baeldung.com/kotlin/csv-files).  Пример для работы с XML [тут](https://www.javatpoint.com/kotlin-android-xml-parsing-using-dom-parser). |
| Практическое задание 3.  Создание бота для Telegram.  **Дедлайн - 15 декабря**  Описание задания  В рамках лабораторной работы требуется реализовать программу для взаимодействия с пользователями в Telegram.  Вариант 1 - Бот с расписанием ЛЭТИ  Целью всей работы является создание бота, который бы позволил получить расписание занятий для любой группы. Бот должен понимать следующие команды:   * near\_lesson GROUP\_NUMBER - ближайшее занятие для указанной группы; * DAY WEEK\_NUMBER GROUP\_NUMBER - расписание занятий в указанный день (monday, thuesday, ...). Неделя может быть четной, нечетной; * tommorow GROUP\_NUMBER - расписание на следующий день (если это воскресенье, то выводится расписание на понедельник, учитывая, что неделя может быть четной или нечетной); * all WEEK\_NUMBER GROUP\_NUMBER - расписание на всю неделю.   Для работы с расписанием необходимо использовать API университета:  <https://digital.etu.ru/api/docs-public/>  Вариант 2 - Своя идея реализации бота  Вы можете реализовать свою идею для бота, например, бот для кино, погоды и прочего. Написанный бот должен иметь развитую логику работы и несколько различных команд.  Регистрация бота  Чтобы зарегистрировать нового бота в телеграмме, вам нужно выполнить несколько простых шагов, которые подробно описаны в этом [руководстве](https://core.telegram.org/bots). Ниже приведен пример регистрации бота.    Для создания бота необходимо использовать команду /newbot. [@BotFather](https://t.me/botfather) запросит имя бота и имя пользователя бота (логин), а затем сгенерирует токен авторизации для вашего нового бота.  Имя бота отображается в контактных данных и в других местах.  Имя пользователя бота - это логин бота и представляет из себя короткое имя, которое будет использоваться в упоминаниях и ссылках на t.me. Обратите внимание, что в дальнейшем имя пользователя изменить нельзя! Имена пользователей (логины) состоят из 5–32 символов и нечувствительны к регистру, и могут включать только латинские символы, числа и символы подчеркивания. Имя пользователя бота должно заканчиваться на bot, например tetris\_bot или TetrisBot.  Токен представляет собой строку типа 110201543:AAHdqTcvCH1vGWJxfSeofSAs0K5PALDsaw, которая требуется для авторизации бота и отправки запросов в [API бота](https://core.telegram.org/bots/api). Держите свой токен в безопасности и храните его, он может быть использован кем угодно для управления вашим ботом.  Пересоздание токена авторизации Telegram бота  Если существующий токен скомпрометирован или был утерян по какой-либо причине, то используйте команду /token для создания нового.  Библиотеки для работы с API Telegram  Вы можете использовать API напрямую, либо использовать готовую библиотеку, что существенно упростит написание кода.  Список библиотек для всех основных языков представлен [тут](https://core.telegram.org/bots/samples).  Для Kotlin рекомендую использовать библиотеку [TelegramBotAPI](https://github.com/InsanusMokrassar/TelegramBotAPI), либо Spring Boot.  Вариант создания бота на Spring Boot  За основу нашего чат-бота возьмём Spring Boot. Код будем писать на Kotlin. Воспользуемся сайтом [Spring Initializr](https://start.spring.io/) для создания заготовки нашего приложения. В настройках выберем Gradle Project и Kotlin, в качестве зависимости нам здесь будет достаточно только Web. Скачаем заготовку проекта и откроем файл build.gradle.kts. Проверьте, что в секции dependencies присутствует org.springframework.boot:spring-boot-starter-web.  Также добавим туда библиотеку для работы с Telegram org.telegram:telegrambots-spring-boot-starter:5.4.0.1.  dependencies {  implementation("org.springframework.boot:spring-boot-starter-web")  implementation("org.telegram:telegrambots-spring-boot-starter:5.4.0.1")  // другие зависимости  }  Теперь создадим новый сервис. В telegram боты можно подключать двумя способами: long polling и web-hook. В случае с long polling наше приложение кидает запрос и ждёт ответа от сервера telegram. Сервер ответит не сразу, а только тогда, когда произойдёт какое-либо событие (например, сообщение от пользователя). А в случае с webhook сервер telegram сам будет дёргать заранее зарегистрированные эндпоинты нашего приложения. В подключенной нами библиотеке поддерживаются оба варианта, но webhook чуть сложнее в настройке. Поэтому рассмотрим long polling.  Унаследуем наш сервис от класса TelegramLongPollingBot. Этот абстрактный класс потребует от нас реализации методов getBotUsername(), getBotToken() и onUpdateReceived().  Первые два метода должны возвращать те самые данные, которые мы получили при регистрации. Однако нельзя их хардкодить в виде констант. Они должны подгружаться из параметров приложения.  @Service  class DevmarkBot : TelegramLongPollingBot() {  @Value("\${telegram.botName}")  private val botName: String = ""  @Value("\${telegram.token}")  private val token: String = ""  override fun getBotUsername(): String = botName  override fun getBotToken(): String = token  Благодаря аннотации @Value Spring сам подставит параметры из файла application.properties. Пропишем их в этом файле, а лучше сразу его переименуем в application.yml, чтобы писать в yaml-формате:  telegram:  botName: devmark\_ru\_bot  token: ${TOKEN}  Здесь мы указываем имя бота (telegram.botName) в явном виде, а вот токен (telegram.token) подгружаем из переменной окружения, т.к. этот токен должен сохраняться в секрете. Переменную окружения можно указывать при запуске приложения из командной строки через опцию -D или непосредственно в Idea. Обработка запросов от пользователя Теперь вернёмся к нашему сервису и реализуем метод onUpdateReceived().  override fun onUpdateReceived(update: Update) {  if (update.hasMessage()) {  val message = update.message  val chatId = message.chatId  val responseText = if (message.hasText()) {  val messageText = message.text  when {  messageText == "/start" -> "Добро пожаловать!"  else -> "Вы написали: \*$messageText\*"  }  } else {  "Я понимаю только текст"  }  sendNotification(chatId, responseText)  }  }  В начале мы проверяем объект типа Update на наличие сообщения с помощью метода hasMessage(). Далее, извлекаем chatId (уникальный идентификатор пользователя в telegram). Затем проверяем, что входящее сообщение содержит текст (а не стикер, к примеру). Если сообщение от пользователя равно строке «/start», то мы приветствуем пользователя. Дело в том, что именно такое сообщение отправляется, когда вы впервые подключаетесь к боту и жмёте кнопку «Start». Для любого другого текста мы просто дублируем его в ответе. Если же текста нет, бот ответит, что понимает только текст.  Отправка сообщения происходит во вспомогательном методе sendNotification().  private fun sendNotification(chatId: Long, responseText: String) {  val responseMessage = SendMessage(chatId.toString(), responseText)  responseMessage.enableMarkdown(true)  execute(responseMessage)  }  В нём мы создаём объект ответа, заполняя chatId и текст ответа. Затем с помощью метода enableMarkdown() включаем режим разметки Markdown. Этот режим позволяет делать простое форматирование текста. Например, текст, обрамлённый с двух сторон символами звёздочки будет отображаться жирным. Затем для отправки сообщения вызываем метод execute() из родительского класса.  Добавление кнопок  Указанный выше способ взаимодействия между пользователем и ботом довольно универсален. Однако чтобы пользователю не набирать каждый раз одни и те же команды, а также чтобы исключить вероятность опечатки, мы можем предоставить ему несколько заранее заготовленных команд в виде кнопок. При нажатии на кнопку бот получит ровно тот текст, который на ней написан. В этом смысле нет разницы между нажатием на кнопку и набором этого текста вручную.  Давайте сделаем так, чтобы пользователю отображалось 4 кнопки, по 2 в ряд. Для этого модифицируем наш метод отправки сообщения пользователю:  private fun sendNotification(chatId: Long, responseText: String) {  val responseMessage = SendMessage(chatId.toString(), responseText)  responseMessage.enableMarkdown(true)  // добавляем 4 кнопки  responseMessage.replyMarkup = getReplyMarkup(  listOf(  listOf("Кнопка 1", "Кнопка 2"),  listOf("Кнопка 3", "Кнопка 4")  )  )  execute(responseMessage)  }  Тут мы просто указываем свойство replyMarkup, для чего вызываем метод getReplyMarkup(), передавая ему на вход список списков строк. Сам метод выглядит примерно так:  private fun getReplyMarkup(allButtons: List<List<String>>): ReplyKeyboardMarkup {  val markup = ReplyKeyboardMarkup()  markup.keyboard = allButtons.map { rowButtons ->  val row = KeyboardRow()  rowButtons.forEach { rowButton -> row.add(rowButton) }  row  }  return markup  }  Создаём объект ReplyKeyboardMarkup. Как нетрудно догадаться по названию, он отвечает за разметку кнопок. Затем проходимся по каждой строке, создавая KeyboardRow и заполняем её кнопками. Для создания кнопки требуется указать только текст.  Наконец, немного модифицируем наш обработчик запросов:  when {  messageText == "/start" -> "Добро пожаловать!"  messageText.startsWith("Кнопка ") -> "Вы нажали кнопку" // обработка нажатия кнопки  else -> "Вы написали: \*$messageText\*"  }  Как только пользователь отправит сообщение, начинающееся со строки «Кнопка », мы определим, что пользователь нажал именно кнопку.  Практическое задание 4.  Подключение базы данных к Telegram-боту.  **Дедлайн - зачетная неделя**  Описание задания  В рамках практической работы требуется реализовать взаимодействие разработанного бота с таблицами в базе данных.  JDBC  В качестве примера реализации рассмотрим библиотеку JDBC.  JDBC расшифровывается как подключение к базе данных Java. Это низкоуровневая библиотека, которая предоставляет функциональные возможности для взаимодействия с базой данных. Чтобы использовать JDBC, вам не нужно зависеть от какой-либо сторонней библиотеки. Он поставляется в комплекте со всеми существующими JDK.  Как использовать JDBC с Kotlin  Шаг 1: Подготовьте таблицу в вашей базе данных  Шаг 2: Найдите драйвер JDBC для вашей базы данных и добавьте его в свой проект  Шаг 3: Установите соединение, используя URL-адрес JDBC  Шаг 4: Напишите и выполните SQL-запросы  Подготовка таблицы  Давайте создадим таблицу под названием users и наполним ее некоторыми данными. В примере используется PostgreSQL. Вы можете использовать базу данных по вашему выбору.  CREATE TABLE users (id SERIAL PRIMARY KEY, name VARCHAR(20));  INSERT INTO users (name) VALUES  ('Kohli'),  ('Rohit'),  ('Bumrah'),  ('Dhawan');  Результат выглядит так:  SELECT \* FROM users;  id | name  ----+--------  1 | Kohli  2 | Rohit  3 | Bumrah  4 | Dhawan  (4 rows)  Добавьте драйвер JDBC в свой проект  Чтобы получить доступ к этим данным из вашего кода Kotlin, вы должны включить драйвер JDBC вашей базы данных в качестве зависимости Gradle. Если вы не используете PostgreSQL, вам нужно найти драйвер JDBC используемой вами базы данных.  dependencies {  implementation("org.postgresql:postgresql:42.3.1")  }  Создайте класс модели  Сначала создайте класс модели для данных, которые мы собираемся получить из базы данных.  data class User(val id: Int, val name: String)  Установите соединение с базой данных через URL-адрес JDBC  Чтобы создать подключение к базе данных, вам понадобится URL-адрес JDBC. Вот как выглядит URL-адрес JDBC:  fun main(){  val jdbcUrl = "jdbc:postgresql://localhost:5432/example"  }  Как вы можете видеть, URL-адрес JDBC состоит из трех основных компонентов. Имя хоста , номер порта и имя базы данных.  Чтобы создать соединение, используйте метод DriverManager.getConnection().  import java.sql.DriverManager  data class User(val id: Int, val name: String)  fun main(){  val jdbcUrl = "jdbc:postgresql://localhost:5432/example"  // get the connection  val connection = DriverManager  .getConnection(jdbcUrl, "postgres", "postgres")  }  Первым аргументом метода DriverManager.getConnection является URL-адрес JDBC, вторым аргументом является имя пользователя базы данных, а последним аргументом является пароль для этого пользователя.  Вы можете проверить, является ли соединение действительным или нет, используя метод isValid() для объекта Connection.  import java.sql.DriverManager  data class User(val id: Int, val name: String)  fun main(){  val jdbcUrl = "jdbc:postgresql://localhost:5432/example"  val connection = DriverManager  .getConnection(jdbcUrl, "postgres", "postgres")  println(connection.isValid(0))  }  SQL-запросы  Теперь вы можете писать и выполнять SQL-запросы, используя соединение. Но поскольку JDBC - это низкоуровневая библиотека, все должно быть сделано вручную.  import java.sql.DriverManager  data class User(val id: Int, val name: String)  fun main(){  val jdbcUrl = "jdbc:postgresql://localhost:5432/example"  val connection = DriverManager  .getConnection(jdbcUrl, "postgres", "postgres")  println(connection.isValid(0))  // the query is only prepared not executed  val query = connection.prepareStatement("SELECT \* FROM users")  // the query is executed and results are fetched  val result = query.executeQuery()  // an empty list for holding the results  val users = mutableListOf<User>()  while(result.next()){  // getting the value of the id column  val id = result.getInt("id")  // getting the value of the name column  val name = result.getString("name")  /\*  constructing a User object and  putting data into the list  \*/  users.add(User(id, name))  }  /\*  [User(id=1, name=Kohli), User(id=2, name=Rohit),  User(id=3, name=Bumrah), User(id=4, name=Dhawan)]  \*/  println(users)  } |