GEEKBRAINS

Факультет Разработчик — Программист. Специализация

**ДИПЛОМНЫЙ ПРОЕКТ**

**«Программа по автоматическому внесению персональных данных получателей услуг в установленные формы документов формата docx»**

**Выполнил:**

Чиртик Александр

Александрович

г. Кемерово

2023 г.

Содержание

**Введение…………………………………………………………………………....3**

**Глава 1. Ввод необходимых данных …………………………………………...5**

* 1. . Создание интерфейса для ввода данных…………………………………….5

1.2. Переход между страницами приложения…………………………………..13

**Глава 2. Обработка полученных данных ……………………………….........17**

2.1. Получение данных из интерфейса ввода …………………………………..17

2.2. Библиотека Apache POI и создание форм шаблонов …..………………....19

2.3 Заполнение шаблонов необходимыми данными……………………………20

**Глава 3. Взаимодействие с базой данных……………………………………..27**

3.1. Подключение к базе данных…………………………………………………27

3.2 Внесение информации в базу данных………………………………………..30

3.3. Получение информации из базы данных……………………………………33

**Заключение………………………………………………………………………..38**

**Список используемой литературы и источников……………………………40**

**Приложения……………………………………………………………………….42**

**Введение**

**Актуальность выбранной темы**. В рамках предыдущей деятельности мне приходилось в большом количестве работать с личными документами граждан. Так как работа по моему направлению деятельности была во многом уникальна, не было возможности передать данные функции другому специалисту. Мне приходилось вручную переносить данные из личных документов граждан в шаблоны документов. А затем добавлять всё те же данные в базу получателей услуг. Такая работа занимала много времени. Да и в целом, выполнять достаточно монотонную работу в современном мире – это уже некая архаичность. Поэтому перед моей организацией встал вопрос об усовершенствовании данной деятельности и её перевода в цифровой формат.

Именно поэтому в качестве дипломного проекта я решил выбрать решение данной проблемы. На мой взгляд, создание отдельного десктопного приложения специально для указанной задачи является конструктивным решением. Моя идея заключается в том, чтобы создать программу для операционной системы Windows, с помощью которой информация о получателях услуг будет автоматически вноситься в установленные шаблоны документов. Так как в разные шаблоны чаще всего вносится одна и та же информация, в программе можно будет внести уникальные данные в единое окно, а затем они автоматически помещаются сразу в несколько подготовленных шаблонов. Одновременно с этим информация о получателях услуг будет помещена в базу данных.

**Целью** данной работы является создание рабочего прототипа десктопного приложения, которое сможет одновременно переносить информацию из интерфейса ввода данных в шаблоны документов формата docx и базу данных MySQL.

**Цель данной работы потребовала сформулировать следующие задачи:**

Во-первых, разработать интерфейс для ввода данных в рамках JavaFX проекта.

Во-вторых, организовать переключение между страницами приложения.

В-третьих, создать способ получения данных из интерфейса ввода и сохранения их в программе.

В-четвёртых, подготовить шаблоны документов формата docx и подключить сторонние библиотеки для работы с Word.

В-пятых, внести полученную информацию в необходимое количество шаблонов.

В-шестых, настроить подключение к базе данных.

В-седьмых, создать метод внесения персональных данных граждан в базу данных.

В-восьмых, создать метод получения информации о созданных ранее документах из базы данных.

**Структура работы.** Данная работа состоит из введения, трех глав, заключения, списка использованной литературы и источников и приложения.

**В проекте** участвовал в качестве Java-разработчика и продукт-менеджера.

**Инструменты**: IntelliJ IDEA, Scene Builder.

**Глава 1. Ввод необходимых данных**

* 1. Создание интерфейса для ввода данных

Для создания пользовательского интерфейса приложения я выбрал платформу JavaFX. Это библиотека для создания на языке Java приложений с графическим интерфейсом, анимацией, насыщенной графикой и интерактивными элементами. Благодаря такому выбору я могу написать управляемый и понятный код на языке программирования Java. Ведь именно на этот язык программирования был сделан упор в программе на курсе обучения. Удобство данной платформы заключается в том, что она уже содержит большинство необходимых мне библиотек. Нельзя не отметить тот факт, что данная платформа оказалась максимально удобной для меня при создании кнопок и таблиц и прочих графических элементов.

Для разработки приложения я выбрал интегрированную среду разработки программного обеспечения IntelliJ IDEA.

Само приложение будет строиться по принципу MVC. То есть вся логика приложения будет разделена на три основные части:

1. Model – классы, которые будут отвечать за основную логику программы (внесение данных в шаблоны документов, взаимодействие с базой данных).
2. View – это будут fxml файлы, отвечающие за внешний вид приложения и взаимодействие с пользователем.
3. Controller – классы, связывающие в себе логику поведения из Model классов и внешний интерфейс приложения.

Запускать приложение мы будем из класса Application, который наследует абстрактный класс Application из библиотеки javafx.application. В нашем классе находится лишь необходимый код для запуска приложения.

Создаем метод start, который принимает автоматически созданный объект класса Stage. Именно класс Stage представляет собой контейнер, в который складываются все элементы интерфейса. Благодаря ему мы видим окно приложения, появляющиеся при запуске. Не забываем и про исключения. Используем ключевое слово throws и checked исключение IOException. Данное исключение будет сигнализировать о возникновении какого-либо исключения ввода-вывода нашего графического интерфейса на экран.

public void start(Stage stage) throws IOException {}

Дальше мы используем экземпляр класса FXMLLoader для отображения файла View.fxml. Данный класс отвечает за обработку кода из файла формата fxml. Передаем в конструктор в качестве параметра путь к необходимому ресурсу с помощью метода getResource.

FXMLLoader fxmlLoader = new FXMLLoader(Application.class.getResource("View.fxml"));

Создаем экземпляр класса Scene, который будет являться контейнером для всех графических элементов внутри объекта Stage. В качестве параметра передаем экземпляр класса FXMLLoader, созданный ранее, с методом load, который возвращает объект типа Parent, так называемый корневой узел. Проще говоря, Parent – это тоже контейнер, но содержащий объекты класса Node (кнопки, текст, поля ввода и т.д.). Благодаря этому наше приложение получит интерфейс из файла View.fxml. Конструктор экземпляра класса Scene принимает ещё 2 параметра: ширину и высоту окна приложения.

Scene scene = new Scene(fxmlLoader.load(), 1200, 700);

Далее устанавливаем иконку приложения в левом верхнем углу. Для этого к контейнеру stage применяем метод getIcons(), а после этого метод добавления add(). Внутри метода add создаем объект класса Image с указанием пути к нужной картинке.

stage.getIcons().add(new Image(…);

Теперь в объект класса Stage добавляем созданную сцену с помощью метода setScene(). Это необходимо для выстраивания структуры интерфейса с учетом вложенности классов друг в друга.

stage.setScene(scene);

Метод stage.show() делает контейнер stage видимым при запуске. Следовательно, раз в него вложены все остальные классы (Scene, Parent), мы можем увидеть все добавленные элементы на экране.

stage.show();

В том же классе Application создаем стандартный метод main. И внутри него запускаем единственный статический метод launch(). Он запустит среду выполнения JavaFX и всё приложение в целом. Метод launch() автоматически понимает, из какого класса его вызывают. Именно поэтому он начинает реализовывать код из класса Application, а не ищет его где-то в другом месте.

Визуальную часть пользовательского интерфейса я буду создавать с помощью инструмента визуального макета Scene Builder. Он позволяет создать необходимый визуал без набора кода, а с помощью наглядного наложения элементов друг на друга.

В программе Scene Builder открываю файл – View.fxml, созданный до этого в проекте. Это первое окно, которое встречает пользователя при запуске приложения. Создаю фоновый слой AnchorPane. На нем будут располагаться все остальные элементы. Размер этого слоя должен совпадать с размерами окна, заданными в классе Application. Сверху на этом слоя я располагаю ещё один элемент AnchorPane, но с меньшей длиной. Этот слой будет отвечать за область окна приложения, на которой в дальнейшем расположиться текст.

Общий размер окна составляет 1200 в ширину и 500 в длину. Второй слой AnchorPane располагает той же шириной, но длина составляет всего лишь 200. Оставшуюся длину окна займет фоновое изображение. В программе Scene Builder за него отвечает элемент ImageView. Я скачал необходимую мне для фона картинку и расположил координаты её расположения в поле Specific раздела Properties в программе Scene Builder.

После этого я добавил надпись «Заполнение документов» в верхнем AnchorPane. Сделал я это с помощью элемента Label.

Последними по порядку, но не по важности элементами остались кнопки. Их на этой странице будет две. Одна - «Заполнить новые документы», будет отвечать за переход к странице для внесения данных в поля ввода. Вторая - «Список созданных», будет открывать таблицу с перечислением уже созданных пакетов документов, полученную из базы данных. Кнопки создаются с помощью элемента Button. И самое главное, кнопкам в разделе Code нужно присвоить fx:id – уникальное название элемента. С помощью этих уникальных id мы сможем прописывать логику поведения приложения при взаимодействии с тем или иным элементом. Кнопке «Заполнить новые документы» присваивается fx:id sendDocuments. А кнопке «Список созданных» - sentList (Приложение №1).

Всего в моем приложении будет четыре окна. Из первой страницы откроется либо окно с возможностью заполнения документов, либо с таблицей уже созданных. Четвертое окно будет открываться после внесения всех необходимых данных о получателе услуг. Это окно засвидетельствует, что всё прошло удачно и нужные документы создались.

Мы создали красивый макет. Но чтобы он работал, ему нужно добавить логику поведения. Если нажать на первую кнопку, то будет то… Если нажать на вторую, то будет это…. Весь код, который будет регулировать данное поведения, пишется в контроллерах. К каждому файлу fxml добавляется контроллер. К файлу View.fxml добавляется класс Controller (Приложение № 7). В программе Scene Builder очень легко получить весь код необходимый для класса Controller. После создания файла fxml в разделе Controller мы прописываем класс, находящийся в IntelliJ IDEA, который будет отвечать за указанный fxml файл. После этого в меню Scene Builder заходим во View, выбираем поле «Show Sample Controller Skeleton», и программа сама сгенерирует нужный код. Нам остается его только скопировать и вставить в класс Controller в IntelliJ IDEA. Мы получаем все элементы, имеющие id, и метод initialize(), отвечающий за поведение при взаимодействии с каким-либо элементом.

Следующий файл – ViewFilling.fxml, представляет собой набор полей для ввода информации. В него попадаем после нажатия кнопки «Заполнить новые документы». Пока она не работает, но вскоре её доработаем. Основу fxml файл берем из View.fxml. Все остальные страницы также будут иметь эту основу, чтобы фон на всех страницах приложения был одинаковый. Меняем текст в элементе Label и добавляем нужные поля ввода данных. Их делаем с помощью элемента TextField. Для моего проекта нужны девять элементов TextField. Внутри самих элементов можно прописать текст, обозначающий какие данные в него вводить. Делается это в разделе Properties в поле Prompt Text. При вводе данных заранее вписанных текст исчезнет. И того у меня девять элементов TextField: ФИО в именительном падеже, Дата рождения, Город (район), Улица, Данные паспорта, Дом, Квартира, Телефон, Дата обращения. К ним я добавляю элемент, который поможет пользователю с выбором пола простым нажатием на нужную ячейку. Таким элементом является CheckBox. Их нужно два. Первому даем обозначение «Мужской», второму «Женский». К ячейке «Мужской» сразу добавляем параметр Selected в поле Specific. Это необходимо для дальнейшей логики обработки данных из этой страницы. Последним полем для получения данных о получателе станет элемент ComboBox. Его особенность заключается в том, что в нем можно выбрать один из нескольких вариантов чего-либо. В моем случае это будут учреждения, в которые предположительно отправятся получатели за услугами. Сами варианты я пропишу чуть позже в классе ControllerFilling (Приложение № 9). Этому элементу в поле Prompt Text также задаем ожидаемые данные – Учреждения.

И последним элементом на этой странице станет кнопка «Сформировать». Она будет запускать весь процесс обработки данных, внесения их в шаблоны документов и базу данных.

По аналогии с предыдущей страницей для каждого элемента, участвующего в работе программы, задается fx:id. Для полей ввода данных: ФИО в именительном падеже - name, Дата рождения - birth, Город (район) - city, Улица - street, Данные паспорта - passport, Дом - house, Квартира - flat, Телефон - phone, Дата обращения – date. Для ячейки выбора пола Мужской – male. Здесь id нужен только для одной ячейки, в дальнейшем станет понятно почему. Для элемента ComboBox с вариантами учреждений – institution. И кнопка «Сформировать» получает id: dataSend.

После этого также создаем в среде разработки класс ControllerFilling. Привязываем данный класс к созданному fxml файлу. Затем копируем код из поля «Show Sample Controller Skeleton» в класс ControllerFilling. В данном случае стоит самостоятельно дописать нужный код внутрь класса. Это связано с элементом ComboBox. Чтобы поместить внутрь элемента все необходимые варианты до метода initialize() прописывается следующее:

private ObservableList<String> CBOptions = FXCollections.observableArrayList("Учреждение 1", "Учреждение 2", "Учреждение 3", "Учреждение 4");

private ComboBox<String> selectedItem = new ComboBox<String>(CBOptions);

Интерфейс ObservableList наследует как интерфейс List, так и интерфейс Observable для того чтобы можно было не только хранить коллекцию переменных, но ловить события. То есть при изменении элемента ComboBox под выбранное событие подстроиться нужный элемент коллекции. У меня таких элемента четыре. Дальше помещаем эту коллекцию в новый ComboBox. Нужно это для помещения списка вариантов в качестве item в изначальный ComboBox с id: institution. Благодаря этому не произойдет конфликта классов. Затем уже в методе initialize() устанавливается значение по умолчанию с помощью метода setValue() на случай, если при заполнении ни один из элементов не будет выбран. В заключении все варианты внутри ObservableList<String> CBOptions помещаются в первый ComboBox с помощью метода setItems(). Теперь при нажатии на всплывающий список можно увидеть все созданные варианты учреждений. Методы setValue() и setItems() используются внутри initialize() из-за того, что должны запускаться только при открытии данной страницы (Приложение № 2).

Третьим из четырех окон будет таблица с данными об уже созданных пакетах документов. О том, как данные попадают в эту таблицу, я расскажу чуть дальше. А сейчас рассмотрим то, как создать живую таблицу с данными в Scene Builder. Почему живую? Просто мы не можем заранее рассчитать, сколько получателей услуг к нам придет за услугами. Поэтому делать таблицу на определенное количество строк нецелесообразно. Background страницы взят, как и раньше, от файла View.fxml. Исключение в том, что картинку для нижней части страницы пришлось закрыть. Её место заняла сама таблица. Верхняя часть осталась прежней, но произошла замена надписи в элементе Label. За создание таблице в Scene Builder отвечает элемент TableView. Для начала задаются отступы таблицы от края окна и других элементов. В моём проекте отступ таблицы сверху равен 202, снизу, справа и слева 1. Первоначально в таблице находятся два столбца. Мне же нужно гораздо больше. Чтобы добавить столбцов используем элемент TableColomn. Они добавляются в верхний раздел таблицы. Моя таблица отображает то количество столбцов, которое находится в базе данных. А именно: id, Дата, Имя, Пол, Дата рождения, Город, Улица, Дом, Квартира, Телефон, Паспорт, Учреждение. После создания необходимого количества столбцов, их надо распределить по ширине. Здесь я отталкиваюсь от того, какую длину займет та или иная информация. Всё, все столбцы добавлены и распределены по ширине. Дальше каждому из них присваиваем fx:id. Как и с элементами ввода данных, выбора и кнопками мы будем взаимодействовать с каждым из столбцов таблицы. Нам нужно передавать информацию из базы данных в каждый из столбцов по отдельности. Поэтому каждому из них я добавил id, с которым дальше будем работать в коде: id – fieldID, Дата – fieldDate, Имя – fieldName, Пол – fieldGender, Дата рождения – fieldBirth, Город – fieldCity, Улица – fieldStreet, Дом – fieldHoume, Квартира – fieldFlat, Телефон – fieldPhone, Паспорт – fieldPassport, Учреждение – fieldInstitution. Почти все готово. Осталось только по аналогии с предыдущими окнами создать ответственный за страницу класс-контроллер. Таким будет класс ControllerList (Приложение № 10). Привязываем его к fxml файлу и копируем код для класса ControllerList из поля «Show Sample Controller Skeleton». Страница для отображения данных из MySQL создана! Но теперь нам надо доработать класс ControllerList. Дело в том, что в скопированном коде не указан тип данных, с которым будет работать таблица. Мы должны задать его самостоятельно. Для работы с базой данных я создам отдельный класс Client, который будет содержать поля соответствующие столбцам таблицы в базе (Приложение № 6). Именно этот класс я передам в качестве типа данных в TableView, так как поля будут совпадать. К столбцам таблицы помимо типа данных Client нужно добавить один из базовых типов, чтобы при получении данных не возникало ошибки. В столбец с индексом fieldID добавляю целочисленный тип данных Integer, поскольку он будет отвечать за порядковый номер строки. Сам номер будет присваиваться в таблице в MySQL автоматически. Все остальные столбцы в дополнение к типу Client получают тип данных String, поскольку эти поля в таблице будут содержать лишь текстовую информацию о получателе услуг (Приложение № 3).

Последним четвертым окном станет обычное информационное сообщения о том, что документы успешно сформированы. Для этого создаем отдельный файл – ViewAfterFilling.fxml. Копируем внутрь него всё содержимое файла View.fxml. После этого заходим в Scene Builder. Открываем созданный файл ViewAfterFilling и удаляем кнопки, которые на этой странице нам не понадобятся. Фон оставляем прежним. Изменения грозят только элементу Label. И это логично, поскольку пользователь должен получить информацию о готовности документов. Можно было оставить всё так, но я решил добавить картинку, чтобы окно смотрелось чуть более заполненным. Добавить картинку можно с помощью элемента ImageView, как делали это ранее. В поле Specific раздела Properties указываем путь к нужной картинке. Так как на этой странице нет элементов, влияющих на логику работы программы, мы не присваиваем fx:id (Приложение № 4).

Все четыре страницы (окна) готовы! Все они представляют собой fxml файлы и находятся в отдельной от основного проекта папке. Согласно структуре JavaFX проекта в IntelliJ IDEA все файлы fxml располагаются в папке resources. И благодаря настройке JavaFX проекта классы-контроллеры, класс Application и все остальные видят файлы fxml, лежащие в resources без указания абсолютного или относительного пути. Достаточно прописать в классе только название файла, и он этот файл найдет сам, поскольку знает, где нужно искать. Таким образом в проекте есть две основные папки: java с классами, интерфейсами, и шаблонами документов; а также resources с fxml файлами и картинками.

* 1. Переход между страницами приложения

У нас есть четыре созданных страницы, на каждую из которых нужно перейти определенным способом. Проект не предусматривает переключение «от каждой к каждой». Страницы будут менять по принципу дерева. В зависимости от того на какую кнопку нажали, выбирается подходящий вариант действия. И под это действие открывается подходящая страница.

Как открывается первая страница, я уже упоминал. В классе Application (Приложение № 5) благодаря методу getResource() выбирается файл из папки resources. Дальше в коде прописано, как этот файл запускается. Получается что файл View.fxml открывается при запуске приложения. На открывшейся странице есть две кнопки. Они ведут к двум другим страницам: ViewFilling.fxml и ViewList.fxml. Кнопка «Заполнить новые документы» откроет окно ViewFilling.fxml с полями для заполнения данных. А кнопка «Список созданных» откроет таблицу, в которой автоматически перейдет информация из базы данных.

Происходит это следующим образом. Ответственный за любые действия на странице – это метод initialize(), располагающийся в классах-контроллерах. Он создается в Scene Builder и копируется вместе со всем остальным кодом из поля «Show Sample Controller Skeleton». На первой странице никаких автоматических действия от открытия страницы не происходит. Два возможных события запустятся только после нажатия кнопок. Чтобы событие при нажатии на кнопку произошло, нужно прописать это в коде. Начнем с кнопки «Заполнить новые документы» с id – sendDocuments. В методе initialize() пишем:

sendDocuments.setOnAction(event -> {});

Выбираем id нужной кнопки. После пишем метод setOnAction. Он обозначает, что при взаимодействии с этим элементом должно произойти то, что прописано в скобках. Соответственно единственным взаимодействием с кнопкой может быть только её нажатие. В скобках по умолчанию пишется слово event (событие), а дальше в фигурных скобках будет код, отвечающий за логику поведения после нажатия. В фигурных скобках нужно прописать следующее:

sendDocuments.getScene().getWindow().hide();

Программа берет сцену и окно, связанные с элементов sendDocuments, и закрывает это окно. Таким образом, обеспечивается последовательный переход между страницами – одна закрывается, другая открывается. Далее, чтобы создать компоненты графического пользовательского интерфейса, объявляем экземпляр класса FXMLLoader. Он нужен для того, чтобы заложить путь и название того файла, который нужно будет открыть. После нажатия на кнопку нужно будет открыть ViewFilling.fxml. К созданному экземпляру класса loader применяем метод load(). Таким способом получим доступ к иерархии объектов из документа fxml. Так как при попытке получить доступ к иерархии мы можем получить исключение операции ввода. Вдруг какой-то из элементов не загрузится или в параметрах метода getResource() окажется название файла, которого не существует. Именно для отлова подобных исключений операция loader.load() помещается в блок try-catch. В блоке try находится код, который нужно выполнить. В блоке catch – исключение IOException (input – output) передаваемое в RuntimeException. После этого создаем экземпляр класса Parent. Он будет реализовывать функциональность для управления вложенными узлами. В объект данного класса передаем созданный ранее loader с методом getRoot(). Данный метод позволяет установить корневой узел. Финальным шагом будет создание объекта класса Stage. Как мы помним, он представляет собой основной контейнер, в который помещается все остальное. Простыми слова – это и есть окно приложения, которое открывается на экране. А уже в нем находятся все остальные элементы. По аналогии с классом Application к stage с помощью метода setScene() устанавливаем необходимую сцену. Для класса Controller такой сценой будет созданный ранее экземпляр класса Parent с именем root. Передав root в метод setScene() наше окно stage получает доступ ко всем элементам, созданным в макете Scene Builder. Теперь осталось только подсказать программе, что окно должно открыться и ожидать дальнейших действий. Сделать это возможно при помощи метода showAndWait(), примененному к stage. По большому счету все сделано. Теперь при нажатии на кнопку «Заполнить новые документы» с id – sendDocuments будет закрываться первая страница приложения. Вслед за этим откроется вторая страница с полями для внесения данных, ожидающая дальнейших действий.

Все переходы между страницами в моем проекте предусматривают один и тот же сценарий, описанный выше. Возвращаясь к первой странице приложения можно повторить тот же трюк с кнопкой «Список созданных». Все, что нужно поменять – это название некоторых файлов. В первой строке мы указываем, что действие будет происходить после нажатия на кнопку с id sentList.

sentList.setOnAction(event -> {});

Далее, как и в предыдущем случае, указываем, что страница, на которой находится кнопка с id sentList, должна закрыться. Создание экземпляров классов FXMLLoader, Parent, Stage не претерпевает изменений за исключением того, что в метод getResource() передается название другого fxml файла - ViewList.fxml. Той самой страницы с таблицей для получения информации из базы данных.

Последний переход между страницами будет совершаться после заполнения персональных данных в поля на странице ViewFilling.fxml. За это отвечает кнопка «Сформировать» с fx:id dataSend. Как и в предыдущих случаях именно схема перехода абсолютно стандартная. После нажатия на кнопку будут происходить и другие действия, связанные с логикой работы приложения. Но именно переключение между двумя страницами ничем не отличается от предыдущих. Мы также указываем название кнопки согласно её идентификатору и устанавливаем на ней событие благодаря методу setOnAction(). Прописываем в коде строчку кода для закрытия окна с id dataSend. Теперь окно закроется. Осталось прописать файл fxml, который должен отобразиться в приложении после нажатия кнопки. Это будет файл ViewAfterFilling.fxml. После закрытия окна для заполнения данных откроется окно с информацией о том, что все нужные документы успешно сформированы.

**Глава 2. Обработка полученных данных**

2.1. Получение данных из интерфейса ввода

Весь процесс получения и обработки данных из полей ввода происходит в классе ControllerFilling. Как было описано ранее, при попадании на страницу с вводом данных пользователь видит перед собой девять текстовых полей для ввода текста, две ячейки для выбора пола, одну из которых нужно выбрать, и развертывающиеся меню с перечнем учреждений. Пользователь вводит и выбирает все нужные данные и нажимает на кнопку «Сформировать». Именно после нажатия на эту кнопку запускается процесс обработки введенных данных. Ранее я уже упоминал, что все процессы происходят в методе initialize(). Здесь вся логика обработки данных также связана с данным методом. Само нажатие запускает и процессы обработки, и процесс переключения на другую страницу. С переключением мы разобрались в предыдущей главе. Теперь рассмотрим обработку информации.

При нажатии на кнопку внутри метода initialize() создаются 11 переменных с типом данных String. В них помещаются данные из полей на странице. Поместить данные в переменную позволяет метод getText(). Также стоит упомянуть о том, что элементы из Scene Builder при переводе их в код имеют параметр доступа private. Но так как метод initialize() находится с ними в одном классе, мы можем использовать элементы интерфейса без проблем с правами доступа. Итак, мы создали 11 переменных String, но данные сразу получат только 9 из них. Всё потому что для элементов CheckBox и ComboBox мы не можем воспользоваться методом getText(). В них пользователь не пишет текст, а выбирает из предложенных вариантов. Поэтому элементы с id gender и institution будут обрабатываться в коде отдельно от остальных. В созданных String переменных я использую названия, совпадающие с id элементов. Это сделано для того, чтобы не запутаться во множестве названий и сохранить стиль написания кода. При получении данных в созданные String переменные используется ключевое слово this. Оно говорит о том, что мы ссылаемся на текущий экземпляр класса. То есть обращаемся к элементам, находящимся в классе, а не в методе внутри класса.

На странице файла ViewFilling.fxml мы создавали две ячейки выбора пола. Обеим были присвоены fx:id. Но в коде нам понадобится только один из этих идентификационных номеров. Я выбрал id male. Хотя можно было выбрать и female. Никакой разницы от этого не будет. Теперь мы сможем выяснить, какой же пол был выбран. В методе initialize() после создания 11 переменных пропишем следующее:

if (male.isSelected()) gender = "Мужской";

else gender = "Женский";

Означает это следующее: если выбран элемент с id male, но в созданную ранее переменную gender помещается строка "Мужской". А если элемент в этим id не выбран - туда помещается строка "Женский". Именно поэтому для решения этого вопроса нужен лишь один элемент из двух, отвечающих за выбор пола. На месте male можно было бы прописать и female, большого различия в логике выбора не будет. Осуществить данный код помогает метод isSelected(), принадлежащий к элементу CheckBox. Он показывает, отмечена ли галочкой нужная ячейка или нет.

Последним элементом, для которого нужно описать логику получения данных, является ComboBox с id institution. Ранее в коде было прописано несколько строчек кода, отвечающих за пополнение элемента множеством вариантов выбора и установку варианта по умолчанию. Теперь осталось только понять, что же выбрал пользователь. В созданную String переменную institution передаем элемент ComboBox с ключевым словом this и применяем к нему 2 метода. Первый метод - getSelectionModel(), обрабатывает выбор элемента, понимая какую часть созданной модели выбрали. Второй метод - getSelectedItem(), возвращает в настоящий момент выбранный объект.И так как мы в private ObservableList<String> CBOptions указали тип данных String, не произойдет конфликта между типами данных. Мы заложили тип String в коллекцию переменных, получили вариант из коллекции с типом String и поместили полученный вариант в переменную с типом String.

Теперь у нас есть 11 String переменных, в которых находятся данные, введенные пользователем в графическом интерфейсе. Давайте же рассмотрим, как мы можем распорядиться полученными данными.

2.2. Библиотека Apache POI и создание форм шаблонов

Следующий блок моего проекта будет связан с работой по внесению данных в нужные шаблоны. Как уже описывалось ранее, основной упор в данном проекте сделан именно на данную задачу. Для того чтобы можно было работать с нужными документами нам потребуется дополнительная библиотека. Я выбрал формат документов docx. Библиотека для обработки документов указанного формата называется Apache POI. Эта библиотека позволяет использовать в Java приложениях файлы Microsoft Office. Чтобы встроить данную библиотеку в свой проект я пошел на Maven Repository. Там нашел нужную библиотеку и скопировал код для добавления в проект. Этот код вставляется в файл pom.xml, находящийся в проекте. Он создается автоматически, если указать при запуске проекта, что данный файл нам необходим. Затем нужная библиотека подгружается в наш проект. Теперь можно работать с документами.

В первую очередь нужно подготовить те шаблоны документов, с которыми дальше и будем работать. Во введение я объяснял значимость данного проекта тем, что на работе мне приходилось вносить во множество разных шаблонов одну и ту же информацию. Для наглядности в проекте я создам только два шаблона. В теории их может быть большое количество, но для понимания сути проекта хватит и двух. Это два docx файла. Один из них содержит только строки с каким-то набором символов. Во втором есть таблица, внутри которой также необходимо разместить данные. Именно наличие таблицы является существенным фактом, который влияет на логику работы программы. Дело в том, что построчный поиск элемента в документе, где нет таблицы, провести невозможно. Когда программа «упирается» в таблицу, она не понимает, что это за элемент, и просто перестает работать. Весь смысл поиска элемента внутри документа будет заключаться в том, что при его нахождении, данный элемент заменится на то слово, которое нам нужно. В первом шаблоне я оставил 10 слов-якорей: date – дата заполнения документа, name – фамилия, имя и отчество, male – пол, birth – дата рождения, city – город проживания, street – улица проживания, house – дом, flat – квартира, phone – номер телефона, passport – данные документа подтверждающего личность. Все перечисленные слова совпадают с полями для ввода данных, расположенных на странице ViewFilling.fxml. Суть заключается в том, что мы получаем данные из полей ввода и меняем на них слова-якоря, располагающиеся в шаблонах. Точно также поступаем со второй формой, обладающей таблицей. Внутри указанной таблице располагаются следующие слова-якоря: date - дата заполнения документа, name - фамилия, имя и отчество, institution – учреждение, предоставляющее услуги. Можно заметить, что 2 якоря совпадают с предыдущей формой. Таким образом, реализуется главный принцип данного проекта – из одного поля ввода данных информация расходится сразу в несколько документов.

2.3. Заполнение шаблонов необходимыми данными

За внесение данных в шаблоны документов будет отвечать класс WordDocuments (Приложение № 11). В нем есть 2 методы. Один отвечает за внесение данных в документы, где все содержимое расположено только в строках. Второй будет добавлять информацию в таблицы. Так как у меня будет 2 шаблона: с таблицей и без неё, оба метода окажутся полезными. Также для класса WordDocuments нужно объявить поведение, которое тот должен реализовывать. Речь идет именно о том, что указанный класс должен содержать 2 метода с заданными параметрами. Для этих целей создан интерфейс InsertToWord (Приложение № 17). В нем собственно и заявлено наличие двух методов с параметрами Client client. О типе данных Client я писал чуть ранее. До этого он был нужен для работы с таблицей в окне файла ViewList.fxml. Сейчас же он понадобится для работы с WordDocuments. Добавляем к названию класса строчку implements InsertToWord. Теперь класс WordDocuments реализует интерфейс InsertToWord. Следовательно, класс в обязательном порядке должен реализовать в себе методы, описанные в интерфейсе.

Первый метод называется insertToFormString. Он ничего не возвращает, поэтому указан void. В качестве параметра принимает переменную типа Client. Как понятно из названия, именно данный метод отвечает за работу с документами без таблиц. Он будет пробегаться по строкам и искать необходимое слово-приманку. В самом начале к методу с помощью throws прибавляется обработка исключений. Здесь могут быть input – output Exception, описанное ранее, и org.apache.poi.openxml4j.exceptions.InvalidFormatException. Последнее исключение проверяет формат открываемого файла, чтобы ему не предоставили формат, который метод не сможет обработать. Все действия будут происходить в блоке try для отлова исключений. После него в обязательном порядке должен находиться блок finally, чтобы не создавать лишний блок catch. Работа с документом начинается с открытия шаблона (Приложение № 19), созданного заранее. Для этого из библиотеки Apache POI выбираем класс XWPFDocument. Он используется специально для создания файла с форматом docx. Мы создаем экземпляр данного класса, чтобы положить в него абстрактный класс OPCPackage из той же сторонней библиотеки. Класс OPCPackage предоставляет контейнер, в котором может храниться несколько объектов данных. В нашем случае там будет храниться документ формата docx. А поместить его туда можно с помощью метода open(). Как понятно из названия, данный метод открывает то, что ему укажут. В нашем случае это будет шаблон документа. В качестве параметра метод open() принимает путь к файлу. Здесь я прописал путь к нужной форме шаблона. Весь остальной процесс в этом метода можно описать 3 словами – «Идем и ищем». То есть нужно просмотреть весь документ в поисках нужного слова. Идти нужно последовательно – от параграфа к строке, от строки к слову. Для определения параграфа внутри текста используется класс XWPFParagraph. Применяем класс XWPFParagraph в цикле for для предотвращения конфликта между типами данных, так как к экземпляру класса XWPFDocument под именем doc будет применен метод getParagraphs(). Метод находит параграфы по таким признакам как отступы слева и справа, сверху и снизу, в том числе и между строками, добавление гиперссылок и границ для параграфа.Выглядит условие цикла следующим образом:

for (XWPFParagraph p : doc.getParagraphs())

В теле цикла для начала создаем коллекцию List с таким типом данных как XWPFRun. Он также взят из библиотеки Apache POI. XWPFRun – это класс с набором данных о выводе текста внутри параграфа. Именно в коллекцию с типом XWPFRun можно поместить все элементы внутри отдельного параграфа. Чтобы получить все элементы, нужно применить метод getRuns() к параграфу. Как видно из предыдущей строчки кода, каждый параграф в цикле будет получать псевдоним «p». К нему нужно применить метод getRuns() и положить все это внутрь коллекции runs. Таким образом, мы получаем весь перечень элементов, находящихся внутри отдельно взятого параграфа.

Следующим логичным этапом будет проходка уже не по тексту для выделения параграфов, а по набору символов, полученных ранее. В первую очередь стоит проверить, не получилась ли коллекция пустой. Применяем проверку на null. Если все в порядке, создаем цикл for по аналогии с предыдущим. Только сейчас вместо класса XWPFParagraph будет использоваться XWPFRun, так как коллекция содержит элементы именно этого типа данных.

for (XWPFRun r : runs)

В итоге получается цикл внутри цикла. Во втором цикле все элементы коллекции будут иметь псевдоним «r». Чтобы рассмотреть каждый отдельный элемент коллекции runs, их каждый цикл нужно будет куда-то помещать. Для этого создана переменная String text. В связи с тем, что мы работаем с символами текста, тип данных String нам подходит. Каждую новую итерацию в переменную будет помещаться новый элемент из коллекции. Поместить его туда можно благодаря методу getText(0).

Имея на руках текст, можно приступать к основной задаче по замене одних слов на другие. Проверяем полученный текст на условие того, что он не является пустым. А также на то, что он содержит нужное нам слово-якорь. Если оба условия соблюдены, то в переменную text помещается та же переменная, но с замененными с помощью метода replace() слова. Программа меняет слово-якорь на переменную client с подходящим методом get. В классе Client к каждому полю добавлены геттеры и сеттеры. Так как в метод insertToFormString() в качестве параметра передается объект класса Client, мы можем воспользоваться любым из геттеров. Тем самым обеспечивается инкапсуляция кода. Получившуюся переменную text с уже измененными данными, устанавливаем в качестве единственно верного текста в тот элемент коллекции runs, в котором сейчас находимся. Установить текст можно с помощью метода setText(). Таким образом, мы взяли элемент коллекции, нашли в нем подходящее слова, произвели замену, установили элементу новый текст в качестве основного и вернули элемент обратно. Подобным образом в методе insertToFormString() поступаем ещё 9 раз, поскольку он отвечает за форму № 1, содержащую 10 слов-якорей. В итоге получится 10 условий if, в каждом из которых будут менять только слово-якорь и метод get для объекта класса Client.

После всех замен мы получили практически новый документ. Но так как потребуется вводить данные много раз для большого числа получателей услуг, шаблоны должны оставаться неизменными. Следовательно, после каждого применения метода из класса WordDocuments необходимо создавать новый файл docx. Рассмотрим этот процесс на примере метода insertToFormString(). Внизу блока try создается новый документ. Делается это с использованием метода write(), примененного к объекту doc. Метод write() в качестве параметра принимает объект класса FileOutputStream. Данный класс, в свою очередь, принимает путь к целевому файлу. Сам класс FileOutputStream будет каждый раз создавать новый объект, если ему указывать каждый раз новые пути. Или переписывать один и тот же файл, если не менять путь. Таким образом, получается, что документ doc будет создан в новом месте, а форма шаблона останется нетронутой.

doc.write(new FileOutputStream("…”));

Второй метод в классе WordDocuments принципиально ничем не отличается от insertToFormString(). Метод insertToFormTable() использует те же классы и методы для своей работы. Отличает его только то, что ко всем остальным добавляются циклы с несколькими новыми типами данных из библиотеки Apache POI. Рассмотрим это подробнее. У нас точно также используются 1 блока: try и finally. Также указывается путь к нужному шаблону (Приложение № 20) в переменной класса XWPFDocument. Различия начинаются с момента объявления циклов. В предыдущем методе было 2 цикла, вложенных друг в друга. Сейчас же потребуется 5 циклов. Более того, циклы с типами XWPFParagraph и XWPFRun будут в этом списке самыми последними.

А первым будет цикл, использующий публичный класс XWPFTable. Как понятно из названия, он отвечает за работу с таблицами внутри документа. С помощью метода getTables() мы получаем все таблицы, находящиеся в документе doc. Следовательно, сколько будет таблиц в документе, столько и будет итераций цикла. В моей форме шаблона находится только одна таблица с несколькими ячейками, поэтому цикл сработает лишь один раз.

for (XWPFTable tbl : doc.getTables())

Следующий цикл ищет строки взятой таблицы. Для этого применяется класс XWPFTableRow.К таблице, взятой из предыдущего цикла применяется метод getRows(). Тем самым мы получаем по отдельности каждую строку, содержащуюся в таблице.

for (XWPFTableRow row : tbl.getRows())

Третьим будет цикл, работающий с каждой из строк, полученной на цикл ранее. В этих строках с помощью класс XWPFTableCell и метода getTableCell будут выделены все ячейки, находящиеся в строке.Можно заметить, что все три новых цикла представляют собой стандартный поиск в трехмерном массиве.

Когда уже дошли до определенной ячейки, можно начинать обрабатывать её содержимое. С этим отлично справятся два цикла из предыдущего метода.

for (XWPFParagraph p : cell.getParagraphs())

for (XWPFRun r : p.getRuns())

Метод insertToFormTable() отвечает за вторую форму шаблона. В ней находится только 3 слова-якоря. Как было описано ранее, два из них совпадают с первой формой. В связи с этим, условий if в методе insertToFormTable() также будет 3. В конце по аналогии с первым методом создается новый docx файл, с замененными словами. Путь к файлу я указал тот же самый, что и в предыдущий раз, чтобы документы хранились в одном месте. Единственное отличие в записи данной строчки кода заключается в названии файла.

Всё что осталось, так это применить созданные методы в коде. Для этого переходим в класс ControllerFilling в метод initialize(). После нажатия на кнопку «Сформировать» создаются переменные String, в которые передается информация из полей ввода. Сразу за этим создается объект класса Client с конструктором специально для методов insertToFormString() и insertToFormTable(). В этот конструктор и передаются все String переменные из начала тела метода initialize(). Тем самым мы подготовили некого абстрактного клиента, данные которого будут обрабатывать методы из класса WordDocuments. Соответственно следующий шаг – это создание объекта класса WordDocuments с именем wd. Теперь применяем к wd оба метода для добавления данных в шаблоны. В качестве аргумента в оба метода передаем созданного абстрактного клиента.

wd.insertToFormString(client);

wd.insertToFormTable(client);

Чтобы вся программа заработала, остается обернуть примененные методы в блок try-catch. С помощью catch отлавливаем исключения IOException и InvalidFormatException. Именно эти исключения ранее прописывали у самих методов в интерфейсе InsertToWord.

**Глава 3. Взаимодействие с базой данных**

3.1. Подключение к базе данных

Работа программа уже налажена таким образом, что мы получаем необходимый результат в виде готовых документов. Осталось усовершенствовать приложение, сделать его более многофункциональным. Я говорю про возможность сохранять сведения в базу данных. Чтобы в любой момент можно было получить список тех, кто прошел через нашу программу. Этому и будет посвящена вся третья глава моей работы.

Осталось выбрать систему управления базой данных, с которой будем работать. Я остановил свой выбор на таком инструменте для визуального проектирования баз данных, как MySQL Workbench, поскольку уже работал с ней в рамках учебного курса. Я установил MySQL Workbench и сам сервер, а также вручную создал новый проект для работы с базами данных - «Project». Оставил по умолчанию hostname – «127.0.0.1», port – «3306», username – «root». Затем установил пароль – «12345». Внутри заранее подготовил ту БД, с которой и будет работать программа – «diploma». Внутри созданной БД формирую таблицу под названием «recipients». Данная таблица будет включать в себя 12 столбцов. Первым будет стандартное поле «id», отвечающее за нумерацию записей. Происходить это будет автоматически. При создании таблицы полю «id» был присвоен тип данных INT, так как единственная его функция – это увеличение порядка целочисленных значений. Также полю «id» добавлено свойство AUTO\_INCREMENT как раз для того, чтобы не увеличивать значение числа вручную, а отдать эту задачу самой СУБД. Оставшиеся 11 полей предназначены для хранения информации, полученной из полей для ввода информации на странице файла ViewFilling.fxml: «client\_date», «client\_name», «client\_gender», «client\_birth», «client\_city», «client\_street», «client\_house», «client\_flat», «client\_phone», «client\_passport», «institution». Все эти поля имеют тип данных VARCHAR(). Для каждого из них указано разное количество символов, но с расчетом на максимальную длину строки с данными на каждый из столбцов в отдельности. В указанных полях специально не добавлялось свойство NOT NULL, поскольку какие-то данные получатель услуг может не предоставить. В связи с этим, возможность хранить пустые записи заранее предусмотрена в таблице.

Теперь, когда таблица в базе данных создана, необходимо подключить её к проекту. Для этого будет создано 3 Java-класса. Первым делом создаем класс DatabaseConfigs (Приложение № 12). В нем будут находиться 5 полей с модификатором доступа protected и типом данных String. Модификатор protected подразумевает, что данные переменные могут быть видны только в самом классе DatabaseConfigs и в классах наследниках. Соответственно все 5 переменных будут использованы в классе DatabaseHandler, который создадим позже. Первая переменная, именуемая «dbHost», хранит в себе значение «localhost». Это IP-адрес хоста, по-другому обозначаемый как «127.0.0.1». Данный адрес мы и прописывали, когда создавали проект в MySQL Workbench. Остальные поля также совпадают с тем, что указано при создании БД. Переменная «dbPort» указывает порт подключения «3306», «dbUser» название юзера «root», «dbPass» - это пароль, а «dbName» обозначает имя базы данных «diploma», внутри проекта.

Вторым классом будет класс констант. Соответственно он будет иметь название DatabaseConst (Приложение № 13). Здесь будут храниться переменные с названиями таблицы и её полей, чтобы в дальнейшем не приходилось каждый раз создавать их в нужном фрагменте кода, а можно было брать уже готовые переменные. У них будет модификатор доступа public, чтобы мы могли достучаться до них из любых классов. Кроме того, они также будут static. Это необходимо для того, чтобы мы могли обращаться к ним без создания объекта на основе класса, а всего лишь через название класса. Также они ещё будут final. Если переменные имеют модификатор final, они становятся константой, их нельзя в дальнейшем изменить. Тип данных у них будет String. После прописываем название переменных. У константных полей принято писать названия в верхнем регистре. Первая переменная будет отвечать за название таблицы в БД. У меня таблица носит название «recipients».

public static final String TABLE\_NAME = "recipients";

Подобных переменных в данном классе будет ещё 11, которые понадобятся в дальнейшем для работы с полями таблицы.

Создадим ещё один класс, упомянутый ранее. Это будет класс DatabaseHandler (Приложение № 14). В нем будут лежать методы для работы с БД. В нем необходимо импортировать несколько библиотек, которые помогут связаться с SQL БД:

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

Класс DatabaseHandler должен наследовать класс DatabaseConfigs, чтобы использовать его переменные. Именно для этих целей в классе DatabaseConfigs переменные имеют модификатор доступа protected. Также DatabaseHandler наследует 3 интерфейса. Каждый из них отвечает за 1 из 3 действий, применимых к базе данных: подключение к БД, внесение в неё данных, получение из неё данных. В каждом из трех интерфейсов (ConnectionDB, InsertDB, SelectDB) прописана структура метода, отвечающего за одно из действий (Приложения № 15, 16, 18). Тем самым, объявляется поведение, которое должен реализовывать класс. Сразу несколько специализированных интерфейсов, объединенных общей функциональностью, нужны, чтобы не образовывать один большой и «неповоротливый» интерфейс.

Дальше в классе DatabaseHandler создаем публичный метод, который возвращает тип данных Connection, импортируемый чуть ранее. Называться метод будет getDbConnection(). Кроме того, данный метод будет отдавать исключения, если в нем что-то пойдет не так. Исключения будут следующие: ClassNotFoundException и SQLException, также добавленный ранее. Внутри метода создаем переменную класса Connection, которую назовем dbConnection. Далее создаем строку. В этой строке будут находиться все те данные, которые позволят подключиться к БД. Во-первых, в стоке прописываем такую структуру как «jdbc:mysql://». Это Java Database Connectivity - стандарт Java, который предоставляет интерфейс для подключения Java к реляционным базам данных. В добавление к нему прописываем данные из класса DatabaseConfigs, а именно: «dbHost», «dbPort», «dbUser». В итоге получилась строка для подключения к нашей базе данных с помощью плагина JDBC. Данный плагин подключаем к проекту отдельно в IntelliJ IDEA через Libraries.

String connectionString = "jdbc:mysql://" + dbHost + ":" + dbPort + "/" + dbName;

Далее прописываем такой метод для загрузки классов как Class.forName(). В нем устанавливаем драйвер, который будем использовать. В нашем случае это «com.mysql.jdbc.Driver». После этого берем переменную dbConnection и передаем в неё прописанное соединение: 1) применяем импортируемый класс DriverManager, который отвечает за работу с различными драйверами; 2) используем метод getConnection() для установки соединения с БД; 3) в метод передаем строку connectionString, а также пользователя и пароль из класса DatabaseConfigs («dbUser», «dbPass»).

dbConnection = DriverManager.getConnection(connectionString, dbUser, dbPass);

После всего возвращаем переменную dbConnection, содержащую правильный метод подключения к БД. В момент применения метода getDbConnection() наша программа сразу будет подключаться к базе данный в MySQL Workbench.

3.2 Внесение информации в базу данных

Для работы программы с реляционной базой данных нужно создать метод, который будет добавлять строки с информацией в созданную таблицу «recipients». С этой целью в классе DatabaseHandler создается метод logging(). В качестве параметров он принимает 11 значений с типом данных String. Количество параметров в методе совпадает с количеством полей записи таблицы в MySQL Workbench и учитывает только поля, в которые будет внесена информация от самого пользователя. Ещё одно поле таблицы «id» не учитывается в методе для передачи информации из программы в таблицу, потому что оно будет проставляться автоматически благодаря свойству AUTO\_INCREMENT. Было неправильно передавать номер id самостоятельно, в связи с этим мы будем работать только с полями String.

После мы создаем строку с SQL запросом к базе данных. Прописываем команду к СУБД так же, как если бы мы это делали внутри MySQL Workbench. Единственное отличие в том, что сейчас не удастся написать строку запроса просто текстом. Для составления полноценной строки необходимо подставить в неё значения из класса DatabaseConst и параметры, полученные в метод чуть ранее. Обычная строка запроса выглядит примерно так - «INSERT INTO table1 (a, b, c) VALUES (1, 222, 333), (2, 555, 666), (3, 555, 333)». Слово INSERT – это основная команда, говорящая о том, что данные будут добавлены в записи таблицы. Далее указывается название таблицы, с которой предстоит работать. Рядом с названием таблицы в скобках прописываются те поля записи, в которые добавиться информация. И уже после этого вместе с ключевым словом VALUES перечисляются вносимые данные. За название таблицы и перечисление полей записи внутри БД отвечает класс DatabaseConst. Как мы помним, внутри указанного класса находятся константные переменные, отвечающие за названия таблицы и полей. Становится понятным также смысл ключевого слова static перед ними. Нам не нужно создавать объект класса DatabaseConst. Мы можем обращаться к его полям напрямую через название класса. Выглядит в коде это вот так:

"INSERT INTO " + DatabaseConst.TABLE\_NAME + "(" + DatabaseConst.CLIENT\_DATE + "," + DatabaseConst.CLIENT\_NAME + "," + DatabaseConst.CLIENT\_GENDER + "," + DatabaseConst.CLIENT\_BIRTH + "," + DatabaseConst.CLIENT\_CITY + "," + DatabaseConst.CLIENT\_STREET + "," + DatabaseConst.CLIENT\_HOUSE + "," + DatabaseConst.CLIENT\_FLAT + "," + DatabaseConst.CLIENT\_PHONE + "," + DatabaseConst.CLIENT\_PASSPORT + "," + DatabaseConst.CLIENT\_INSTITUTION + ")"

Таким образом, все переменные из класса DatabaseConst задействованы разом в одном и том же месте кода. Теперь необходимо указать те значения, которые будут вставлены в указанные поля. После слова VALUES в скобках прописываем 11 знаков вопроса. В дальнейшем в эти знаки вопроса будут автоматически подставляться данные. Делается при помощи специального класса PreparedStatement. Это отдельный класс из библиотеки java.sql. Он позволяет подготовить запрос, отформатировать его. Также его удобно использовать для вставки параметров в SQL. Везде, где нужно это сделать, пишется знак вопроса. В тот момент, когда PreparedStatement создан для определенного оператора SQL (в нашем случае INSERT), можно вставить параметры в расположение знака вопроса. Реализовать это можно с помощью метода set с добавлением нужного типа данных. В моем случае для всех 11 знаков вопроса применяется метод setString(). В метод передается порядковый номер знака вопроса и то, что должно стоять на его месте. Естественно, передаваемая переменная должна совпадать с типом данных метода, а именно String. Создаем объект ps. После знака «=» используем метод getDbConnection() для подключения к необходимой базе данных. В дополнение к нему используем метод prepareStatement(), передавая в качестве параметра созданный ранее запрос insert. Метод prepareStatement() осуществляет обработку запроса к БД.

Последнее, что нужно сделать – это выполнить команду-запрос. Для этого к объекту ps необходимо применить метод executeUpdate(). Данный метод используется при обновлении базы данных. Он возвращает тип данных int с указанием того, сколько записей в БД было подвержено обновлению.

Все вышеописанное нужно обернуть в try/catch для отлова исключений. Исключениями могут быть SQLException при работе с БД и ClassNotFoundException при попытке загрузить класс, которого нет в classpath, во время выполнения метода Class.forName() (есть в методе подключения к БД getDbConnection()).

Осталось только вызвать метод logging() внутри класса ControllerFilling. После того как в классе проведены все процедуры, связанные с заполнением документов docx, создаем объект класса DatabaseHandler. У него вызываем метод logging(), в который передаем все созданные ранее в методе initialize() String переменные. Это ещё раз подтверждает удобство использования отдельных переменных, внутри которых находятся данные из полей ввода информации. Ведь в таком виде их можно использовать сразу в нескольких методах. Вызов метода logging() идет сразу после методов для работы с документами и перед конструкцией смены окон в приложении. Поэтому данный метод будет срабатывать автоматически сразу после нажатия кнопки «Сформировать». Так как метод срабатывает автоматически, он будет отправлять в таблицу данные обо всех получателях услуг без дополнительного подтверждения.

3.3. Получение информации из базы данных

Следующим шагом будет создание метода в классе DatabaseHandler для получения данных непосредственно из базы данных. Изначально для этой цели импортируем ещё одну библиотеку из java.sql под названием ResultSet (import java.sql.ResultSet;). При создании метода получения данных, который будет называться getClient(), в качестве возвращаемого значения используем коллекцию с массивами строк. Данное решение будет объяснено чуть позже. А сейчас вернемся к классу ResultSet. Данный класс представляет собой результирующий набор данных. Также он обеспечивает приложению построчный доступ к результатам запроса. Обеспечивается подобная функциональность с помощью набора get-методов, которые организуют доступ к полям записи в таблице.

Первым делом создаем объект класса ResultSet, в который будет передаваться результат запроса к базе данных, а именно запись о конкретном получателе услуг. Даем объекту имя resSet и оставляем его поначалу пустым.

Дальше создаем коллекцию allData. В неё будут переданы все записи из таблицы «recipients». Каждая отдельная запись представляет собой массив String значений, поэтому коллекция работает с типом данных String[]. Выбор коллекции объясняется тем, что записей может сильно отличаться в процессе. В начале работы их может быть 10, а через полгода - 500.

По аналогии с методом logging() необходимо создать SQL запрос к базе данных. Он будет отличаться от предыдущего, так как теперь мы не передаем данные в таблицу, а наоборот копируем их оттуда. Создаем String переменную select и передаем в неё стандартную запись для получения всех данных из таблицы «SELECT \* FROM ». В конце записи указываем лишь название той таблицы, из которой хотим получить данные. Здесь нам поможет класс DatabaseConst, в котором название таблицы было прописано заранее. К этому названию, как мы помним, можно обратиться через название класс. Поэтому в конце строки указываем DatabaseConst.TABLE\_NAME.

После этого копируем конструкцию try/catch из метода logging(). Удаляем лишние для текущего метода поля с установкой значений вместо знаков вопроса. Оставляем строку с установкой связи с БД. Только теперь передаем в метод prepareStatement переменную select.

PreparedStatement ps = getDbConnection().prepareStatement(select);

В методе logging() мы использовали executeUpdate() для обновления базы данных. Сейчас вместо этой команды используем метод executeQuery(). Данный метод сделан специально для команды SELECT, которая возвращает данные в виде ResultSet. Чуть ранее переменная ResultSet уже была создана. В неё и передаем данные, полученные с помощью executeQuery().

resSet = ps.executeQuery();

Теперь мы получаем всю информацию из таблицы в resSet. Но данные хранятся там как единое целое. А нам, чтобы с ним работать, необходима каждая строка в отдельности. Для этого создаем цикл while. И в цикле while будем проходить по отдельности всех получателей, которые были возвращены по результатам запроса. В этом поможет метод next(). Он используется для перемещения к следующей строке ResultSet, делая её текущей. Возвращает true, если есть следующая строка, в противном случае возвращает false.Таким образом, нам удается пройти по всем записям таблицы. Осталось понять, что делать с каждой отдельной записью? Для данных целей внутри цикла создан массив строк, состоящий из 12 ячеек, под названием values. В каждую ячейку будет добавлена информация из конкретного столбца таблицы «recipients». В методе logging() для добавления информации вместо вопросительных знаков мы использовали метод setString(). Сейчас же нам потребуется его противоположность – метод getString(). По названию становится ясно, что он возвращает строковое значение. Данное значение предаем в ячейку массива values. Для этого достаточно только передать в метод getString() порядковый номер колонки таблицы в БД. Так как в 11 столбцах таблицы используется тип данных String, метод getString() вполне успешно работает. Но мы получаем данные из 12 столбцов. И первым из них является поле «id». Поскольку данное поле имеет тип данных int, метод getString() к нему не применить. Решить данную проблему получилось с помощью перевода целочисленного значения в строку. Я использовал класс Integer и его метод toString(), поместив внутрь метода полученные целочисленные данные. Сами данные получаем с помощью метода getInt() с указанием порядкового номера столбца.

Integer.toString(resSet.getInt(1))

В конечном итоге на каждую итерацию цикла while мы получаем массив строк с 12 элементами типа String. Данный массив внутри итерации добавляется в коллекцию allData, созданную ранее. После окончания работы цикла на руках остается коллекция с некоторым количеством массивов строк. Данная коллекция и возвращается по итогу работы метода.

Применять созданный метод мы будем в классе ControllerList. Именно он отвечает за работу страницы, которая открывается после нажатия кнопки «Список созданных». Данная страница состоит из таблицы, полностью повторяющей своей структурой таблицу из базы данных в MySQL Workbench. В классе ControllerList так же как и в остальных контроллер-классах есть метод initialize(), отвечающий за поведение программы при открытии страницы или взаимодействие с какими-то элементами на странице. Нам он также понадобится, но чуть позже. Сперва создадим метод, который будет разбивать массив строк на отдельные ячейки и добавлять их в интерфейс таблицы. Для этого создаем объект класса ObservableList, так же как делали это в классе ControllerFilling. Имя ему будет list. Только сейчас тип данных не String, а созданный нами Client. И варианты в конце мы тоже не добавляем.

Затем требуется разработать метод добавления записей из базы данных в list. Прописываем void метод initData(). В него ничего не передаем. Внутри метода создаем экземпляр класса DatabaseHandler. Затем создаем коллекцию массивов строк и кладем в неё результат работы метода getClient().

List<String[]> tableValues = dbHandler.getClient();

Мы получили необходимые данные в нужном месте. Теперь осталось разделить их на части и подставить в подходящие ячейки таблицы. Для подобных целей используем стандартный цикл for. Он будет работать до тех пор, пока коллекция tableValues не закончится. Внутри цикла в коллекцию list последовательно добавятся все записи из БД. Но не просто добавятся целиком, а разделятся на части. Всё благодаря конструктору класс Client. Конструктор принимает 12 значений. Поэтому одну запись мы делим на 12 частей благодаря методу get(i)[] с указанием внутри квадратных скобок индекс нужного столбца. По итогу после каждой итерации цикла for в list добавляется 12 элементов, полностью совпадающие по порядку с полями графической таблицы tableView.

Когда все готово, в методе initialize() прописываем необходимые параметры для добавления полученных элементов в таблицу. Для каждого из 12 столбцов на странице ViewList.fxml прописываем fx:id с применением метода setCellValueFactory(). Указанный метод предоставляет способ отображения данных в ячейке. Внутри прописывается new PropertyValueFactory с указанием двух типов данных: Client и String. Ведь именно с этими типами данных сталкивается программа при добавлении данных в таблицу tableView. И в конце в качестве ссылки прописывается названием подходящей под элемент страницы переменной из класса Client.

Перед этим в самом начале благодаря автоматическому запуску метода initialize() реализуется метод initData(), тем самым заполняя ObservableList коллекцию под названием list.

Последним шагом в работе метода initialize() станет внесение всех данных из коллекции list в таблицу tableView в качестве записи полей. Реализуется подобное действие благодаря методу setItems, уже использованному нами ранее в коде программы.

tableView.setItems(list);

**Заключение**

В заключении стоит отметить, что все поставленные задачи были выполнены и реализованы в конкретных решениях. По итогу получилась рабочая программа, включающая в себя 4 графических интерфейса, базу данных, созданные документы формата docx, а также множество сторонних библиотек, подключенных к проекту.

Удалось создать 4 страницы приложения, 3 из которых имеют вполне определенный функционал. Первая страница предоставляет выбор двух возможных вариантов развития событий: отправиться создавать документы на основе шаблонов или просмотреть список уже созданных документов с данными получателей услуг.

Вторая страница имеет самый большой функционал. Она состоит из набора элементов для ввода информации. Но это только внешняя оболочка. Внутри прописана вся логика работы с введенной информацией. Во-первых, она вставляется в шаблоны документов Microsoft Word. Во-вторых, эти данные добавляются в таблицу подключенной СУБД MySQL Workbench.

Третья страница представляет собой таблицу записями из базы данных. Все процессы при открытии данной страницы запускаются автоматически. Таблица не имеет заданного количества строк, поэтому заполнять её можно до того момента, как закончится память.

Четвертая страница появляется после второй и сигнализирует об успешности заполнения документов.

Как показала проведенная работа, javaFX является достаточно удобной и многофункциональной платформой для создания приложений с насыщенным графическим интерфейсом. Вместе с программой визуального средства разработки Scene Builder становится возможным создание самого разнообразного визуала страницы.

Отвечая на цель данного исследования можно сказать, о том, что реализация функций для работы с базой данных и шаблонами документов прописана в полном объеме.

Код приложения разделен сразу на несколько классов, интерфейсов и fxml файлов. Каждый класс отвечает за строго определенный набор функций. Помогают им в этом интерфейсы. А за визуальную составляющую страниц отвечают fxml файлы. За счет подобного разделения соблюдаются принципы SOLID.

Проект придерживается структуры MVC. Вся логика приложения разделена на три основные части: Model – классы, отвечающие за основную логику программы; View – fxml файлы, формирующие внешний вид приложения и определяющие взаимодействие с пользователем; Controller – классы, связывающие в себе логику поведения из Model классов и внешний интерфейс приложения.

Внутри проекта также находятся сторонние элементы, поддерживающие работу программы. Это шаблоны документов docx, картинка для фона страниц, а также дополнительные иконки. Без иконок можно было обойтись, но с ними приложение выглядит чуть веселее.

В дальнейшем я планирую дорабатывать данную программу и довести её до полноценного приложения, увеличив функционал и усовершенствовав визуальную составляющую.

**Список использованной литературы и источников**

**Список источников**

1. Генерация файлов Word в Apache POI// [Электронный ресурс] - https://habr.com/ru/articles/503444/.

2. Интерфейс ResultSet //[Электронный ресурс] - https://java-online.ru/jdbc-resultset.xhtml.

3. Руководство JavaFX для начинающих - Hello JavaFX

// [Электронный ресурс] - https://betacode.net/10623/javafx-tutorial-for-beginners.

4. Учебник по набору результатов Java// [Электронный ресурс] - https://ru.linux-console.net/?p=6349#gsc.tab=0.

5. Apache POI - HWPF and XWPF - Java API to Handle Microsoft Word Files

// [Электронный ресурс] - https://poi.apache.org/components/document/.

6. API Java для документов Word OOXML// [Электронный ресурс] - https://products.fileformat.com/ru/word-processing/java/apache-poi-xwpf/.

7. ChoiceBox (JavaFX 8)// [Электронный ресурс]- https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/ChoiceBox.htm.

8. Introducing JDBC// [Электронный ресурс] - https://docs.oracle.com/en/database/oracle/oracle-database/19/jjdbc/introducing-JDBC.html#GUID-864DB502-5E50-4044-8132-33D6AAF8927A.

9. IOException Класс// [Электронный ресурс] - https://learn.microsoft.com/ru-ru/dotnet/api/java.io.ioexception?view=xamarin-android-sdk-13.

10. JavaFX – Guide: Изучение JavaFX// [Электронный ресурс] - http://javafx-guide.blogspot.com/p/javafx.html.

11. ObservableList (JavaFX 8)// [Электронный ресурс]- https://docs.oracle.com/javase/8/javafx/api/javafx/collections/ObservableList.html.

12. SelectionModel (JavaFX 8.0)// [Электронный ресурс] - https://spec-zone.ru/RU/Java/FX/8/docs/api/javafx/scene/control/SelectionModel.html.

13. XWPFTable (POI API Documentation)// [Электронный ресурс] – https://poi.apache.org/apidocs/dev/org/apache/poi/xwpf/usermodel/XWPFTable.html.

14. XWPFTableCell (POI API Documentation)// [Электронный ресурс] – https://poi.apache.org/apidocs/dev/org/apache/poi/xwpf/usermodel/XWPFTableCell.html.

15. XWPFTableRow (POI API Documentation)// [Электронный ресурс] – https://poi.apache.org/apidocs/dev/org/apache/poi/xwpf/usermodel/XWPFTableRow.html.

**Список литературы**

1. Прохоренок Н.А., JavaFX/ Санкт-Петербург: «БХВ-Петербург», 2020 – 770 С.

2. Castillo C., JavaFX Scene Builder User Guide/ Oracle, 2012 – 45 С.

3. Klausen P., Development of applications with JavaFX/ «Bookboon», 2018 – 79 C.

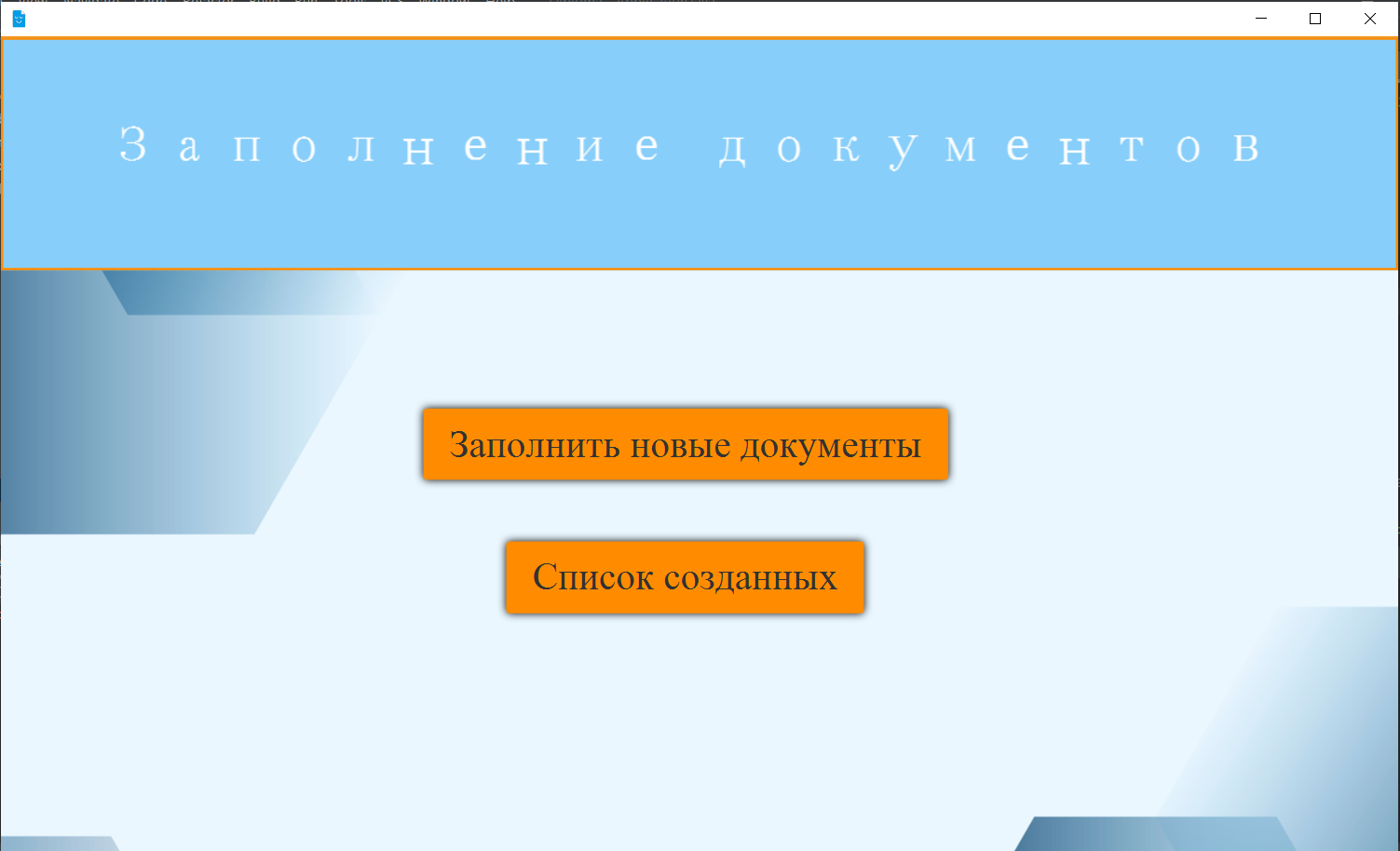
4. Potts J., Hildebrandt N., Gordon J., Castillo C., JavaFX Getting Started with JavaFX/ Oracle, 2014 – 68 C.

5. Wesley A., JDBC™ API Tutorial and Reference/ 2003 - 1345 С.

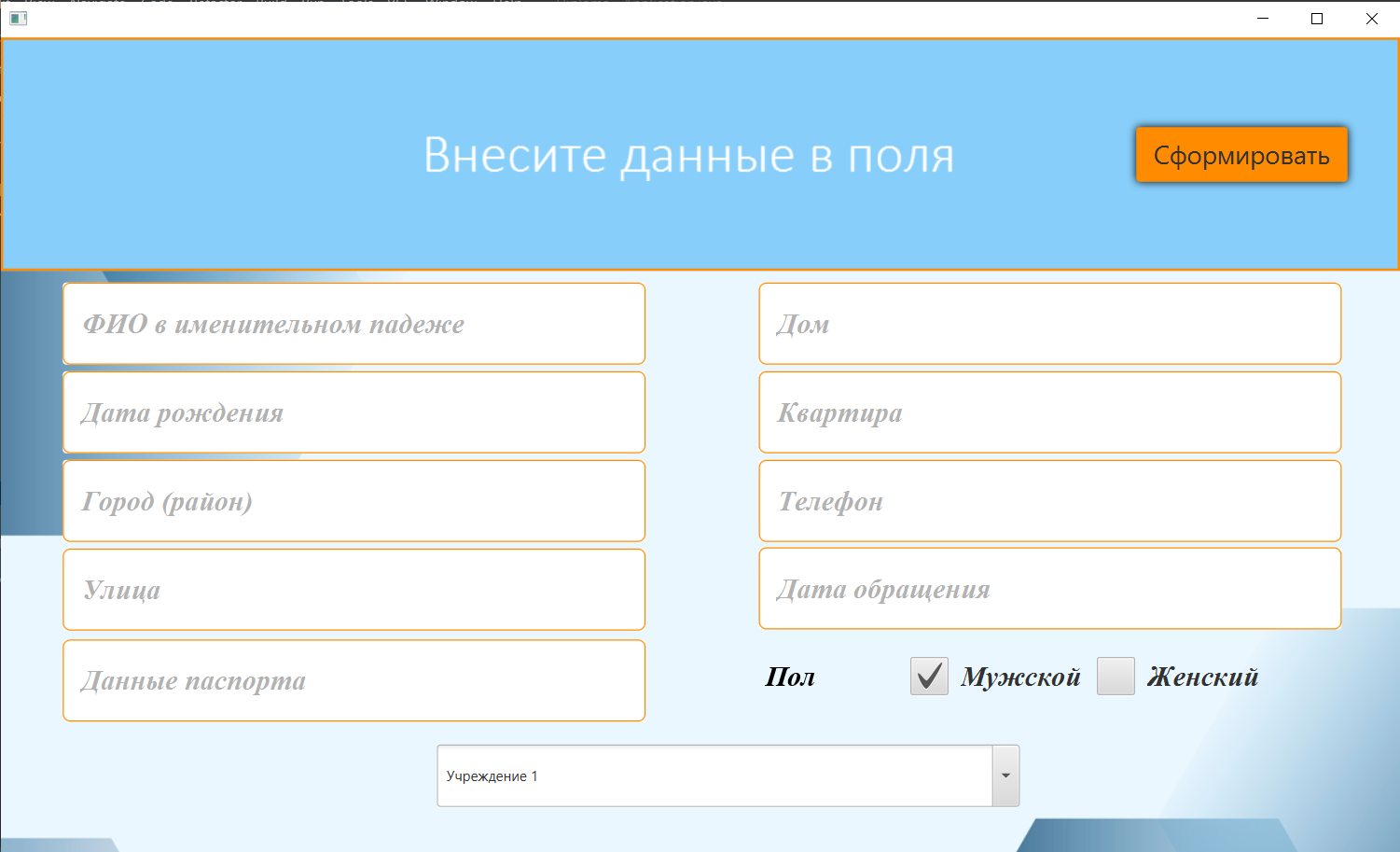
6. Apache\_poi\_word\_tutorial/ 2017 - 7 С.

**Приложения**

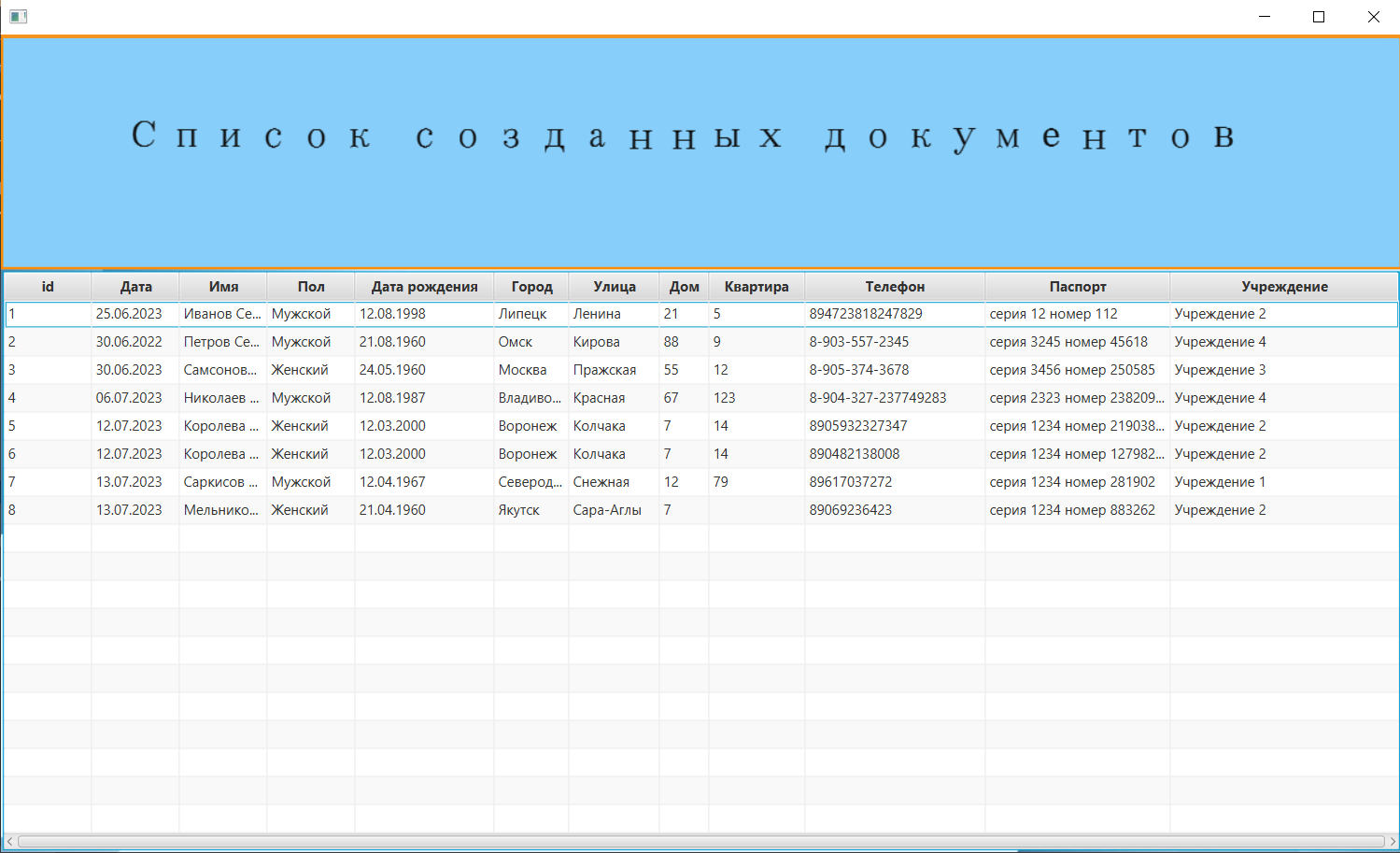
**Приложение № 1 (**View.fxml**)**

****

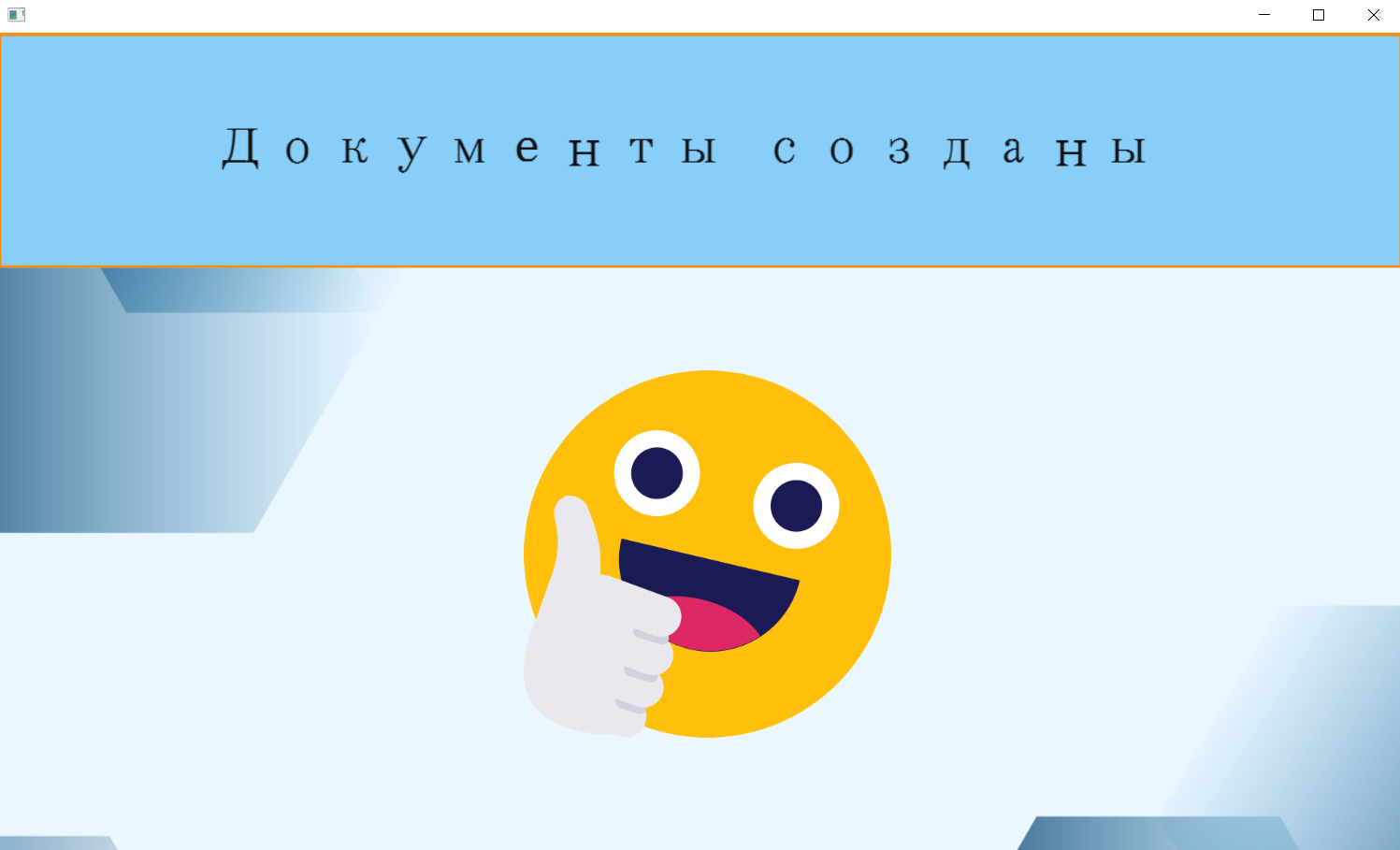
**Приложение № 2 (**ViewFilling.fxml**)**

****

**Приложение № 3 (**ViewList.fxml**)**

****

**Приложение № 4 (**ViewAfterFilling.fxml**)**



**Приложение № 5 (**class Application**)**

package com.example.diploma;  
  
import javafx.fxml.FXMLLoader;  
import javafx.scene.Scene;  
import javafx.scene.image.Image;  
import javafx.stage.Stage;  
  
import java.io.IOException;  
  
public class Application extends javafx.application.Application {  
  
 */\*\*  
 \* Метод для запуска приложения. Создание контейнера для отображения всех графических элементов  
 \*/* @Override  
 public void start(Stage stage) throws IOException {  
 FXMLLoader fxmlLoader = new FXMLLoader(Application.class.getResource("View.fxml")); // Загрузка интерфейса приложения, содержащегося в файле View.fxml  
 Scene scene = new Scene(fxmlLoader.load(), 1200, 700); // Размер окна приложения  
 stage.getIcons().add(new Image("file:/C:/Users/александр/Desktop/ДИПЛОМ Geekbrains/ОСНОВНОЕ - ПРОЕКТ/Diploma/src/main/resources/com/example/diploma/Pictures/Icon.png")); // Иконка приложения  
 stage.setScene(scene);  
 stage.show();  
 }  
  
 public static void main(String[] args) {  
 *launch*();  
 }  
}

**Приложение № 6 (**class Client**)**

package com.example.diploma;  
  
*/\*\*  
 \* Класс с данными получателя услуг, которые будут записаны в документы  
 \*/*public class Client {  
 private String id;  
 private String date;  
 private String name;  
 private String gender;  
 private String birth;  
 private String city;  
 private String street;  
 private String house;  
 private String flat;  
 private String phone;  
 private String passport;  
 private String institution;  
  
 */\*\*  
 \* Конструктор для метода initData класса ControllerList  
 \*/* public Client(String id, String date, String name, String gender, String birth, String city, String street,String house, String flat, String phone, String passport, String institution) {  
 this.id = id;  
 this.date = date;  
 this.name = name;  
 this.gender = gender;  
 this.birth = birth;  
 this.city = city;  
 this.street = street;  
 this.house = house;  
 this.flat = flat;  
 this.phone = phone;  
 this.passport = passport;  
 this.institution = institution;  
 }  
  
 */\*\*  
 \* Конструктор для методов insertToFormString и insertToFormTable класса WordDocuments  
 \*/* public Client(String date, String name, String gender, String birth, String city, String street,String house, String flat, String phone, String passport, String institution) {  
  
 this.date = date;  
 this.name = name;  
 this.gender = gender;  
 this.birth = birth;  
 this.city = city;  
 this.street = street;  
 this.house = house;  
 this.flat = flat;  
 this.phone = phone;  
 this.passport = passport;  
 this.institution = institution;  
 }  
  
 public String getDate() {  
 return date;  
 }  
  
 public void setDate(String date) {  
 this.date = date;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getGender() {  
 return gender;  
 }  
  
 public void setGender(String gender) {  
 this.gender = gender;  
 }  
  
 public String getBirth() {  
 return birth;  
 }  
  
 public void setBirth(String birth) {  
 this.birth = birth;  
 }  
  
 public String getCity() {  
 return city;  
 }  
  
 public void setCity(String city) {  
 this.city = city;  
 }  
  
 public String getStreet() {  
 return street;  
 }  
  
 public void setStreet(String street) {  
 this.street = street;  
 }  
  
 public String getHouse() {  
 return house;  
 }  
  
 public void setHouse(String house) {  
 this.house = house;  
 }  
  
 public String getFlat() {  
 return flat;  
 }  
  
 public void setFlat(String flat) {  
 this.flat = flat;  
 }  
  
 public String getPhone() {  
 return phone;  
 }  
  
 public void setPhone(String phone) {  
 this.phone = phone;  
 }  
  
 public String getPassport() {  
 return passport;  
 }  
  
 public void setPassport(String passport) {  
 this.passport = passport;  
 }  
  
 public String getInstitution() {  
 return institution;  
 }  
  
 public void setInstitution(String institution) {  
 this.institution = institution;  
 }  
  
 public String getId() {  
 return id;  
 }  
  
 public void setId(String id) {  
 this.id = id;  
 }  
}

**Приложение № 7 (**class Controller**)**

package com.example.diploma;  
  
import java.io.IOException;  
import java.net.URL;  
import java.util.ResourceBundle;  
  
import javafx.fxml.FXML;  
import javafx.fxml.FXMLLoader;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.stage.Stage;  
  
*/\*\*  
 \* Класс для регулирования приветственной страницы. В нем есть две кнопки для перехода либо к  
 \* заполнению документов,либо к просмотру уже созданных пакетов документов из базы данных  
 \*/*public class Controller {  
  
 @FXML  
 private ResourceBundle resources;  
  
 @FXML  
 private URL location;  
  
 @FXML  
 private Button sendDocuments;  
  
 @FXML  
 private Button sentList;  
  
 */\*\*  
 \* Переход на две другие страницы  
 \*/* @FXML  
 private void initialize() {  
  
 sendDocuments.setOnAction(event -> { // Действия при нажатии на кнопку с ip SendDocuments  
  
 sendDocuments.getScene().getWindow().hide();  
  
 FXMLLoader loader = new FXMLLoader();  
 loader.setLocation(getClass().getResource("ViewFilling.fxml")); // Переход на страницу заполнения документов  
  
 try {  
 loader.load();  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
  
 Parent root = loader.getRoot();  
 Stage stage = new Stage();  
 stage.setScene(new Scene(root));  
 stage.showAndWait();  
 });  
  
 sentList.setOnAction(event -> { // Действия при нажатии на кнопку с ip SentList  
 sentList.getScene().getWindow().hide();  
  
 FXMLLoader loader = new FXMLLoader();  
 loader.setLocation(getClass().getResource("ViewList.fxml")); // Переход на страницу со списком уже заполненных документов  
  
 try {  
 loader.load();  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
  
 Parent root = loader.getRoot();  
 Stage stage = new Stage();  
 stage.setScene(new Scene(root));  
 stage.showAndWait();  
 });  
 }  
}

**Приложение № 8 (**class ControllerAfterFilling**)**

package com.example.diploma;  
  
import java.net.URL;  
import java.util.ResourceBundle;  
  
import javafx.fxml.FXML;  
  
*/\*\*  
 \* Класс, который демонстрирует страницу после создания докуметов  
 \*/*public class ControllerAfterFilling {  
 @FXML  
 private ResourceBundle resources;  
 @FXML  
 private URL location;  
 @FXML  
 private void initialize() {}  
}

**Приложение № 9 (**class ControllerFilling**)**

package com.example.diploma;  
  
import java.io.IOException;  
import java.net.URL;  
import java.util.ResourceBundle;  
  
import com.example.diploma.Database.DatabaseHandler;  
import javafx.collections.FXCollections;  
import javafx.collections.ObservableList;  
import javafx.fxml.FXML;  
import javafx.fxml.FXMLLoader;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.scene.control.CheckBox;  
import javafx.scene.control.ComboBox;  
import javafx.scene.control.TextField;  
import javafx.stage.Stage;  
import org.apache.poi.openxml4j.exceptions.InvalidFormatException;  
  
*/\*\*  
 \* Класс заполнения данных, которые будут автоматически вставлены в шаблоны документов  
 \*/*public class ControllerFilling {  
  
 @FXML  
 private ResourceBundle resources;  
  
 @FXML  
 private URL location;  
  
 @FXML  
 private TextField birth;  
  
 @FXML  
 private TextField city;  
  
 @FXML  
 private Button dataSend;  
  
 @FXML  
 private TextField date;  
  
 @FXML  
 private TextField flat;  
  
 @FXML  
 private TextField house;  
  
 @FXML  
 private ComboBox<String> institution;  
  
 //Создание вариантов выбора для элемента ComboBox <String> institution  
 private ObservableList<String> CBOptions = FXCollections.*observableArrayList*("Учреждение 1", "Учреждение 2", "Учреждение 3", "Учреждение 4");  
 private ComboBox<String> selectedItem = new ComboBox<String>(CBOptions);  
  
 @FXML  
 private CheckBox male;  
  
 @FXML  
 private TextField name;  
  
 @FXML  
 private TextField passport;  
  
 @FXML  
 private TextField phone;  
  
 @FXML  
 private TextField street;  
  
 @FXML  
 private void initialize() {  
  
 institution.setValue("Учреждение 1"); //Установка варианта "Учреждение 1" по умолчанию для элемента ComboBox <String> institution  
 institution.setItems(CBOptions);  
  
 */\*\*  
 \* Действия при нажатии на кнопку с ip dataSend  
 \*/* dataSend.setOnAction(event -> {  
  
 String date = this.date.getText();  
 String name = this.name.getText();  
 String gender = "";  
 String birth = this.birth.getText();  
 String city = this.city.getText();  
 String street = this.street.getText();  
 String house = this.house.getText();  
 String flat = this.flat.getText();  
 String phone = this.phone.getText();  
 String passport = this.passport.getText();  
 String institution = this.institution.getSelectionModel().getSelectedItem();  
  
 if (male.isSelected()) gender = "Мужской";  
 else gender = "Женский";  
  
 Client client = new Client(date, name, gender, birth, city, street, house, flat, phone, passport, institution);  
  
 WordDocuments wd = new WordDocuments(); //Внесение данных в шаблоны Word документов  
 try {  
 wd.insertToFormString(client);  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 } catch (InvalidFormatException e) {  
 throw new RuntimeException(e);  
 }  
  
 try {  
 wd.insertToFormTable(client);  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 } catch (InvalidFormatException e) {  
 throw new RuntimeException(e);  
 }  
  
 DatabaseHandler db = new DatabaseHandler(); // Создание экземпляра класса DatabaseHandler для вызова метода logging  
 db.logging(date, name, gender, birth, city, street, house, flat, phone, passport, institution); // Занесение данных в базу данных в MySQL

dataSend.getScene().getWindow().hide(); // Закрытие окна  
 FXMLLoader loader = new FXMLLoader();  
 loader.setLocation(getClass().getResource("ViewAfterFilling.fxml")); // Переход на страницу с уведомлением о завершении заполнения документов  
  
 try {  
 loader.load();  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
  
 Parent root = loader.getRoot();  
 Stage stage = new Stage();  
 stage.setScene(new Scene(root));  
 stage.showAndWait();  
 });  
 }  
}

**Приложение № 10 (**class ControllerList**)**

package com.example.diploma;  
  
import java.net.URL;  
import java.util.List;  
import java.util.ResourceBundle;  
  
import com.example.diploma.Database.DatabaseHandler;  
import javafx.collections.FXCollections;  
import javafx.collections.ObservableList;  
import javafx.fxml.FXML;  
import javafx.scene.control.TableColumn;  
import javafx.scene.control.TableView;  
import javafx.scene.control.cell.PropertyValueFactory;  
  
*/\*\*  
 \* Класс вывода всех получателей услуг из базы данных на экран приложения  
 \*/*public class ControllerList {  
  
 @FXML  
 private ResourceBundle resources;  
  
 @FXML  
 private URL location;  
  
 @FXML  
 private TableColumn<Client, String> fieldBirth;  
  
 @FXML  
 private TableColumn<Client, String> fieldCity;  
  
 @FXML  
 private TableColumn<Client, String> fieldDate;  
  
 @FXML  
 private TableColumn<Client, String> fieldFlat;  
  
 @FXML  
 private TableColumn<Client, String> fieldGender;  
  
 @FXML  
 private TableColumn<Client, String> fieldHoume;  
  
 @FXML  
 private TableColumn<Client, Integer> fieldID;  
  
 @FXML  
 private TableColumn<Client, String> fieldInstitution;  
  
 @FXML  
 private TableColumn<Client, String> fieldName;  
  
 @FXML  
 private TableColumn<Client, String> fieldPassport;  
  
 @FXML  
 private TableColumn<Client, String> fieldPhone;  
  
 @FXML  
 private TableColumn<Client, String> fieldStreet;  
  
 @FXML  
 private TableView<Client> tableView;  
  
 private ObservableList<Client> list = FXCollections.*observableArrayList*(); //Создание списка, в который по очереди будут добавлены все строки из базы данных  
  
 */\*\*  
 \* Метод добавления строк из базы данных в список list  
 \*/* private void initData() {  
 DatabaseHandler dbHandler = new DatabaseHandler();  
 List<String[]> tableValues = dbHandler.getClient();  
 for (int i = 0; i < tableValues.size(); i++) {  
 list.add(new Client(tableValues.get(i)[0], tableValues.get(i)[1], tableValues.get(i)[2], tableValues.get(i)[3],  
 tableValues.get(i)[4], tableValues.get(i)[5], tableValues.get(i)[6], tableValues.get(i)[7],  
 tableValues.get(i)[8], tableValues.get(i)[9], tableValues.get(i)[10], tableValues.get(i)[11]));  
 }  
 }  
  
 @FXML  
 private void initialize() {  
  
 initData(); // При попадании на страницу метод initData() запускается автоматически  
  
 fieldID.setCellValueFactory(new PropertyValueFactory<Client, Integer>("id"));  
 fieldDate.setCellValueFactory(new PropertyValueFactory<Client, String>("date"));  
 fieldName.setCellValueFactory(new PropertyValueFactory<Client, String>("name"));  
 fieldGender.setCellValueFactory(new PropertyValueFactory<Client, String>("gender"));  
 fieldBirth.setCellValueFactory(new PropertyValueFactory<Client, String>("birth"));  
 fieldCity.setCellValueFactory(new PropertyValueFactory<Client, String>("city"));  
 fieldStreet.setCellValueFactory(new PropertyValueFactory<Client, String>("street"));  
 fieldHoume.setCellValueFactory(new PropertyValueFactory<Client, String>("house"));  
 fieldFlat.setCellValueFactory(new PropertyValueFactory<Client, String>("flat"));  
 fieldPhone.setCellValueFactory(new PropertyValueFactory<Client, String>("phone"));  
 fieldPassport.setCellValueFactory(new PropertyValueFactory<Client, String>("passport"));  
 fieldInstitution.setCellValueFactory(new PropertyValueFactory<Client, String>("institution"));  
  
 tableView.setItems(list); // Внесение элементов из списка list в ячейки таблицы tableView  
 }  
}

**Приложение № 11 (**class WordDocuments**)**

package com.example.diploma;  
  
import com.example.diploma.Interfaces.InsertToWord;  
import org.apache.poi.openxml4j.opc.OPCPackage;  
import org.apache.poi.xwpf.usermodel.\*;  
  
import java.io.\*;  
import java.util.List;  
  
*/\*\*  
 \* Класс с методами добавления данных из полей класса ControllerFilling в шаблоны документов  
 \*/*public class WordDocuments implements InsertToWord {  
  
 */\*\*  
 \* Метод для вставки данных в шаблоны документов без таблиц внутри  
 \*/* public void insertToFormString(Client client)  
 throws IOException, org.apache.poi.openxml4j.exceptions.InvalidFormatException {  
 try {  
 // Определение документа Word, который будет основой для добавления персональных данных  
 XWPFDocument doc = new XWPFDocument(  
 OPCPackage.*open*("C:\\Users\\александр\\Desktop\\ДИПЛОМ Geekbrains\\ОСНОВНОЕ - ПРОЕКТ\\Diploma\\src\\main\\java\\com\\example\\diploma\\Forms\\Форма № 1.docx"));

// Проход по параграфам внутри документа  
 for (XWPFParagraph p : doc.getParagraphs()) {  
 List<XWPFRun> runs = p.getRuns();  
 if (runs != null) {  
 for (XWPFRun r : runs) {  
 String text = r.getText(0);

// Замена найденного опорного слова на данные из формы заполнения (client.getDate())  
 if (text != null && text.contains("date")) {  
 text = text.replace("date", client.getDate());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("name")) {  
 text = text.replace("name", client.getName());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("male")) {  
 text = text.replace("male", client.getGender());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("birth")) {  
 text = text.replace("birth", client.getBirth());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("city")) {  
 text = text.replace("city", client.getCity());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("street")) {  
 text = text.replace("street", client.getStreet());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("house")) {  
 text = text.replace("house", client.getHouse());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("flat")) {  
 text = text.replace("flat", client.getFlat());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("phone")) {  
 text = text.replace("phone", client.getPhone());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("passport")) {  
 text = text.replace("passport", client.getPassport());  
 r.setText(text, 0);  
 }  
 }  
 }  
 }  
  
 // Создание нового документа на основе шаблона уже с заполеннными данными получателя услуг  
 doc.write(new FileOutputStream("C:\\Users\\александр\\Desktop\\new1.docx"));  
 } finally {  
 }  
 }  
  
 */\*\*  
 \* Метод для вставки данных в шаблоны документов c таблицами внутри  
 \*/* public void insertToFormTable(Client client)  
 throws IOException, org.apache.poi.openxml4j.exceptions.InvalidFormatException {  
 try {

// Определение документа Word, который будет основой для добавления персональных данных  
 XWPFDocument doc = new XWPFDocument(  
 OPCPackage.*open*("C:\\Users\\александр\\Desktop\\ДИПЛОМ Geekbrains\\ОСНОВНОЕ - ПРОЕКТ\\Diploma\\src\\main\\java\\com\\example\\diploma\\Forms\\Форма № 2.docx"));

// Проход по таблицам внутри документа  
 for (XWPFTable tbl : doc.getTables()) {  
 for (XWPFTableRow row : tbl.getRows()) {  
 for (XWPFTableCell cell : row.getTableCells()) {  
 for (XWPFParagraph p : cell.getParagraphs()) {  
 for (XWPFRun r : p.getRuns()) {  
 String text = r.getText(0);

// Замена найденного опорного слова на данные из формы заполнения (client.getDate())  
 if (text != null && text.contains("date")) {  
 text = text.replace("date", client.getDate());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("name")) {  
 text = text.replace("name", client.getName());  
 r.setText(text, 0);  
 }  
 if (text != null && text.contains("institution")) {  
 text = text.replace("institution", client.getInstitution());  
 r.setText(text, 0);  
 }  
 }  
 }  
 }  
 }  
 }  
  
 // Создание нового документа на основе шаблона уже с заполеннными данными получателя услуг  
 doc.write(new FileOutputStream("C:\\Users\\александр\\Desktop\\new2.docx"));  
 } finally {  
 }  
 }  
}

**Приложение № 12 (**class DatabaseConfigs**)**

package com.example.diploma.Database;  
  
*/\*\*  
 \* Класс предназначен для соединения с базой данных  
 \*/*public class DatabaseConfigs {  
 protected String dbHost = "localhost"; // IP-адрес 127.0.0.1 для подключения к БЛ  
 protected String dbPort = "3306"; // Порт подключения по умолчанию  
 protected String dbUser = "root"; // Учетная запись суперпользователя  
 protected String dbPass = "12345"; // Пароль для входа в учетную запись  
 protected String dbName = "diploma"; // Название БД  
}

**Приложение № 13 (**class DatabaseConst**)**

package com.example.diploma.Database;  
  
*/\*\*  
 \* Класс с названиями полей в таблице в MySQL  
 \*/*public class DatabaseConst {  
 public static final String *TABLE\_NAME* = "recipients";  
 public static final String *CLIENT\_DATE* = "client\_date";  
 public static final String *CLIENT\_NAME* = "client\_name";  
 public static final String *CLIENT\_GENDER* = "client\_gender";  
 public static final String *CLIENT\_BIRTH* = "client\_birth";  
 public static final String *CLIENT\_CITY* = "client\_city";  
 public static final String *CLIENT\_STREET* = "client\_street";  
 public static final String *CLIENT\_HOUSE* = "client\_house";  
 public static final String *CLIENT\_FLAT* = "client\_flat";  
 public static final String *CLIENT\_PHONE* = "client\_phone";  
 public static final String *CLIENT\_PASSPORT* = "client\_passport";  
 public static final String *CLIENT\_INSTITUTION* = "institution";  
}

**Приложение № 14 (**class DatabaseHandler**)**

package com.example.diploma.Database;  
  
import com.example.diploma.Interfaces.ConnectionDB;  
import com.example.diploma.Interfaces.InsertDB;  
import com.example.diploma.Interfaces.SelectDB;  
  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
import java.sql.SQLException;  
import java.sql.ResultSet;  
import java.util.ArrayList;  
import java.util.List;  
  
*/\*\*  
 \* Класс с метода взаимодействия с базой данных в MySQL  
 \*/*public class DatabaseHandler extends DatabaseConfigs implements ConnectionDB, InsertDB, SelectDB {  
  
 */\*\*  
 \* Метод для подключения к БД  
 \*/* public Connection getDbConnection() throws ClassNotFoundException, SQLException {  
  
 Connection dbConnection; // Экземпляр класса Connection для подключения к БД  
  
 String connectionString = "jdbc:mysql://" + dbHost + ":" + dbPort + "/" + dbName; // jdbc - стандарт взаимодействия Java-приложений с различными СУБД

Class.*forName*("com.mysql.jdbc.Driver");  
 dbConnection = DriverManager.*getConnection*(connectionString, dbUser, dbPass);  
 return dbConnection;  
 }  
  
 */\*\*  
 \* Метод внесения данных в БД  
 \*/* public void logging(String date, String name, String gender, String birth, String city, String street,String house, String flat, String phone, String passport, String institution) {  
  
 String insert = "INSERT INTO " + DatabaseConst.*TABLE\_NAME* + "(" + DatabaseConst.*CLIENT\_DATE* + "," + DatabaseConst.*CLIENT\_NAME* + "," + DatabaseConst.*CLIENT\_GENDER* + "," + DatabaseConst.*CLIENT\_BIRTH* + "," + DatabaseConst.*CLIENT\_CITY* + "," + DatabaseConst.*CLIENT\_STREET* + "," + DatabaseConst.*CLIENT\_HOUSE* + "," + DatabaseConst.*CLIENT\_FLAT* + "," + DatabaseConst.*CLIENT\_PHONE* + "," + DatabaseConst.*CLIENT\_PASSPORT* + "," + DatabaseConst.*CLIENT\_INSTITUTION* + ")" + " VALUES(?,?,?,?,?,?,?,?,?,?,?)";  
  
 try {  
 PreparedStatement ps = getDbConnection().prepareStatement(insert);  
 ps.setString(1, date);  
 ps.setString(2, name);  
 ps.setString(3, gender);  
 ps.setString(4, birth);  
 ps.setString(5, city);  
 ps.setString(6, street);  
 ps.setString(7, house);  
 ps.setString(8, flat);  
 ps.setString(9, phone);  
 ps.setString(10, passport);  
 ps.setString(11, institution);  
  
 ps.executeUpdate();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* Метод извлечения информации из БД  
 \*/* public List<String[]> getClient() {  
  
 ResultSet resSet; // ResultSet - класс, позволяющий получить и содержать результат запроса к БД  
  
 List<String[]> allData = new ArrayList<>(); // Набор строк, каждая из которых содержит информацию о получателе услуг (несколько столбцов из таблицы в БД)  
  
 String select = "SELECT \* FROM " + DatabaseConst.*TABLE\_NAME*;  
 try {  
 PreparedStatement ps = getDbConnection().prepareStatement(select);  
 resSet = ps.executeQuery();  
  
 while (resSet.next()) {  
  
 String[] values = new String[12];  
  
 values[0] = Integer.*toString*(resSet.getInt(1));  
 values[1] = resSet.getString(2);  
 values[2] = resSet.getString(3);  
 values[3] = resSet.getString(4);  
 values[4] = resSet.getString(5);  
 values[5] = resSet.getString(6);  
 values[6] = resSet.getString(7);  
 values[7] = resSet.getString(8);  
 values[8] = resSet.getString(9);  
 values[9] = resSet.getString(10);  
 values[10] = resSet.getString(11);  
 values[11] = resSet.getString(12);  
  
 allData.add(values);  
 }  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 return allData;  
 }  
}

**Приложение № 15 (**interface ConnectionDB**)**

package com.example.diploma.Interfaces;  
  
import java.sql.Connection;  
import java.sql.SQLException;  
  
public interface ConnectionDB {  
 public Connection getDbConnection() throws ClassNotFoundException, SQLException;  
}

**Приложение № 16 (**interface InsertDB**)**

package com.example.diploma.Interfaces;  
  
public interface InsertDB {  
 public void logging(String date, String name, String gender, String birth, String city, String street,  
 String house, String flat, String phone, String passport, String institution);  
}

**Приложение № 17 (**interface InsertToWord**)**

package com.example.diploma.Interfaces;  
  
import com.example.diploma.Client;  
  
import java.io.IOException;  
  
public interface InsertToWord {  
 void insertToFormString(Client client)  
 throws IOException, org.apache.poi.openxml4j.exceptions.InvalidFormatException;  
  
 void insertToFormTable(Client client)  
 throws IOException, org.apache.poi.openxml4j.exceptions.InvalidFormatException;  
}

**Приложение № 18 (**interface SelectDB**)**

package com.example.diploma.Interfaces;  
  
import java.util.List;  
  
public interface SelectDB {  
 public List<String[]> getClient();  
}

**Приложение № 19 (**шаблон документа № 1**)**

Форма № 1

**date** №

(дата составления)

1. Фамилия, имя, отчество (при наличии) **name**

2. Пол **male** 3. Дата рождения **birth**

4. Адрес места жительства:

город (район) **city**

улица **street** дом **house**

квартира **flat** телефон **phone**

5. Серия, номер паспорта или данные иного документа, удостоверяющего личность, дата выдачи этих документов, наименование выдавшего органа

**passport**

С содержанием индивидуальной программы предоставления социальных услуг **согласен(а)**

(подпись получателя услуг или его законного представителя) (расшифровка подписи)

Должность

(должность и подпись лица, уполномоченного на подписание)

М.П.

**Приложение № 20 (**шаблон документа № 2**)**

Форма № 2

|  |  |  |
| --- | --- | --- |
|  | **date** | Дата обращения |
|  | Дата обработки |
| ФИО **name**    (Ф.И.О. полностью)  обратился в **institution**  (наименование поставщика услуг)  за признанием нуждающимся(-щейся) в обслуживании.  После рассмотрения заявления и документов принято решение о признании заявителя  нуждающимся(-щейся) в предоставлении услуг. | | |
| Должность | \_\_\_\_\_\_\_\_  (подпись) (расшифровка подписи) | |
| М.П. | | |