|  |  |  |
| --- | --- | --- |
|  |  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных технологий

**Отчет по практической работе №7**

по дисциплине «Разработка мобильных приложений»

|  |  |
| --- | --- |
| **Выполнил:**  Студент группыИКБО-20-23 | Комисарик М.А. |
| **Проверил:**  Старший преподаватель кафедры МОСИТ | Шешуков Л.С. |

Москва 2025 г.СОДЕРЖАНИЕ

[**РТУ МИРЭА** 1](#_Toc196445865)

[1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ 3](#_Toc196445866)

[1.1 Service 3](#_Toc196445867)

[1.2 Диалоговые окна 18](#_Toc196445868)

[2 ПРАКТИЧЕСКОЕ ЗАДАНИЕ 34](#_Toc196445869)

[2.1 Разметка 34](#_Toc196445870)

[2.2 Реализация 36](#_Toc196445871)

[2.3 Тестирование 40](#_Toc196445872)

[ЗАКЛЮЧЕНИЕ 44](#_Toc196445873)

# ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ

## Service

Сервис в Android — это компонент приложения, предназначенный для выполнения длительных операций или работы с ресурсоёмкими задачами в фоновом режиме без предоставления пользовательского интерфейса. Сервисы продолжают работать в фоне даже когда пользователь переключается на другие приложения. Данный механизм идеально подходит для воспроизведения аудио, выполнения сетевых запросов, обработки данных и фонового мониторинга информации.

Все сервисы наследуются от класса Service и, по аналогии с Activity, сервис имеет свой жизненный цикл и методы, связанные с ним:

* onCreate(),
* onStartCommand(Intent intent, int flags, int startId),
* onBind(Intent intent),
* onUnbind(Intent intent),
* onDestroy().

На рисунке Рисунок 1 представлена иллюстрация жизненного цикла сервиса.

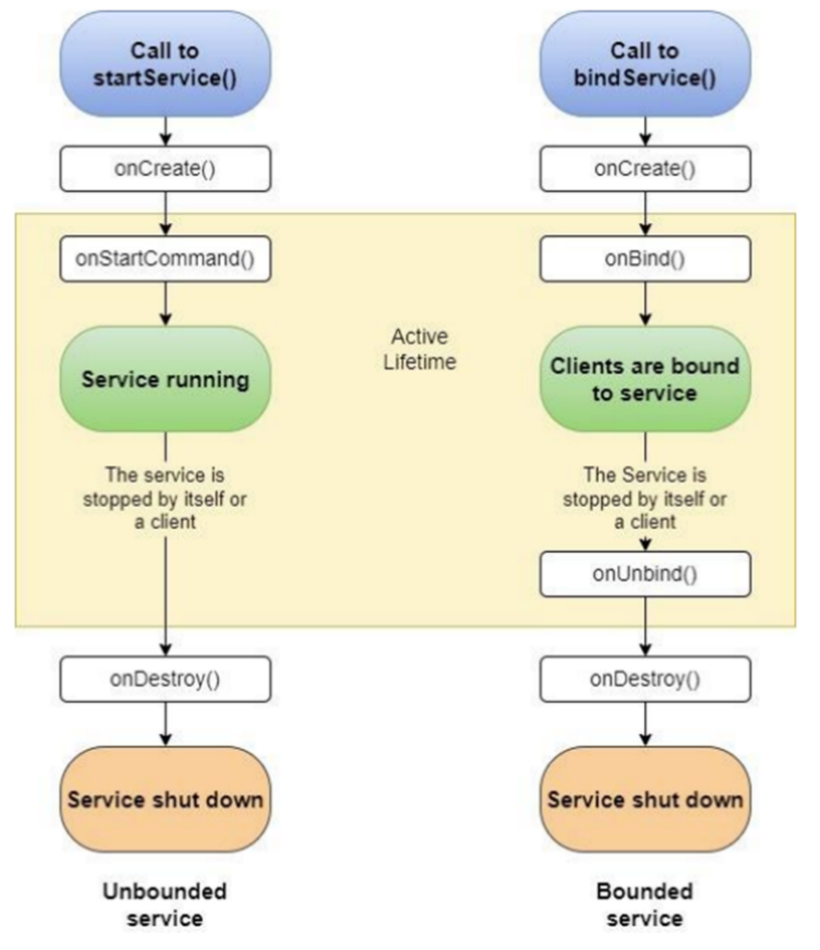


Рисунок – Жизненный цикл сервиса

Метод *onCreate()* вызывается при создании сервиса. Это первый вызов, который получает сервис, и он используется для однократной инициализации, такой как создание ресурсов, которые будут использоваться в течение всего времени существования сервиса. Метод onCreate() вызывается только один раз перед вызовом onStartCommand() или onBind().

Метод *onStartCommand()* вызывается каждый раз, когда компонент (например, Activity) запрашивает запуск сервиса через startService(). В этом методе сервис может выполнять любые операции, включая запуск потока для выполнения сложной задачи в фоне. Метод возвращает константу, указывающую, как система должна вести себя, если сервис уничтожается до того, как он завершит выполнение своей работы.

Метод *onBind()* вызывается, когда другой компонент хочет привязаться к сервису через bindService(). Если сервис не предоставляет интерфейс для клиентов, то он должен возвращать null. Метод onBind() вызывается только один раз для каждого клиента при первом связывании.

Метод *onUnbind()* вызывается, когда все клиенты отсоединились от определенного интерфейса сервиса. После этого вызова, если необходимо, сервис может остановить себя через stopSelf().

Метод *onDestroy()* вызывается, когда сервис больше не используется и собирается быть уничтоженным. Это последний вызов, который получает сервис, и он используется для освобождения ресурсов, таких как потоки, зарегистрированные приемники, обработчики и т.д.

В качестве примера, создадим сервис, который будет воспроизводить музыку. Предварительно загрузим медиафайл формата mp3 в наш проект.

Чтобы это сделать, необходимо создать в папке res папку raw. Именно в этой папке хранятся различные файлы, которые сохраняются в исходном виде. После этого нужно загрузить медиафайл в папку res/raw таким же способом, как добавляются изображения (Рисунок 2).

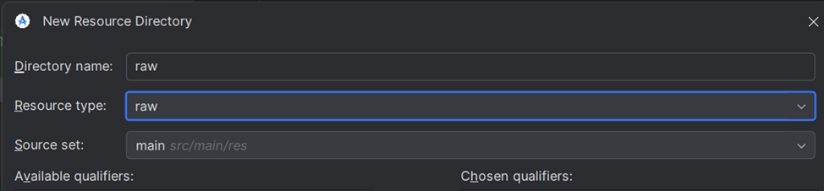


Рисунок – Создание директории для хранения ресурсов в исходном виде

После того, как файл добавлен нужно создать новый класс сервиса. Это можно сделать, нажав правой кнопкой мыши на "java"→ "New" → "Service" → "Service" (Рисунок 3).

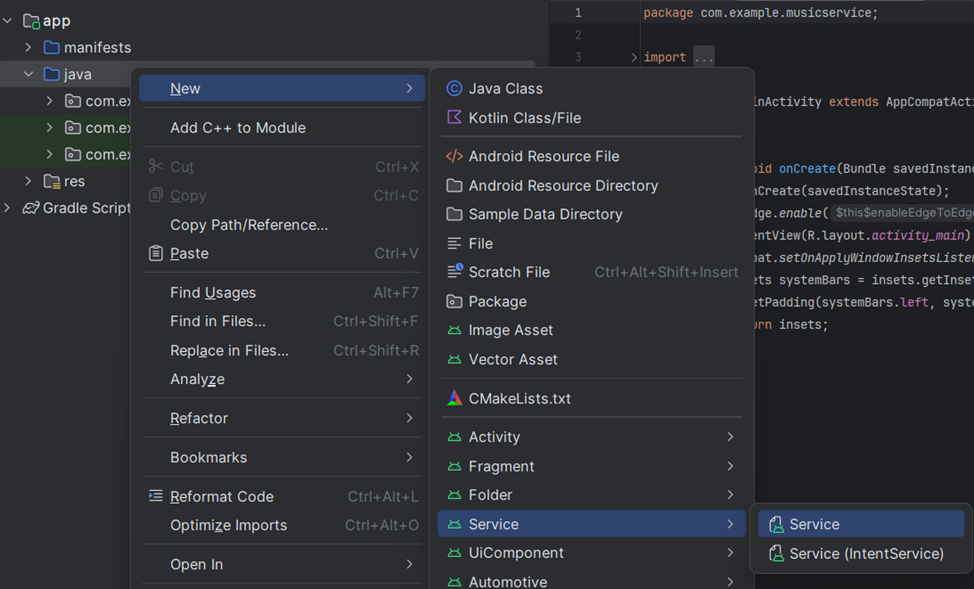


Рисунок – Создание нового сервиса через контекстное меню

При создании нового сервиса через диалоговое окно "New Android Component" необходимо указать имя класса (например, MusicService), которое будет использоваться для реализации сервиса. В этом же окне доступны настройки Enabled и Exported — флаги, определяющие базовое поведение сервиса. Значение Enabled включено по умолчанию, что позволяет системе создавать экземпляр сервиса, а параметр Exported отключен, ограничивая доступ к сервису только в рамках текущего приложения (Рисунок 4).

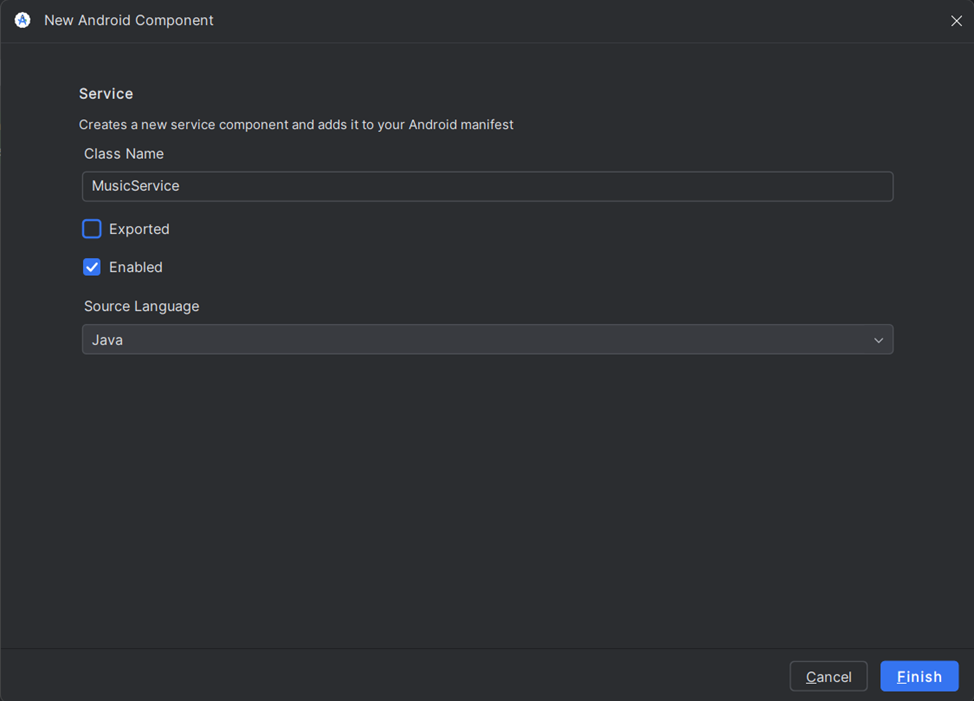


Рисунок – Окно создания нового сервиса

Созданный сервис с указанными настройками автоматически добавляется в файл манифеста.

Теперь нужно прописать логику работы сервиса (Рисунок 5).



Рисунок – Описание логики работы сервиса

Для воспроизведения музыкального файла сервис будет использовать компонент MediaPlayer.

В сервисе переопределяются четыре метода жизненного цикла. Метод onBind() возвращает null, так как наш сервис не должен иметь возможность привязки.

В методе onCreate() инициализируется медиа-проигрыватель с помощью музыкального ресурса, который добавлен в папку res/raw.

В методе onStartCommand() начинается воспроизведение.

Метод onStartCommand() может возвращать одно из значений, определяющих поведение сервиса при неожиданном завершении процесса системой:

* START\_STICKY: сервис автоматически перезапускается, но метод onStartCommand() вызывается с параметром Intent, равным null,
* START\_REDELIVER\_INTENT: сервис перезапускается с тем же объектом Intent, который был передан при последнем вызове,
* START\_NOT\_STICKY: сервис не перезапускается автоматически после завершения.

В методе onDestory() останавливается воспроизведение и освобождаются ресурсы.

При создании сервиса через контекстное меню Android Studio (New → Service → ...), объявление сервиса автоматически добавляется в файл AndroidManifest.xml. Однако если автоматическое добавление не сработало, требуется вручную прописать объявление сервиса в манифесте (Рисунок 6).

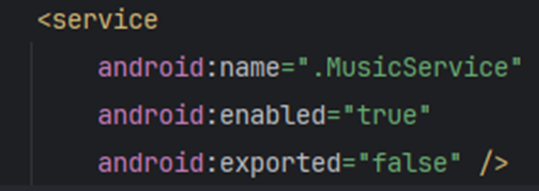


Рисунок – Объявление сервиса в манифесте

Регистрация сервиса осуществляется в узле <application> с помощью элемента <service>. Основные атрибуты:

* android:name — обязательный атрибут, содержащий полное имя класса сервиса (включая пакет),
* android:enabled — определяет, может ли система создавать сервис (по умолчанию "true"),
* android:exported — указывает, могут ли компоненты других приложений обращаться к сервису,
* android:permission — определяет разрешения, необходимые для взаимодействия с сервисом,
* android:process — задает имя процесса для запуска сервиса,
* android:isolatedProcess — если "true", сервис запускается в изолированном процессе с ограниченными правами,
* android:icon — значок сервиса, представляет собой ссылку на ресурс drawable,
* android:label — название сервиса, которое отображается пользователю.

В методе onCreate() класса MainActivity реализуем запуск фонового сервиса с помощью метода startService(). Создаем объект Intent, явно указывая целевой сервис MusicService.class, который требуется запустить. Передаем этот Intent в startService(), что инициирует создание и старт сервиса, если он не был ранее запущен.

В методе onDestroy() класса MainActivity обеспечиваем корректную остановку сервиса при завершении работы. Формируем аналогичный Intent с указанием того же класса сервиса MusicService.class и передаем его в метод stopService(). Это приводит к вызову метода onDestroy() сервиса и освобождению всех занимаемых им ресурсов (Рисунок 7).

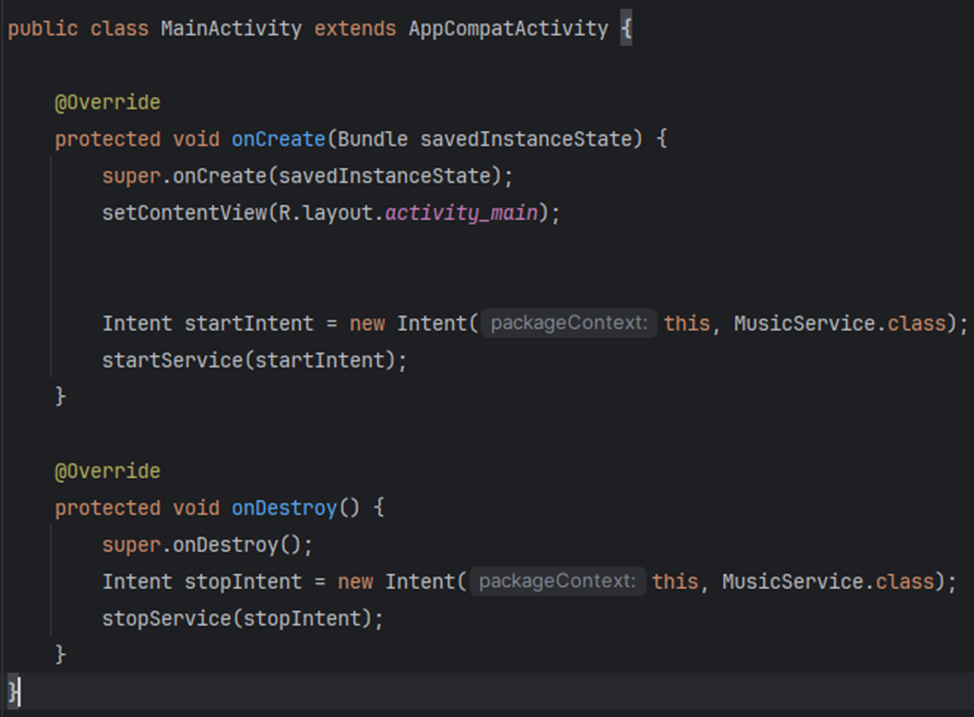


Рисунок – Запуск и остановка сервиса через класс MainActivity

Мы рассмотрели создание Unbound Service, который работает в фоновом режиме независимо от активности других компонентов приложения. Такой сервис идеально подходит для выполнения длительных операций, таких как воспроизведение музыки или загрузка файлов, когда не требуется постоянное взаимодействие с другими частями приложения. Unbound Service продолжает работать даже после того, как компонент, который его запустил, был уничтожен, пока не завершит свою задачу или пока его явно не остановят.

Теперь перейдём к Bound Service, который предоставляет более тесное взаимодействие между сервисом и другими компонентами приложения, такими как Activity или Fragment. Bound Service позволяет клиентам привязываться к нему, вызывать его методы напрямую и получать данные в реальном времени. Этот тип сервиса особенно полезен, когда требуется постоянный обмен информацией, например, для управления воспроизведением музыки с возможностью паузы, перемотки или изменения громкости. В отличие от Unbound Service, Bound Service автоматически уничтожается, когда все клиенты отвязываются, если только он не был также запущен через startService().

Для реализации Bound Service необходимо добавить в класс сервиса специальный механизм привязки. Внутри класса MusicService создаётся внутренний класс MusicBinder, который наследуется от стандартного класса Binder. Этот класс содержит всего один метод getService(), предназначенный для получения ссылки на текущий экземпляр сервиса. Такой подход является стандартным для реализации привязки в Android и позволяет компонентам приложения получить доступ к публичным методам сервиса.

Объект класса MusicBinder создаётся как поле сервиса и инициализируется при создании экземпляра MusicService. Это гарантирует, что все клиенты будут получать один и тот же объект для привязки. В методе onBind() сервиса возвращается этот заранее созданный объект binder, когда компоненты запрашивают привязку к сервису (Рисунок 8).

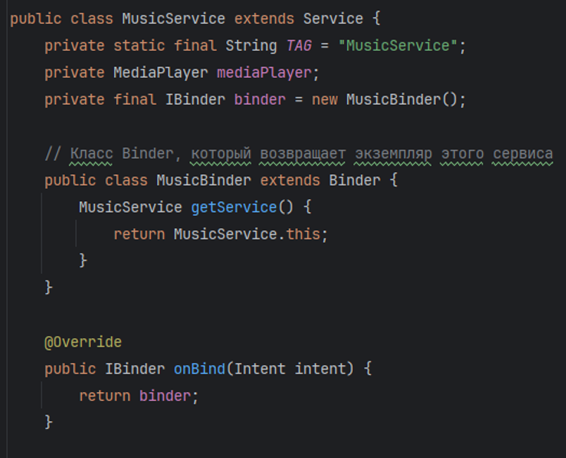


Рисунок – Реализация базовой структуры привязки

В сервис добавляются публичные методы, которые будут доступны для вызова из Activity после успешной привязки. Метод playMusic() проверяет состояние MediaPlayer и запускает воспроизведение музыки, если оно не было начато ранее. При этом в лог выводится сообщение о начале воспроизведения, что помогает отслеживать работу приложения во время отладки.

Метод pauseMusic() выполняет приостановку воспроизведения, если музыка в данный момент играет. Это позволяет временно остановить звук без полной остановки сервиса. Как и в случае с playMusic(), состояние изменений фиксируется в системном логе. Для полной остановки музыки предназначен метод stopMusic(), который не только останавливает воспроизведение, но и возвращает MediaPlayer в исходное состояние с помощью метода prepare(), что необходимо для последующих запусков.

Дополнительно реализован метод isPlaying(), который возвращает текущее состояние воспроизведения (Рисунок 9).

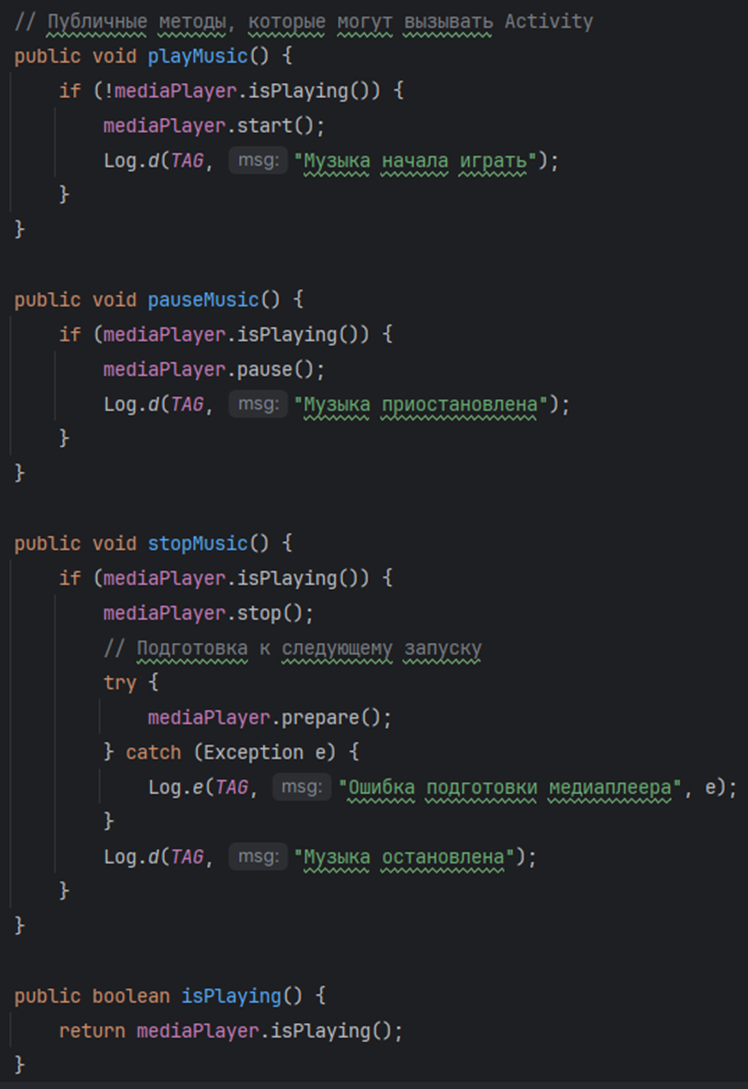


Рисунок – Методы управления сервисом

В файле разметки activity\_main.xml создадим вертикальный LinearLayout как корневой элемент интерфейса. Внутри него разместим четыре кнопки управления с уникальными идентификаторами: кнопку с идентификатором playButton для запуска воспроизведения, кнопку с идентификатором pauseButton для постановки на паузу, кнопку с идентификатором stopButton для полной остановки и кнопку с идентификатором statusButton для проверки состояния (Рисунок 10).



Рисунок – Разметка для MainActivity

В классе MainActivity создадим механизм привязки к сервису MusicService. Объявим поле musicService для хранения ссылки на сервис и флаг isBound, который будет отслеживать состояние подключения. Реализуем объект ServiceConnection, содержащий два основных метода для работы с привязкой.

В методе onServiceConnected получим экземпляр сервиса через переданный IBinder, преобразовав его к типу MusicService.MusicBinder. Установим флаг isBound в true и выведем уведомление о успешном подключении. Метод onServiceDisconnected сработает при неожиданном разрыве связи с сервисом, сбросит флаг isBound и проинформирует пользователя об отключении (Рисунок 11).

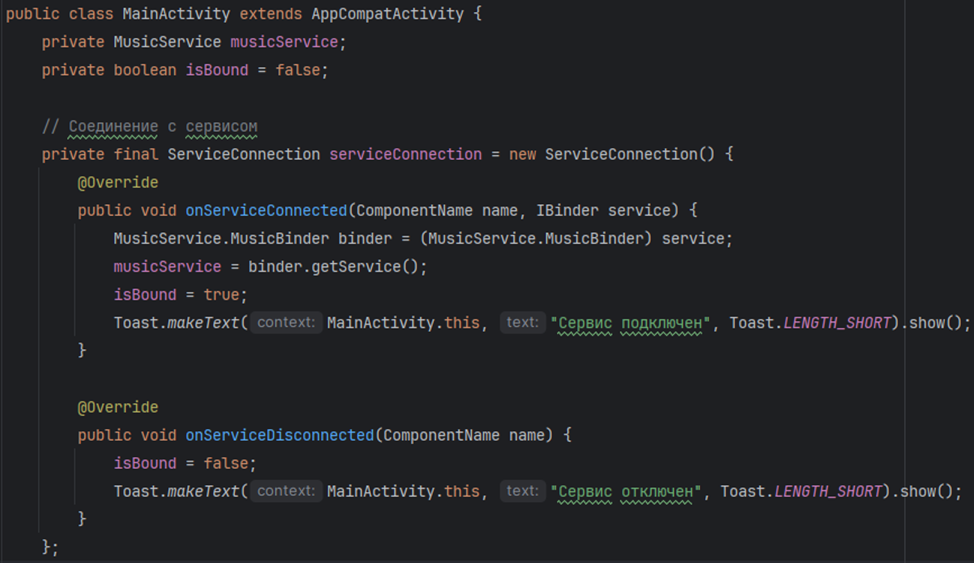


Рисунок – Механизм привязки к сервису в MainActivity

В методе onCreate() MainActivity реализуем привязку к сервису MusicService с помощью метода bindService(). Этот метод принимает три ключевых параметра.

Первым параметром передаем Intent, который явно указывает на класс сервиса MusicService.class, что позволяет системе точно идентифицировать, к какому сервису нужно подключиться.

Вторым параметром идет объект ServiceConnection (serviceConnection), который мы создали ранее. Этот объект содержит callback-методы onServiceConnected() и onServiceDisconnected(), которые система вызовет при успешном подключении к сервису или при неожиданном разрыве соединения соответственно.

Третий и последующие параметры — флаги, определяющие поведение привязки. Мы используем константу BIND\_AUTO\_CREATE, которая указывает системе автоматически создать сервис, если он еще не был запущен. Это стандартный вариант для большинства случаев. Другие возможные флаги включают BIND\_ABOVE\_CLIENT (приоритетное обслуживание), BIND\_IMPORTANT (повышенный приоритет) или BIND\_WAIVE\_PRIORITY (отказ от приоритета), однако на этом наборе не ограничиваются (Рисунок 12).

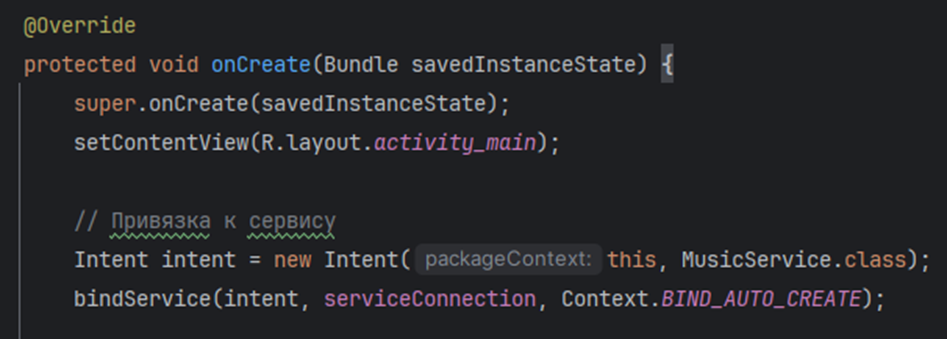


Рисунок – Привязка к сервису

В методе onCreate() MainActivity настраиваем обработку нажатий для кнопок управления музыкальным сервисом. Сначала получаем ссылки на все кнопки через findViewById(), используя их идентификаторы из разметки: playButton для воспроизведения, pauseButton для паузы, stopButton для остановки и statusButton для проверки состояния.

Устанавливаем для каждой кнопки соответствующий обработчик нажатий, который будет вызывать методы нашего сервиса. При нажатии на кнопку воспроизведения будет вызываться метод playMusic(), кнопка паузы активирует метод pauseMusic(), а кнопка остановки запустит метод stopMusic(). Отдельная кнопка статуса использует метод isPlaying() для проверки текущего состояния воспроизведения, на основе которого выводит всплывающее уведомление (Toast). Важно отметить, что все эти действия выполняются только при успешном подключении к сервису, что контролируется проверкой флага isBound перед каждым вызовом методов сервиса (Рисунок 13).

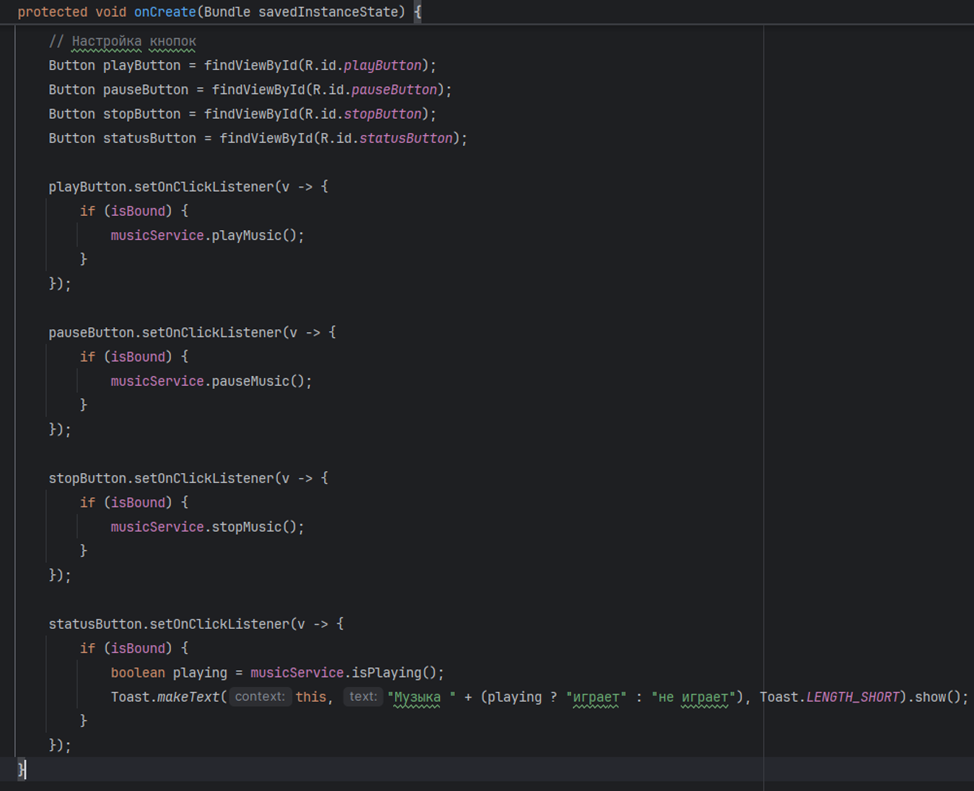


Рисунок – Добавление обработчиков нажатия для кнопок

В методе onDestroy() реализуется отвязка от сервиса при уничтожении Activity. Проверяется флаг isBound, и если подключение активно, вызывается метод unbindService(), куда передается ранее созданный объект serviceConnection. После успешной отвязки флаг isBound сбрасывается в false, что фиксирует разрыв соединения с сервисом.

Этот механизм обеспечивает корректное завершение работы с сервисом и освобождение ресурсов. Важно отметить, что отвязка выполняется только при активном подключении, что предотвращает попытки разрыва несуществующего соединения. Вызов super.onDestroy() гарантирует правильное выполнение стандартных процедур уничтожения Activity (Рисунок 14).

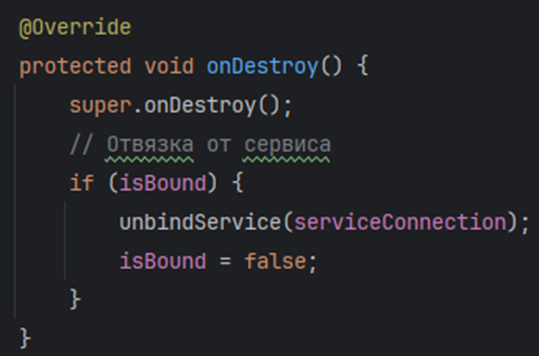


Рисунок – Отвязка от сервиса

При переходе на Bound Service реализуется двустороннее взаимодействие между Activity и сервисом. Это позволяет напрямую управлять воспроизведением музыки через вызов методов сервиса. Интерфейс приложения теперь содержит четыре основные функции: запуск воспроизведения, постановку на паузу, полную остановку и проверку текущего статуса проигрывателя.

Сообщение "Сервис подключен" подтверждает успешное установление связи между Activity и MusicService. Такая архитектура обеспечивает мгновенную реакцию на пользовательские действия, так как все команды выполняются напрямую через привязанный сервис. В отличие от Unbound Service, где взаимодействие было односторонним, Bound Service позволяет не только отправлять команды, но и получать текущее состояние проигрывателя в реальном времени (Рисунок 15).

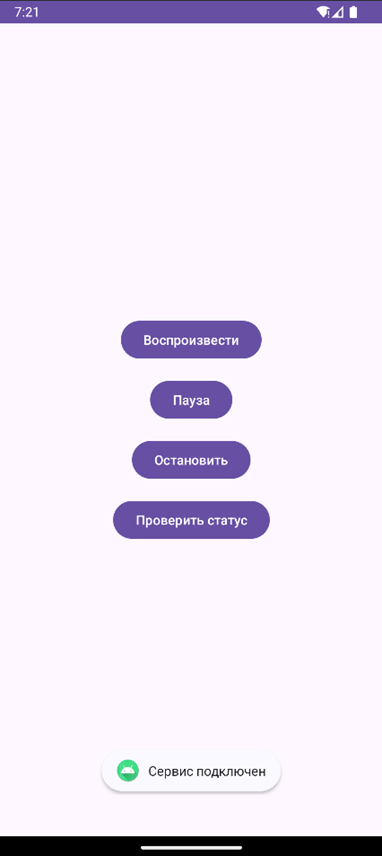


Рисунок – Демонстрация приложения, подключенного к сервису

## Диалоговые окна

Диалоговые окна в Android представляют собой небольшие окна, которые отображаются поверх основного содержимого приложения для передачи важного сообщения пользователю или запроса действия от пользователя. Эти окна могут использоваться для подтверждения действий, информирования о событиях, ввода данных и других задач, требующих внимания пользователя.

Среди основных типов диалоговых окон можно выделить:

* AlertDialog: используется для отображения предупреждений и предложения пользователю совершить выбор. Может включать кнопки для обработки действий пользователя, такие как «Да», «Нет» или «Отмена»,
* DatePickerDialog и TimePickerDialog: позволяют пользователю выбрать дату или время соответственно,
* Custom Dialog: позволяет создавать диалоговые окна с пользовательским макетом, что может включать любые элементы управления.

Разберем созданием каждого типа диалоговых окон.

Приведём пример создания AlertDialog. AlertDialog.Builder представляет собой мощный инструмент для создания диалоговых окон в Android-приложениях. Рассмотрим процесс построения диалога с использованием этого класса.

Инициализацию Builder'а выполняем, передавая контекст текущей Activity. Затем последовательно настраиваем основные компоненты диалога. Устанавливаем заголовок, который будет отображаться в верхней части окна, используя метод setTitle(). Добавляем текстовое сообщение через setMessage(), размещаемое под заголовком. Для привлечения внимания пользователя можно задать соответствующую иконку методом setIcon(), выбрав из стандартных ресурсов или собственных изображений (Рисунок 16).

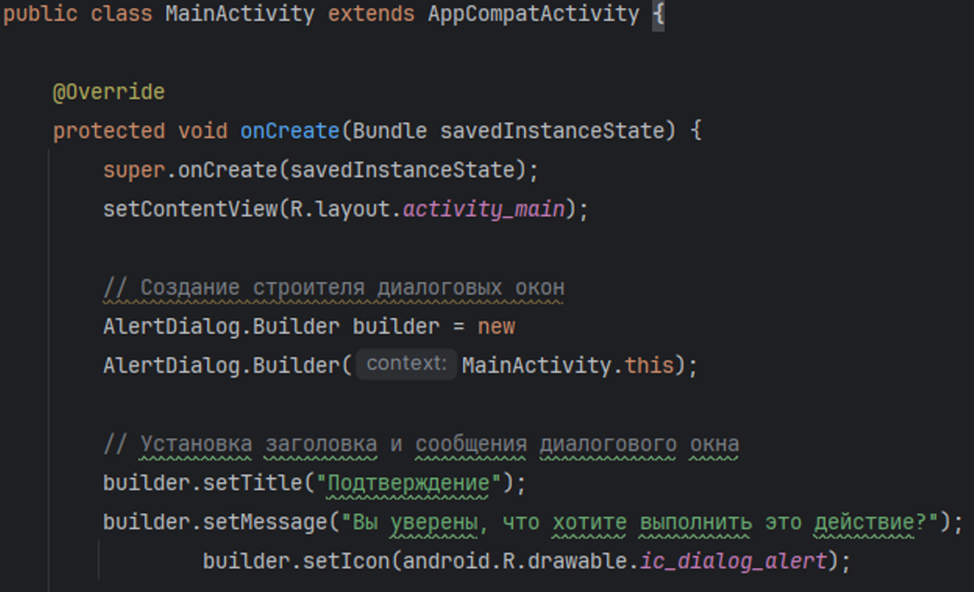


Рисунок – Создание объекта класса AlertDialog.Builder

При настройке кнопок диалогового окна используем метод setPositiveButton() для создания кнопки подтверждения, куда передаём текст кнопки и объект DialogInterface.OnClickListener с реализацией нужной логики в методе onClick(). Для кнопки отмены применяем метод setNegativeButton() с аналогичными параметрами, где в обработчике onClick() обычно вызываем метод dismiss() у переданного объекта DialogInterface, чтобы закрыть диалог. Дополнительную кнопку можно добавить через метод setNeutralButton(), если требуется предоставить пользователю альтернативный вариант действия. Каждый обработчик получает параметры dialog (ссылка на само диалоговое окно) и which (идентификатор нажатой кнопки), что позволяет гибко управлять поведением диалога в зависимости от действий пользователя (Рисунок 17).

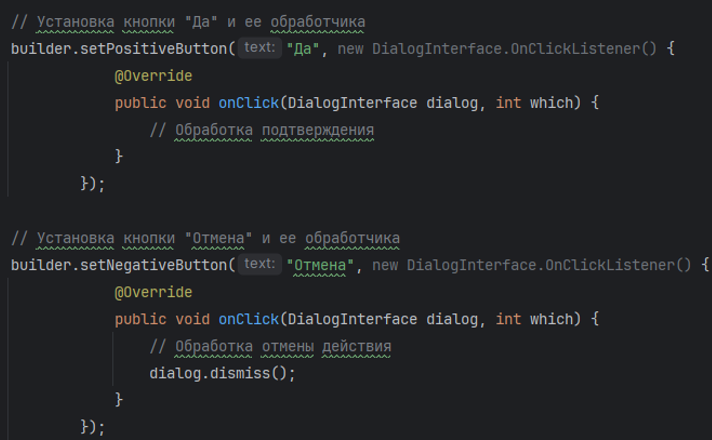


Рисунок – Добавление кнопок в диалоговое окно

При обработке нажатий в AlertDialog параметр which может содержать одно из следующих стандартных значений:

* DialogInterface.BUTTON\_POSITIVE (значение -1) для положительной кнопки (подтверждение),
* DialogInterface.BUTTON\_NEGATIVE (значение -2) для отрицательной кнопки (отмена),
* DialogInterface.BUTTON\_NEUTRAL (значение -3) для нейтральной кнопки.

На практике проверка значения which обычно не требуется, так как каждый тип кнопки имеет свой отдельный обработчик. Однако этот параметр может быть полезен, если все кнопки обрабатываются в едином обработчике или при создании кастомных диалогов с дополнительными кнопками. В таких случаях через сравнение значения which с константами можно точно определить, какая именно кнопка была нажата.

После завершения настройки создадим экземпляр AlertDialog методом create() и отобразим его пользователю вызовом show(). Такой подход позволяет гибко конфигурировать диалоговое окно, добавляя различные элементы управления по мере необходимости (Рисунок 18).

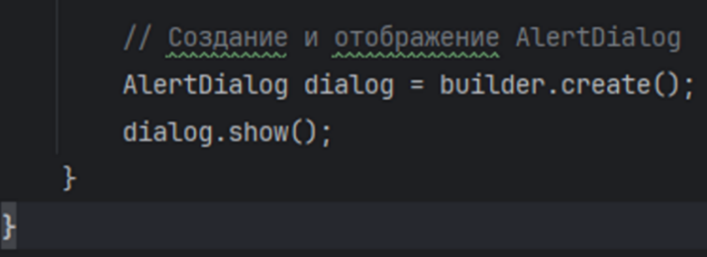


Рисунок – Создание и отображение AlertDialog

Для расширения функциональности AlertDialog.Builder предоставляет дополнительные методы конфигурации. Метод setView() позволяет добавить кастомную разметку в диалоговое окно, что дает возможность создавать сложные формы с различными элементами управления. Для работы с элементами выбора доступны методы setSingleChoiceItems() для радиокнопок, setMultiChoiceItems() для чекбоксов и setItems() для простых списков.

Метод setCancelable() определяет, можно ли закрыть диалог нажатием вне его области или кнопкой "Назад".

Посмотрим на созданное нами диалоговое окно типа AlertDialog (Рисунок 19).

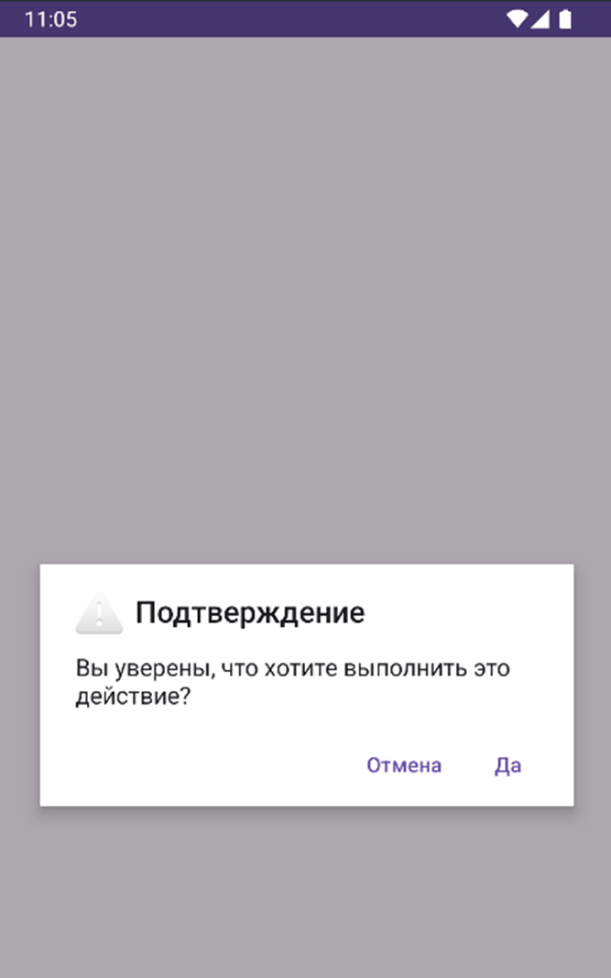


Рисунок – Демонстрация AlertDialog

Класс AlertDialog.Builder предоставляет набор методов для настройки диалогового окна.

Основные методы конфигурации:

* setTitle() – задает текстовый заголовок,
* setCustomTitle() – устанавливает кастомное View для заголовка,
* setMessage() – добавляет текстовое сообщение,
* setCustomMessage() – позволяет использовать свое View для сообщения,
* setIcon() – добавляет иконку в заголовок,
* setCancelable() – определяет возможность отмены (по умолчанию true).

Методы для настройки кнопок:

* setPositiveButton() – добавляет кнопку подтверждения действия,
* setNegativeButton() – добавляет кнопку отмены действия,
* setNeutralButton() – добавляет дополнительную кнопку с альтернативным действием,
* setOnCancelListener() – устанавливает обработчик события отмены диалога,
* setOnDismissListener() – устанавливает обработчик события закрытия диалога.

Методы для отображения списков:

* setItems() – добавляет простой список элементов для выбора, где каждый элемент можно выбрать однократно. При выборе элемента диалог автоматически закрывается,
* setSingleChoiceItems() – реализует список с возможностью выбора только одного элемента (аналог RadioButton). Отображает маркер выбранного элемента и позволяет изменить выбор перед подтверждением,
* setMultiChoiceItems() – настраивает список с возможностью множественного выбора (аналог CheckBox). Позволяет выбрать несколько элементов одновременно, сохраняя состояние каждого,
* setCursor() – предназначен для работы с данными из базы данных через объект Cursor. Он позволяет отображать в диалоговом окне элементы, полученные из SQL-запроса, автоматически используя указанную текстовую колонку для вывода содержимого,
* setAdapter() – подключает кастомный адаптер для отображения сложных списков с индивидуальной разметкой элементов. Дает полный контроль над данными и их визуальным представлением.

Завершающие методы:

* сreate() – создает объект AlertDialog,
* show() – создает и сразу показывает диалог.

Для создания TimePickerDialog используется конструктор класса, а не паттерн Builder, в отличие от AlertDialog. Конструктор TimePickerDialog принимает следующие параметры: контекст приложения, обработчик выбора времени OnTimeSetListener, начальные значения часов и минут, а также флаг использования 24-часового формата.

Обработчик OnTimeSetListener содержит метод onTimeSet() с тремя параметрами: view (ссылка на сам TimePicker), selectedHour (выбранный час) и selectedMinute (выбранные минуты). Этот метод вызывается при подтверждении выбора времени пользователем.

При реализации TimePickerDialog в классе MainActivity сначала объявим поля hour и minute типа int для хранения времени. Установим значения по умолчанию — 12 часов и 0 минут. Эти переменные будут использоваться для начальной настройки диалога и сохранения выбранных значений.

Для создания TimePickerDialog вызовем его конструктор, передав следующие параметры:

* контекст текущей Activity (this),
* объект OnTimeSetListener с переопределенным методом onTimeSet(),
* начальные значения часов и минут (хранящиеся в полях hour и minute),
* флаг is24HourView со значением true для 24-часового формата.

В методе onTimeSet() обработчика реализуем логику обновления интерфейса: сохраняем выбранные пользователем значения часов (selectedHour) и минут (selectedMinute) в соответствующие поля класса, затем форматируем их в строку вида "ЧЧ:ММ" с добавлением ведущих нулей для однозначных значений, после чего обновляем текст в TextView, отображая пользователю итоговый результат выбора. Это позволяет наглядно демонстрировать установленное время и сохранять последние выбранные значения для дальнейшего использования в приложении (Рисунок 20).



Рисунок – Создание экземпляра TimePickerDialog

Вызовем метод show() диалогового окна, чтобы сразу вывести его на экран (Рисунок 21).

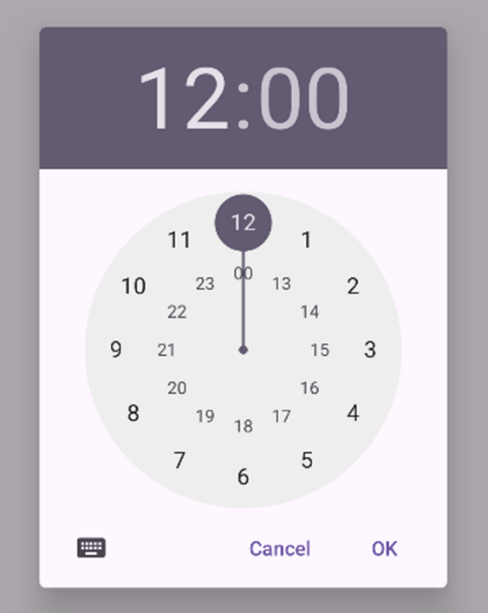


Рисунок – Демонстрация отображения TimePickerDialog

Для создания DatePickerDialog, аналогично TimePickerDialog, используем конструктор класса, а не паттерн Builder. В классе MainActivity объявим три поля для хранения даты: year (год), month (месяц, где 0 соответствует январю) и day (день месяца), установив им значения по умолчанию.

Вызовем конструктор класса DatePickerDialog, передав следующие аргументы:

* контекст текущей Activity (this),
* объект OnDateSetListener с переопределенным методом onDateSet(),
* начальные значения года, месяца и дня (хранящиеся в полях year, month и day).

В обработчике onDateSet() реализуем логику обработки выбранной пользователем даты. При получении новых значений года (selectedYear), месяца (selectedMonth) и дня (selectedDay) при выборе их пользователем с помощью диалогового окна обновляем соответствующие поля класса, сохраняя выбранные параметры для дальнейшего использования в приложении. Особое внимание уделяем работе с месяцами — поскольку DatePicker использует нумерацию месяцев от 0 (январь) до 11 (декабрь), при отображении даты пользователю необходимо увеличивать значение месяца на 1 для приведения к привычному формату. После создания экземпляра DatePickerDialog, вызовем его метод show() для вывода диалогового окна на экран (Рисунок 22).

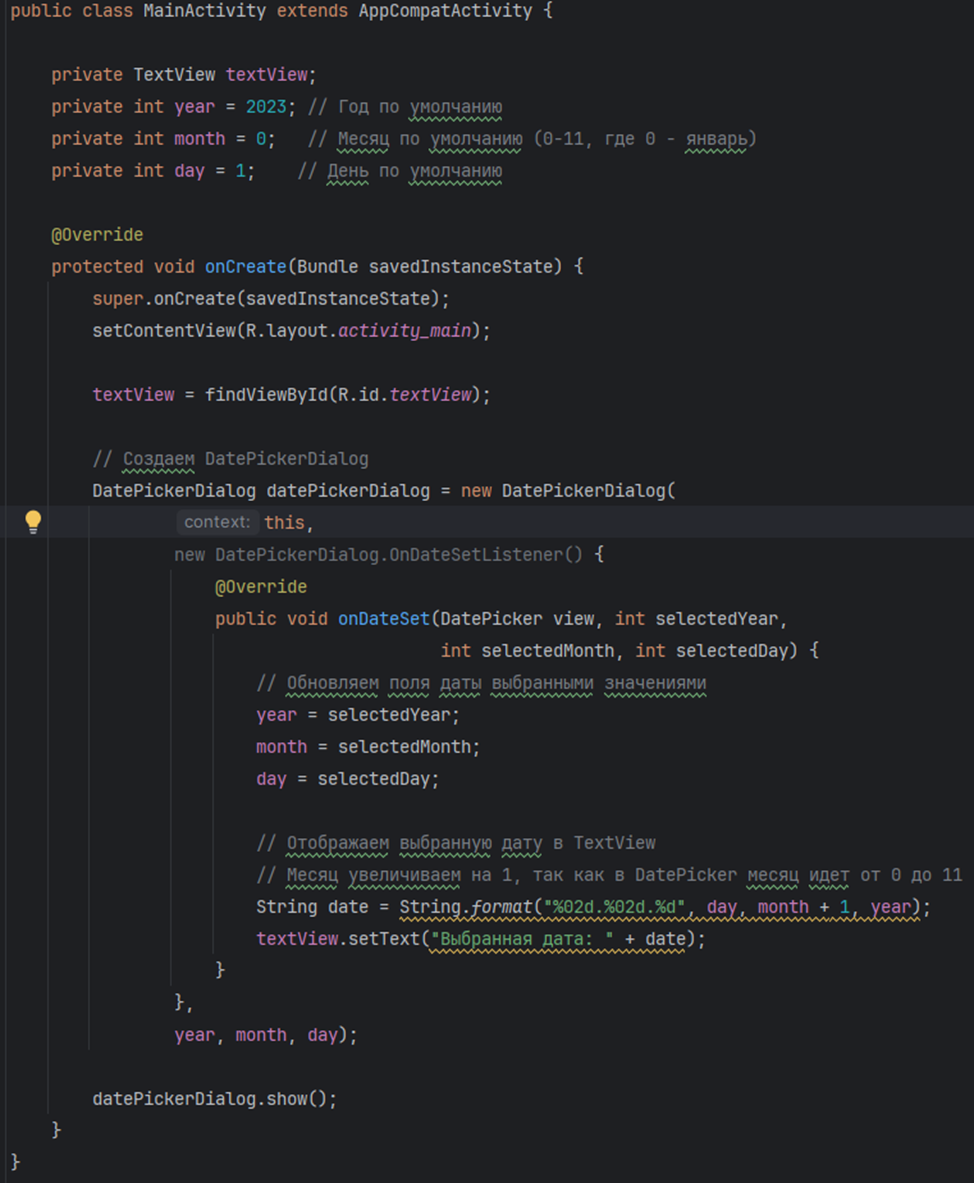


Рисунок – Создание экземпляра DatePickerDialog

Отображение диалогового окна выбора времени представлено на рисунке Рисунок 23.

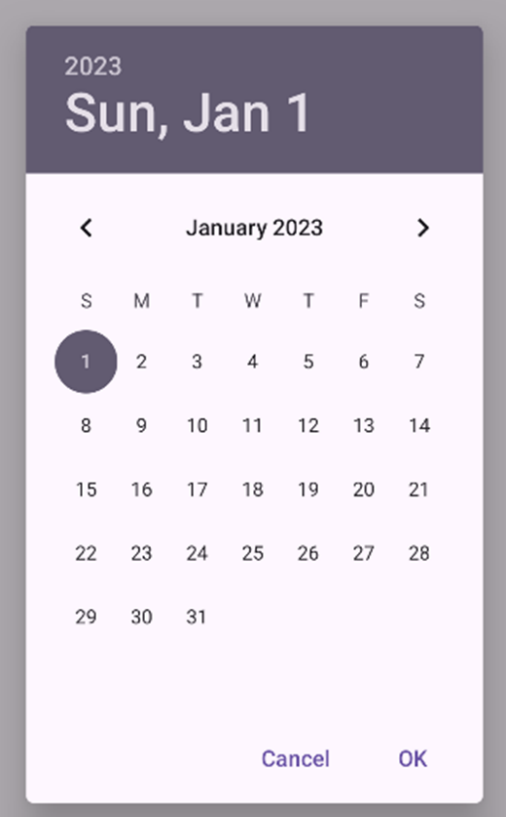


Рисунок – Демонстрация отображения DatePickerDialog

Создание пользовательского диалогового окна требует особого подхода к проектированию его интерфейса. Основное отличие от стандартных диалогов заключается в необходимости самостоятельно разрабатывать макет, который будет определять внешний вид и состав элементов управления. Для реализации такого решения сначала создается отдельный XML-файл разметки, например custom\_dialog.xml, где размещаются все необходимые компоненты пользовательского интерфейса (Рисунок 24).

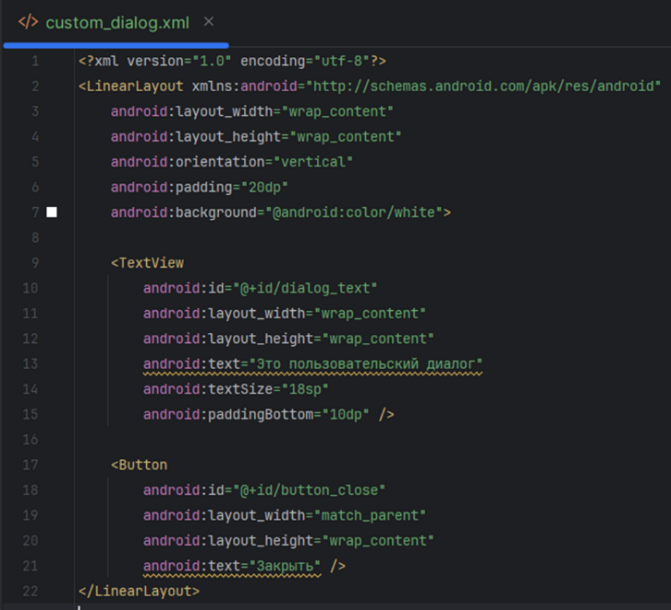


Рисунок – Файл разметки пользовательского диалогового окна

В классе MainActivity реализуем отображение пользовательского диалогового окна с использованием созданного макета. Создаем экземпляр класса Dialog, передавая текущий контекст в конструктор, и устанавливаем кастомную разметку через метод setContentView(), указав идентификатор файла R.layout.custom\_dialog. Для обработки нажатия кнопки закрытия находим соответствующий элемент интерфейса через findViewById() и устанавливаем обработчик события, который вызывает метод dismiss() для закрытия диалога. Завершаем настройку вызовом show(), который отображает подготовленное окно пользователю (Рисунки Рисунок 25-Рисунок 26).

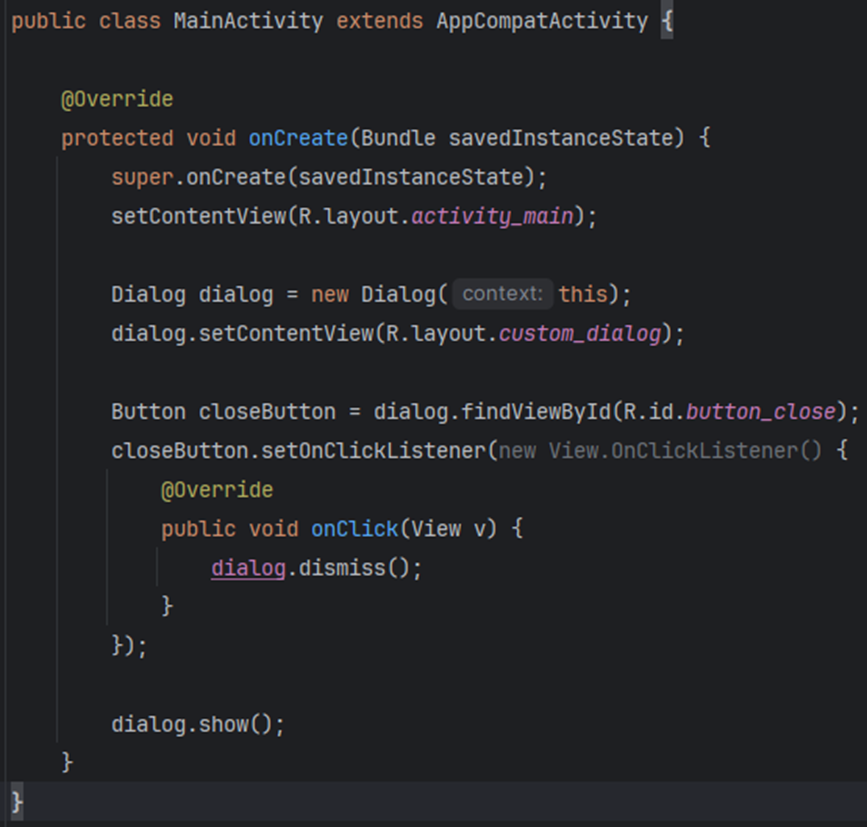


Рисунок – Создание пользовательского диалогового окна

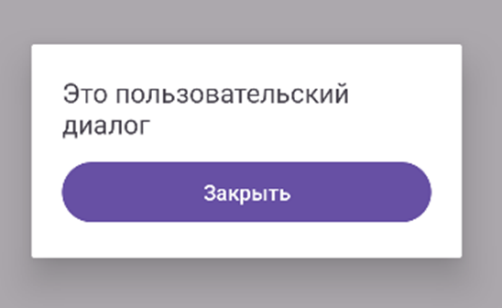


Рисунок – Отображение пользовательского диалогового окна

Использование DialogFragment для диалоговых окон предоставляет значительные преимущества по сравнению с созданием экземпляров класса Dialog в Activity. Данный подход лучше обрабатывает поворот устройства, нажатие системной кнопки "Назад" и адаптируется под различные размеры экранов благодаря встроенной архитектуре фрагментов.

Для реализации диалогового окна необходимо создать класс, унаследованный от DialogFragment. В проекте следует добавить новый класс фрагмента, например MyDialogFragment, который будет содержать логику работы диалогового окна.

Класс фрагмента для диалогового окна сочетает стандартную функциональность фрагмента с его полным жизненным циклом и дополнительные возможности, предоставляемые базовым классом DialogFragment. Для создания такого диалога разработчику доступны два основных подхода:

* первый подход предполагает переопределение метода onCreateDialog(), который должен возвращать готовый объект Dialog. Этот вариант позволяет использовать стандартные диалоги Android, такие как AlertDialog или DatePickerDialog, с сохранением всех преимуществ DialogFragment,
* второй подход заключается в использовании традиционного для фрагментов метода onCreateView(), где определяется кастомная разметка диалогового окна. Такой способ дает полный контроль над интерфейсом и поведением диалога, позволяя реализовать любые дизайнерские решения.

Приведём пример создания диалогового окна согласно первому подходу.

В методе onCreateDialog() класса MyDialogFragment создается и настраивается стандартное диалоговое окно типа AlertDialog. С помощью объекта AlertDialog.Builder устанавливаются основные параметры: заголовок "Подтверждение" через метод setTitle(), текст сообщения "Вы уверены..." с помощью setMessage(), иконка предупреждения из системных ресурсов через setIcon().

Добавляются две кнопки взаимодействия: "Отмена" как отрицательная кнопка (setNegativeButton()) и "Ок" как положительная (setPositiveButton()). Обе кнопки пока не содержат обработчиков событий (параметр listener установлен в null). Завершающим этапом метод create() преобразует построитель в готовый объект Dialog, который возвращается как результат работы метода (Рисунок 27).

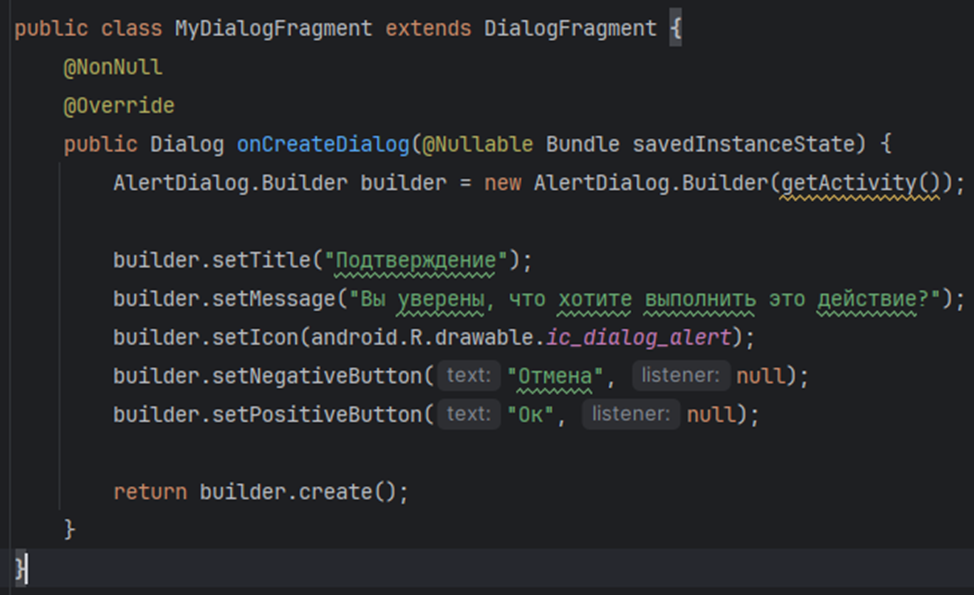


Рисунок – Расширение класса DialogFragment

В методе onCreate() класса MainActivity создаётся объект класса DialogFragment, а затем для отображения диалога используется вызов метода show(), в который передаются следующие аргументы:

* результат вызова getSupportFragmentManager(), который возвращает системный менеджер фрагментов, ответственный за их жизненный цикл внутри Activity,
* строковый тег "CustomDialog", выполняющий роль уникального идентификатора для последующего поиска диалогового окна и управления им.

Создание и отображение диалогового окна в MainActivity представлено на рисунке Рисунок 28.



Рисунок – Создание и отображение экземпляра MyDialogFragment

Отображение диалогового окна, созданного с помощью расширения класса DialogFragment, представлено на рисунке Рисунок 29.

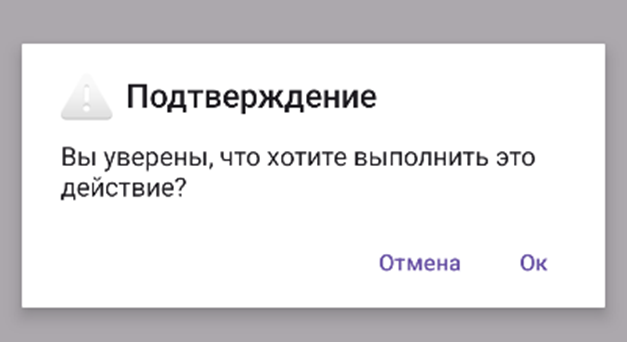


Рисунок – Отображение диалогового окна, созданного с помощью расширения класса DialogFragment

Для передачи данных в диалоговое окно с целью уточнения информации, можно использовать значения, введенные пользователем в элементы интерфейса Activity. Например, при нажатии на кнопку с идентификатором buttonAlert происходит получение текста из поля ввода textName, который затем подставляется в сообщение диалогового окна, созданного с помощью AlertDialog.Builder (Рисунки Рисунок 30-Рисунок 31).

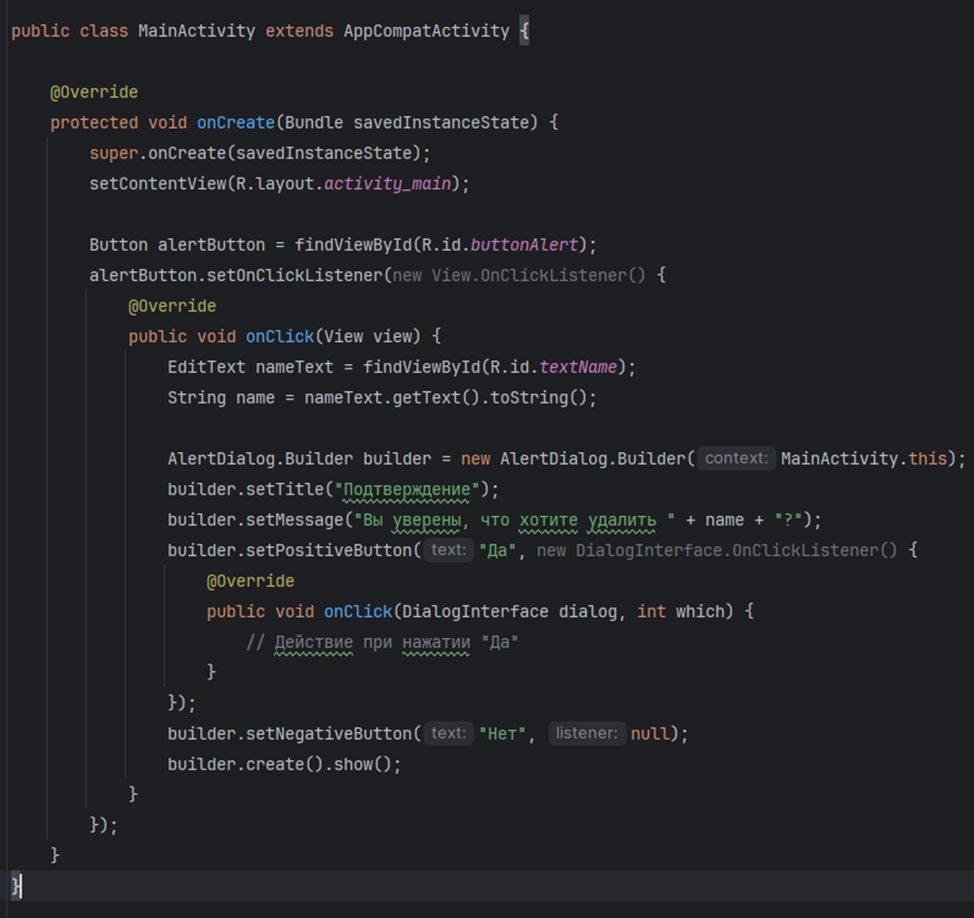


Рисунок – Пример передачи данных из поля ввода в диалоговое окно

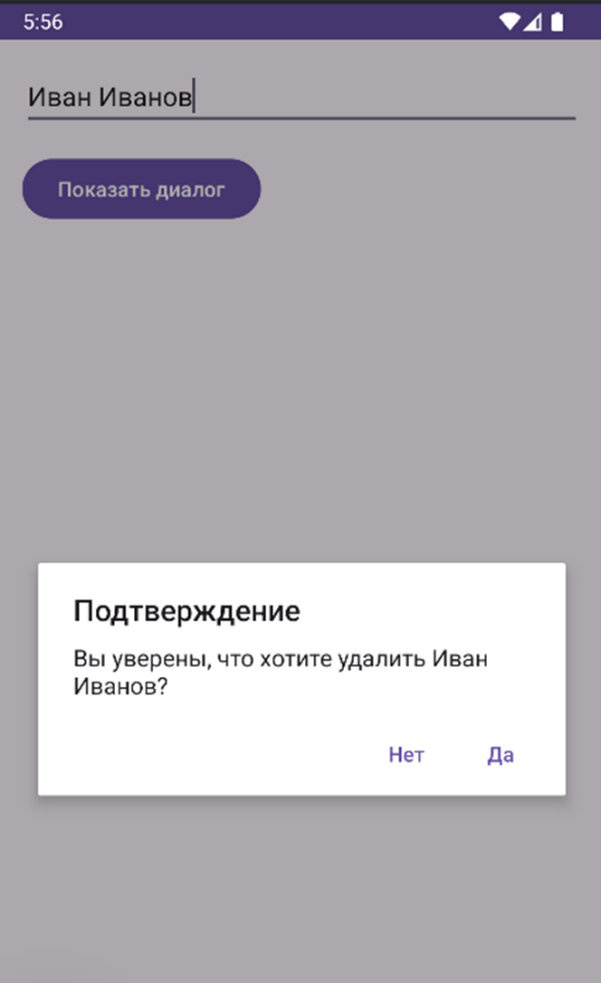


Рисунок – Интерфейс подтверждения в диалоговом окне с подстановкой введенных данных

# ПРАКТИЧЕСКОЕ ЗАДАНИЕ

## Разметка

Создадим проект Pract7 (Рисунок 32).

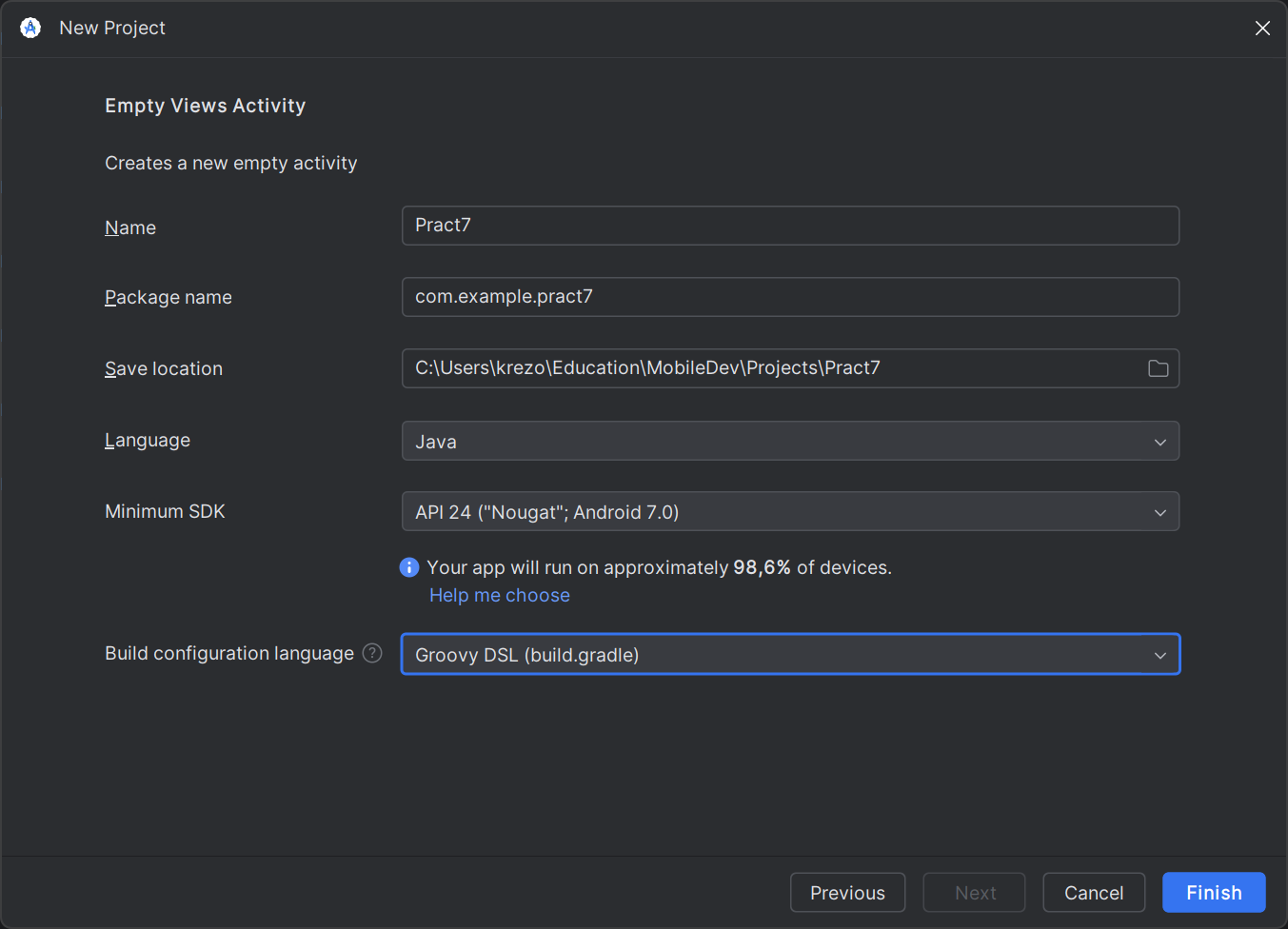


Рисунок – Создание проекта

Заполним созданный файл разметки activity\_main.xml кнопками и текстовыми полями для тестирования сервиса и диалоговых окон (Рисунки Рисунок 33-Рисунок 34).

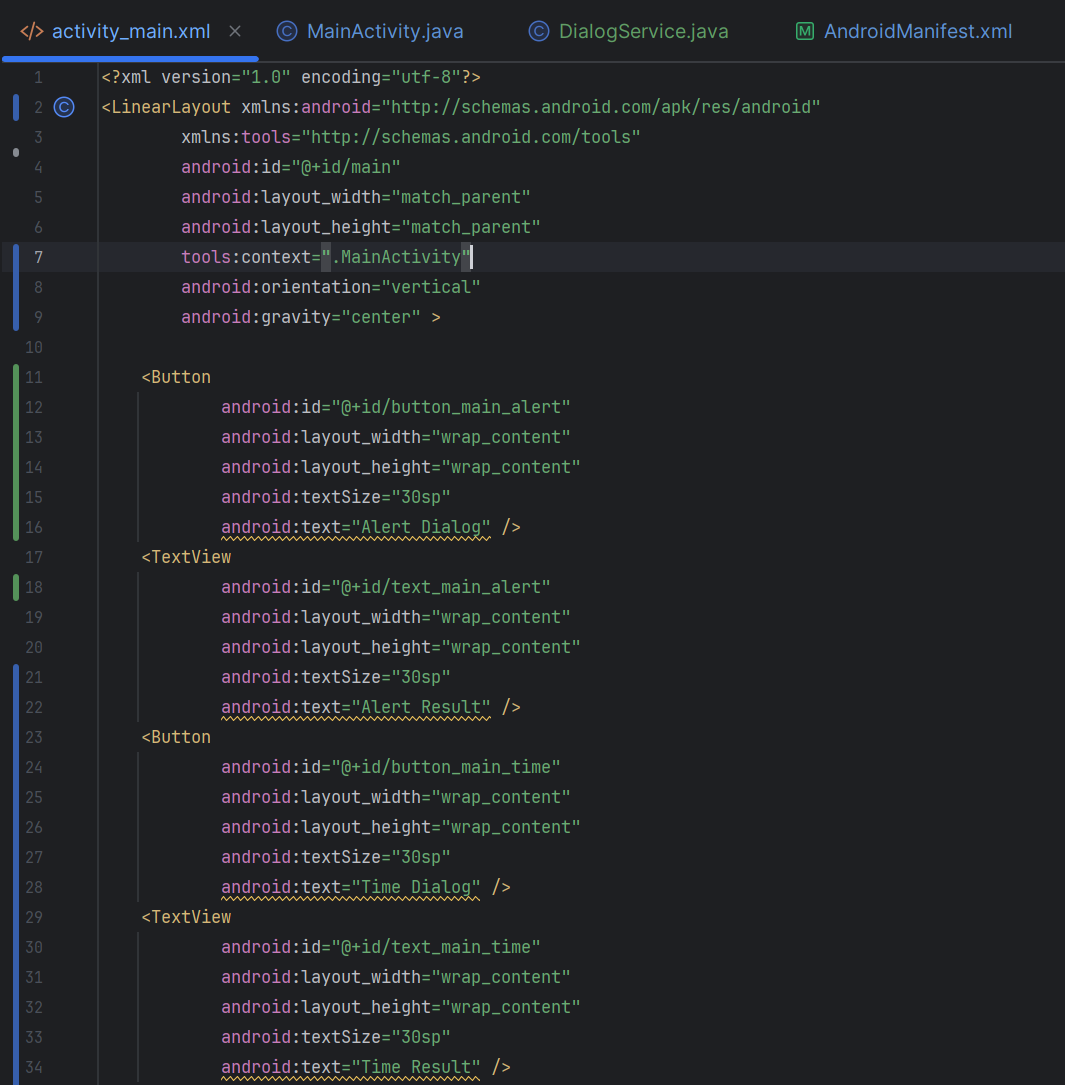


Рисунок – Файл разметки activity\_main.xml, часть 1



Рисунок – Файл разметки activity\_main.xml, часть 2

Также создадим файл разметки пользовательского диалога dialog\_custom.xml (Рисунок 35).

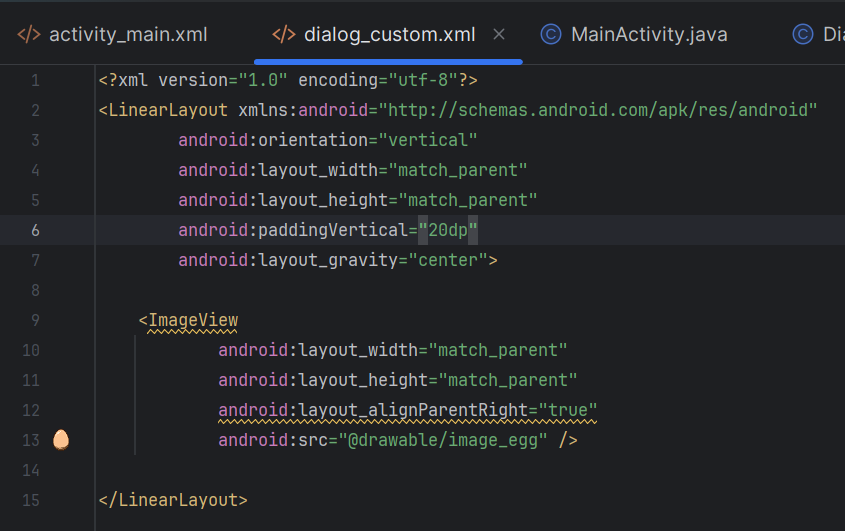


Рисунок – Файл разметки dialog\_custom.xml

## Реализация

Для начала создадим класс пользовательского диалога CustomDialog, наследуя его от DialogFragment. Реализуем в нем конструктор, принимающий слушали нажатий на варианты ответа диалога, а также переопределим метод onCreateDialog() родительского класса (Рисунок 36).

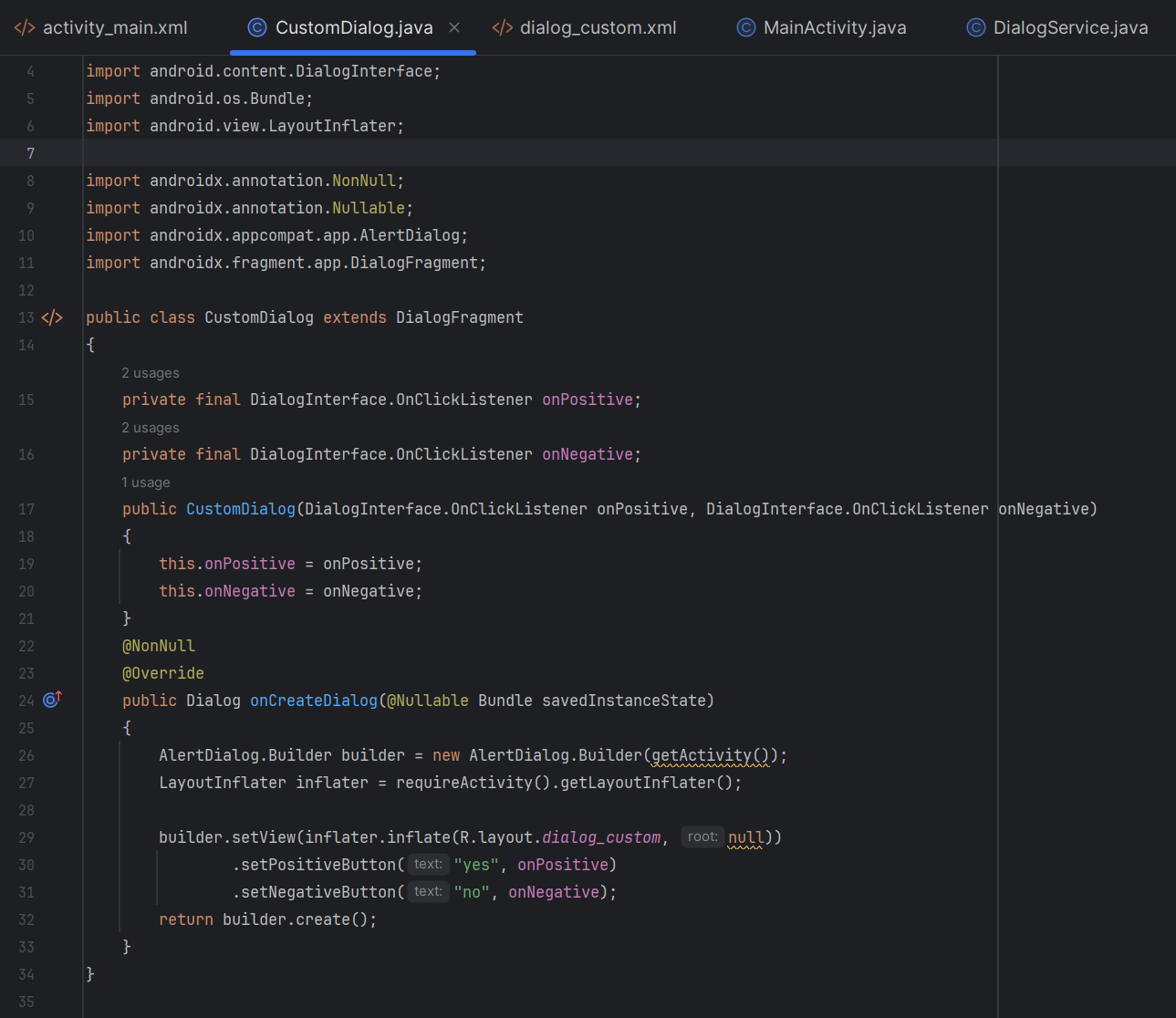


Рисунок – Описание класса CustomDialog

В методе onCreateDialog() создается класс AlertDialog.Builder, с помощью которого наполняется и возвращается класс диалога на основе класса AlertDialog.

Далее создадим сервис DialogService, с помощью которого и будем отображать диалоговые окна.

Сервис необходимо будет активно использовать, поэтому реализуем в нем метод onBind(), а также класс DialogBinder (Рисунок 37).

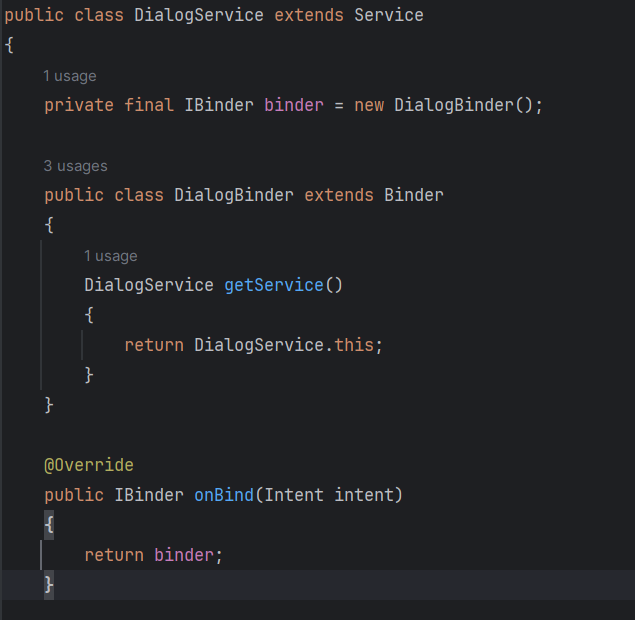


Рисунок – Реализация метода onBind() класса DialogService

Затем опишем методы для создания каждого из видов диалоговых окон:

* showAlert() для создания AlertDialog;
* showTimePicker() для создания TimePickerDialog;
* showDatePicker() для создания DatePickerDialog;
* showCustom() для создания CustomDialog.

Каждый из этих методов принимает соответствующие методы-обработчики нажатия на варианты ответа в диалоговом окне (Рисунок 38).

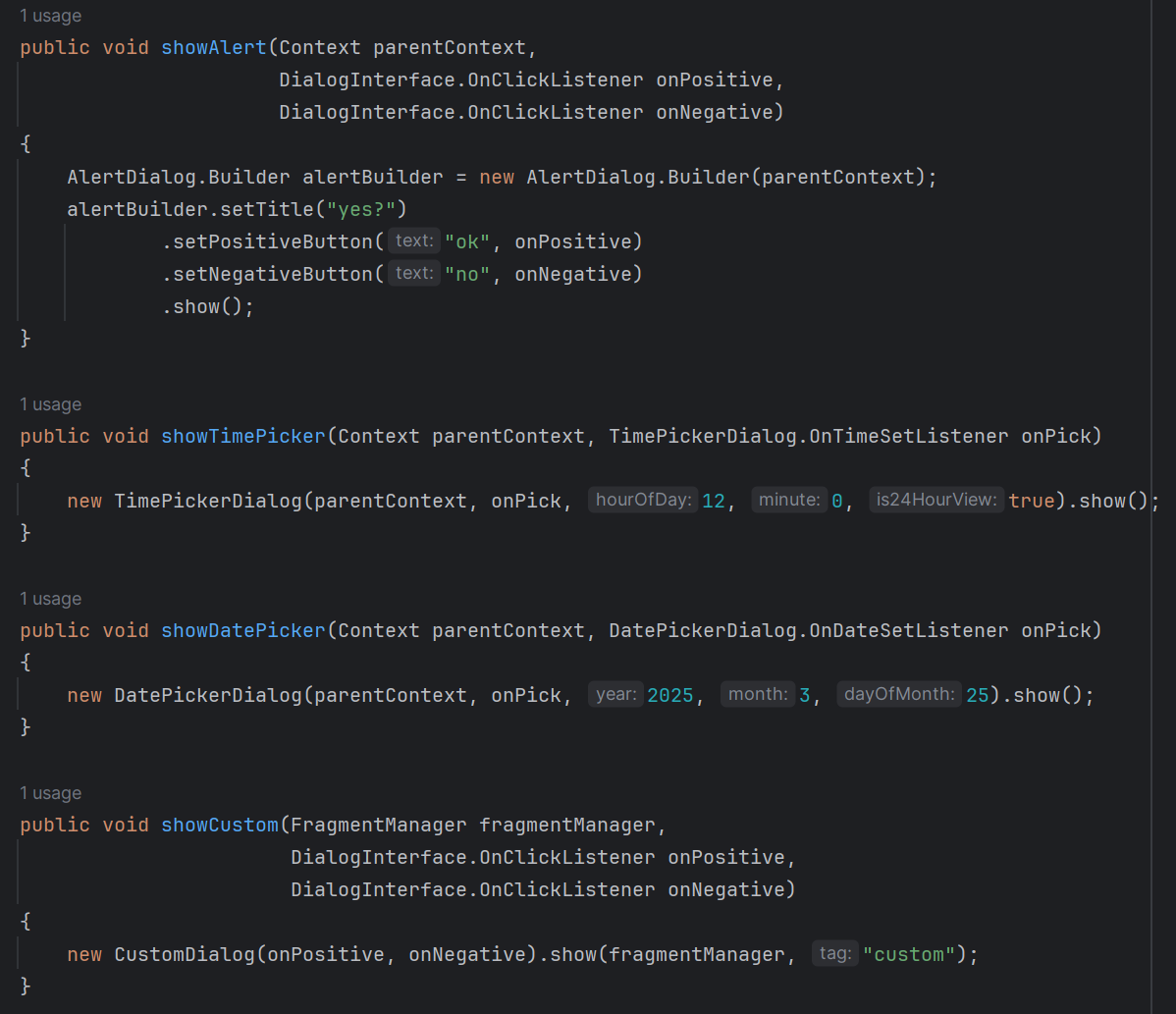


Рисунок – Методы класса DialogService

Чтобы использовать данный сервис в классе MainActivity необходимо воспользоваться Binding механизмом, для чего необходимо реализовать класс ServiceConnection (Рисунок 39).

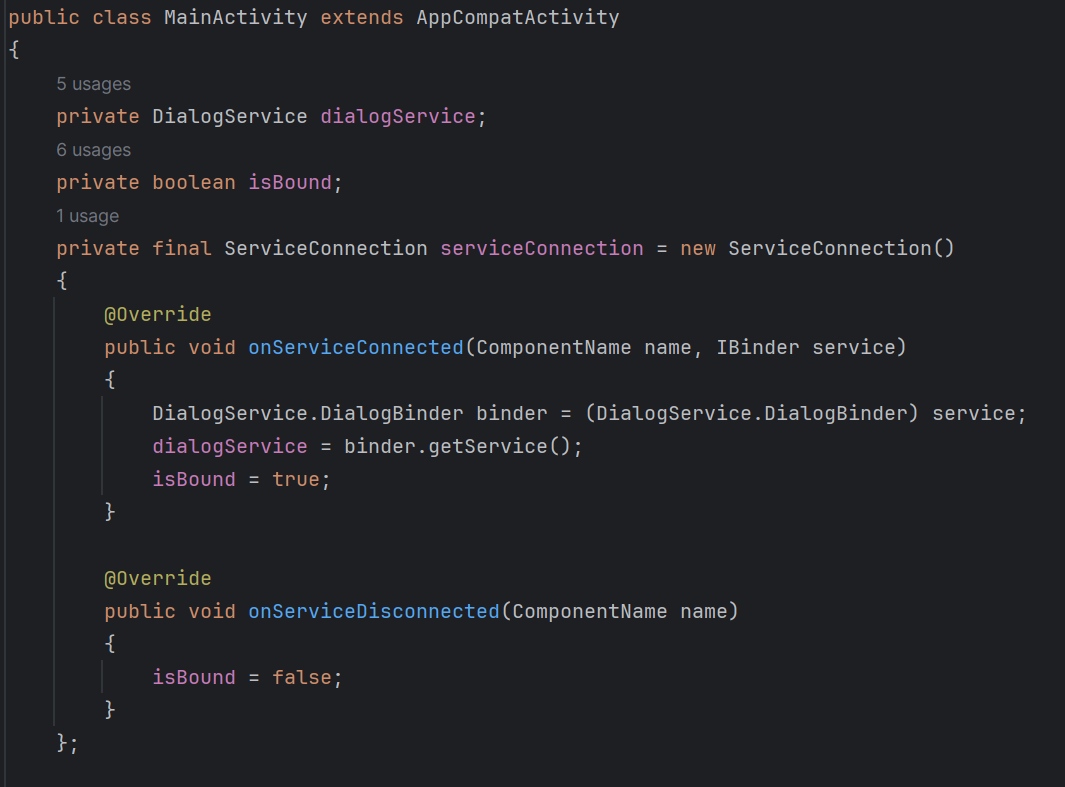


Рисунок – Описание анонимного класса ServiceConnection

На рисунке Рисунок 40 показано получение сервиса DialogService с помощью метода bindService().

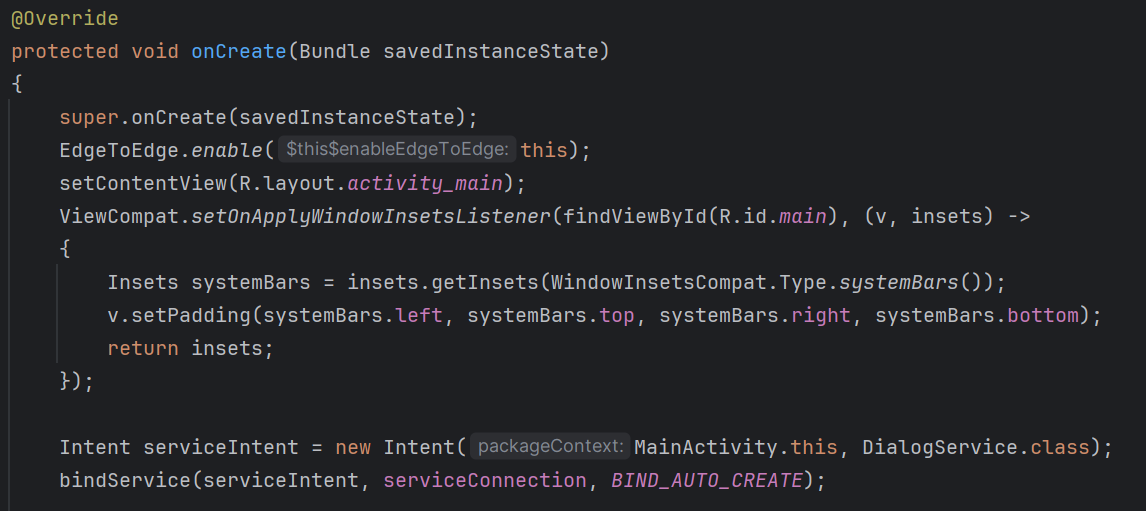


Рисунок – Получение сервиса DialogService

На рисунках Рисунок 41-Рисунок 42 показан запуск диалоговых окон с помощью сервиса с получением информации из окон и применении ее к текстовым полям.

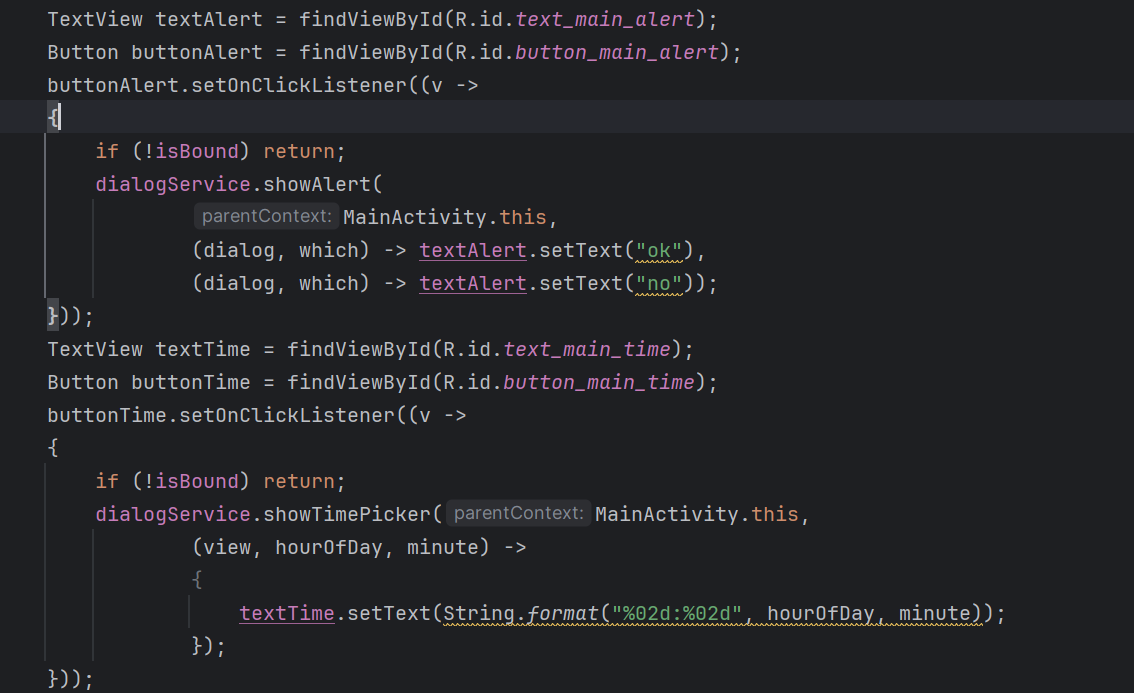


Рисунок – Описание метода onCreate() класса MainActivity, часть 1



Рисунок – Описание метода onCreate() класса MainActivity, часть 2

## Тестирование

Откроем приложение (Рисунок 43).

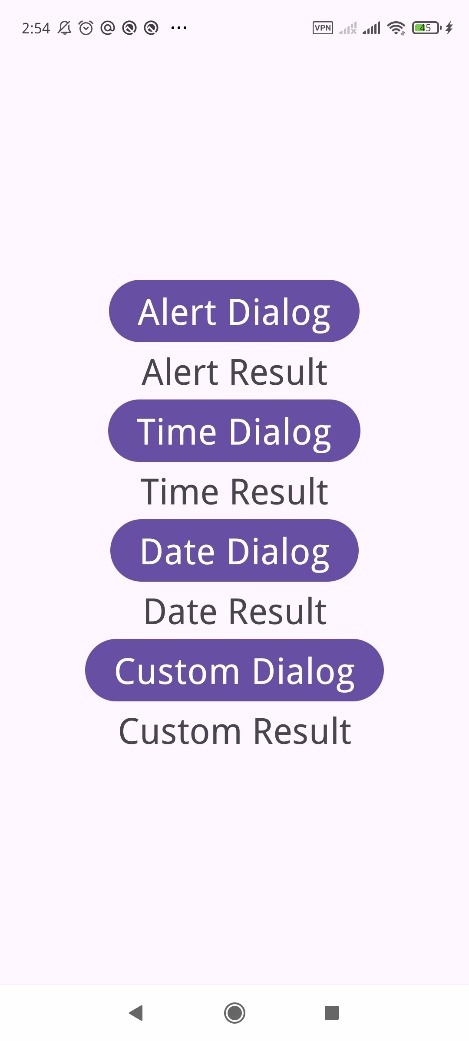


Рисунок – Начальный экран приложения

После нажатия на кнопку “Alert Dialog” открылось диалоговое окно с двумя вариантами ответа (Рисунок 44).

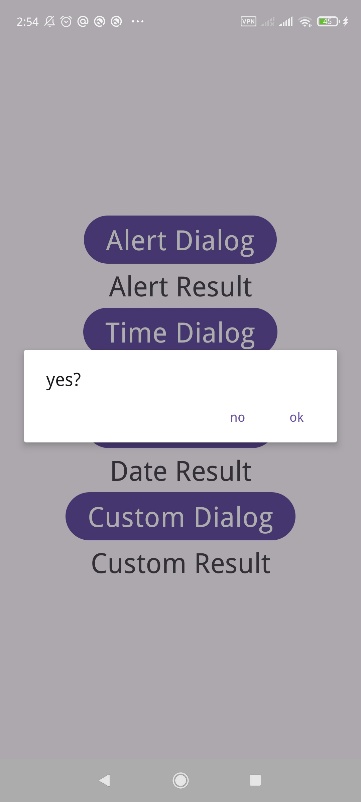


Рисунок – Диалоговое окно AlertDialog

После нажатия на кнопку “Time Dialog” открылось диалоговое окно выбора времени (Рисунок 45).

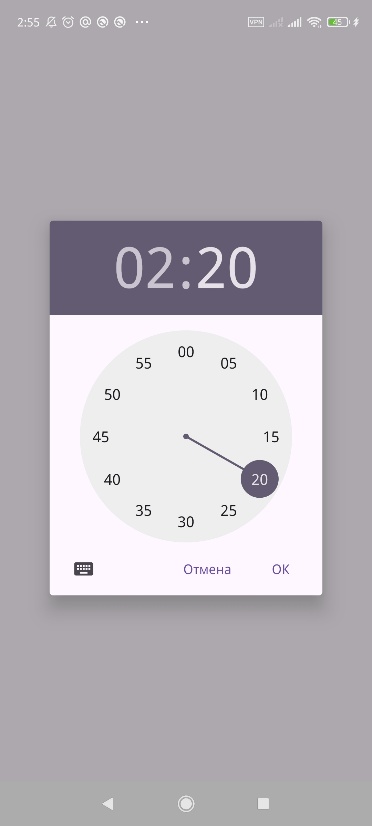


Рисунок – Диалоговое окно TimePickerDialog

После нажатия на кнопку “Date Dialog” открылось диалоговое окно выбора даты (Рисунок 46).

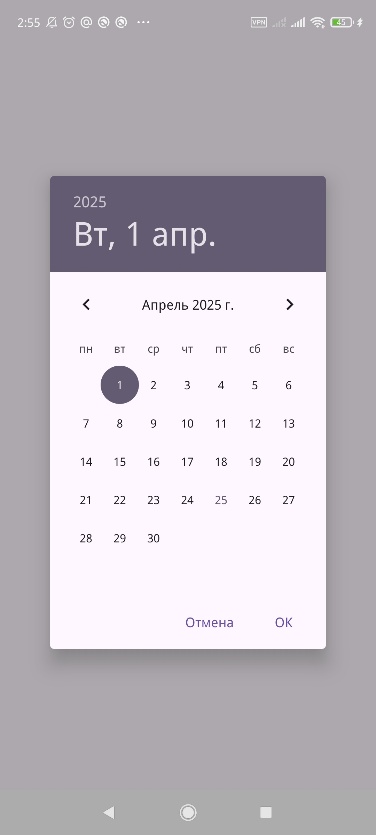


Рисунок – Диалоговое окно DataPickerDialog

После нажатия на кнопку “Custom Dialog” открылось пользовательское диалоговое окно (Рисунок 47).

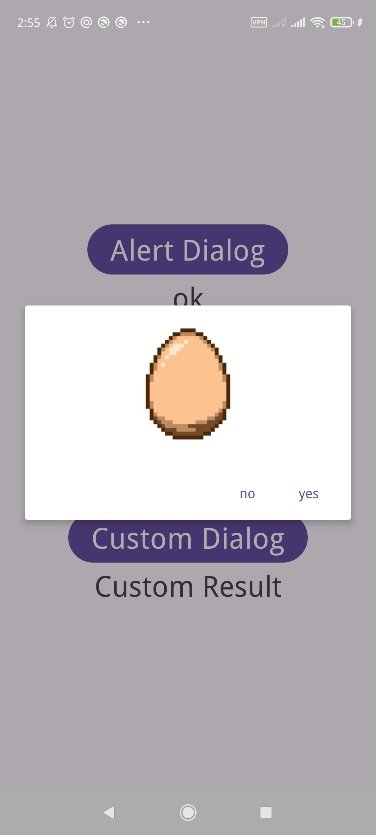


Рисунок – Диалоговое окно CustomDialog

После взаимодействия с каждым диалоговым окном соответствующее текстовое поле изменилось в соответствии с выбранным вариантом в каждом из диалогов (Рисунок 48).

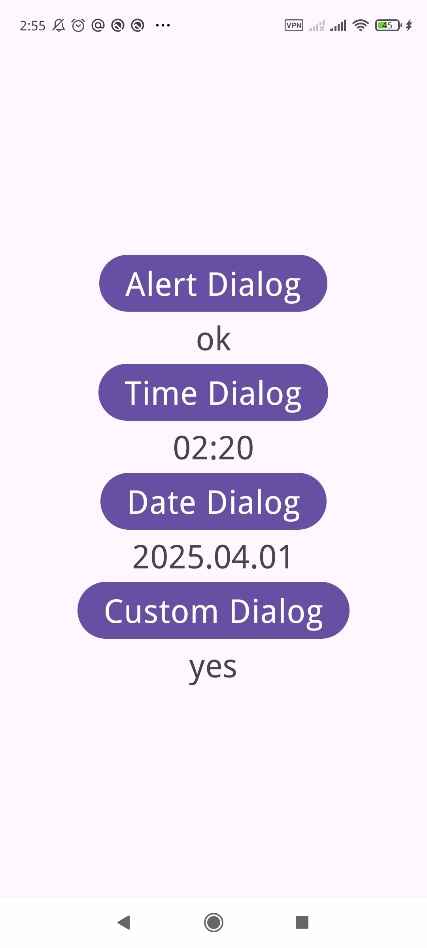


Рисунок – Экран приложения после изменений

Тестирование прошло успешно.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы было изучено использование различных типов диалоговых окон в Android-приложениях. Было проведено знакомство с базовыми диалогами (AlertDialog), специализированными элементами выбора даты и времени (DatePickerDialog и TimePickerDialog), а также реализовали собственный тип диалогового окна с пользовательской разметкой. Особое внимание было уделено механизмам передачи данных между диалоговыми окнами и основной активностью. Также были изучены принципы работы Service — важного компонента Android для выполнения фоновых операций. Был создан и запущен собственный сервис.