1. Основные компоненты Android-приложения. View – компоненты. Группы компонентов.
2. Файл layout. Расположение. Назначение. Формат. Содержание: набор View по умолчанию, виды компоновки компонентов, присвоение идентификаторов, обращение к компонентам через код. Виды разметок. Комбинирование компонентов. Подключение разметки.
3. Структура Android приложения. Список каталогов, назначение файлов. Папки src, gen, assets, res, файлы AndroidManifest.xml, build.gradle.
4. Activity: назначение, три основных состояния, переход между состояниями. Перечень методов, вызываемых системой для изменения Activity.
5. Жизненный цикл Activity. Содержание основных методов Activity.
6. Activity: запуск второй activity через intent, получение результата работы второй activity. Создание класса на основе суперкласса android.app.Activity, дополнительные записи в манифесте.
7. Назначение и применение Intent. Конструктор Intent и его параметры. Явный и неявный вызовы Activity.
8. Task – группа. Назначение Task. Экран Home. Состояния Activity из Task. Набор back stack.
9. Service: назначение, виды, основные методы. Жизненный цикл сервисов. Запуск и остановка сервиса. Методы onCreate, onDestroy. Необходимые правки манифеста.
10. Передача данных в сервис. Параметры для старта сервиса onStartCommand (Intent, флаги, счетчик вызовов) и остановки сервиса stopSelf, stopSelfResult. Необходимые правки манифеста.
11. Получение обратной связи из сервиса с помощью PendingIntent. Метод createPendingResult. Вызов PendingIntent. Необходимые правки манифеста.
12. Получение обратной связи из сервиса с помощью BroadcastReceiver. Метод BroadcastReceiver.. Создание и настройка IntentFilter. Регистрация IntentFilter и Action. Получение Intent-ов BroadcastReciver-ом. Поиск и извлечение данных.
13. Использование биндинга для синхронного подключения к сервису. ServiceConnection. Объект ServiceConnection. Методы onServiceConnection, bindService, unbindService, onServiceDisconnected.
14. Обмен данными в локальном биндинге. Процесс для сервиса и приложения. Интерфейс IBinder.
15. Многопоточность и процессы: основные принципы в Android. Класс Thread. Методы start, run объекта Runnable. Требования к реализации. Пример реализации.
16. Работа с файловой системой. Чтение и сохранение файлов. Класс android.content.Context, его назначение и методы: deleteFile, fileList, getCacheDir, getDir, getExternalCacheDir, getExternalFilesDir, getFileStreamPath, openFileInput, openFileOutput. Режимы MODE\_PRIVATE, MODE\_APPEND. Примеры.
17. Работа с базами данных SQLite. Подключение к базе данных SQLite. Классы SQLiteDatabase, SQLiteCursor, SQLiteQueryBuilder, SQLiteStatement, SQLiteOpenHelper пакета android.database.sqlite. Система типов: INTEGER, REAL, TEXT, BLOB. Создание и открытие базы данных. Примеры.
18. Класс SQLiteOpenHelper для получения данных из SQLite. Применение SQLiteOpenHelper: методы onCreate(), onUpgrade(). объект SQLiteDatabase. Запросы к SQLite средствами метода execSQL().
19. Объект Cursor для запросов к объекту SQLiteDatabase. Методы getCount(), moveToFirst(), get\*(columnIndex). Класс CursorAdapter и подкласс SimpleCursorAdapter. Параметры конструктора SimpleCursorAdapter.
20. Библиотека Room. Назначение, применение, минимальная реализация. Обзор аннотаций Entity, Dao Database. Класс RoomDatabase. Примеры.
21. Аннотация (класс) Entity из библиотеки Room: создание таблицы, описание полей, типов, модификаторов доступа, ключей. Методы Setter, Getter, конструктор, индексы: назначение, описание, примеры.
22. Интерфейс Dao. Аннотации Insert, Update: добавление возврат данных. Режимы вставки. Метод Delete, аннотация @Transaction. Примеры.
23. Получение данных с помощью запросов Query библиотеки Room. List, массив, Cursor. Метод с аннотацией Query: параметры запроса, возвращаемые типы. LiveData, Subsets, Insert, Update и Delete. Примеры.
24. Класс Fragment. Фрагменты в Android. Сущность, назначение, описание. Элементы управления фрагментом. Библиотека AndroidX Fragment library и её подключение. Добавление фрагмента в Activity: элемент FragmentContainerView, атрибут android:name. Добавление шаблона и логики фрагмента.
25. Жизненный цикл фрагмента. Константы перечисления Lifecycle.State: сушность, методы реализации.
26. Класс RecyclerView.Adapter: назначение, особенности, минимальная реализации собственного адаптера списка. Методы onCreateViewHolder, onBindViewHolder, getItemCount.
27. Паттерн MVP: описание, назначение, пример использования. Преимущества и недостатки. Библиотека moxy-community: схема, возможности. Аннотации Moxy: @InjectPresenter, @InjectViewState, @StateStrategyType, @GenerateViewState. Компоненты Moxy: MoxyPresenter, MvpView, MvpViewState.
28. Сенсоры. Ускорение, ориентация. Классы и методы: SensorManager, getSensorList, getDefaultSensor. registerListener. Интерфейс SensorEventListener и методы onAccuracyChanged, onSensorChanged. Константы свойств сенсора. Примеры.
29. Определение местоположения. GPS координаты. Интерфейсы, классы, методы и константы: LocationManager, requestLocationUpdates, GPS\_PROVIDER, NETWORK\_PROVIDER, LocationListener, onLocationChanged, showLocation, onProviderDisabled, checkEnabled, onProviderEnabled, checkEnabled, getLastKnownLocation, onStatusChanged, OUT\_OF\_SERVICE, TEMPORARILY\_UNAVAILABLE, AVAILABLE. Назначение, примеры использования.
30. Google maps. Создание и настройка проекта. Карта и события. API ключ, Google Play services. Разрешения manifest. Метод getMap получения объекта GoogleMap. Метод setMapType. Управление картой методом getUiSetting. Интерфейсы, слушетели и методы: setOnMapClick, OnMapClickListener интерфейса, OnMapClick.
31. **Основные компоненты Android-приложения. View – компоненты. Группы компонентов.**

Основные компоненты Android-приложения включают:

1.Activity: Это компонент, который представляет собой один экран пользовательского интерфейса или одно действие, которое пользователь может выполнить. Activity обрабатывает взаимодействие с пользователем и отображает пользовательский интерфейс.

2.Service: Это компонент, который выполняет долговременные операции в фоновом режиме без прямого взаимодействия с пользователем. Service может выполнять задачи, такие как загрузка файлов, воспроизведение музыки или обработка сетевых запросов.

3.BroadcastReceiver: Это компонент, который отслеживает и реагирует на системные или пользовательские события, такие как прием сообщений, изменение состояния сети или зарядки устройства. BroadcastReceiver позволяет приложению получать и обрабатывать сообщения, даже если оно не активно.

4.ContentProvider: Это компонент, который предоставляет структурированный способ обмена данными между приложениями. ContentProvider позволяет приложению предоставлять и получать доступ к данным, хранящимся в файловой системе, базе данных или других источниках данных.

Теперь перейдем к компонентам View внутри Activity:

View - это компоненты пользовательского интерфейса, которые отображаются на экране и позволяют пользователю взаимодействовать с приложением. Некоторые распространенные компоненты View в Android включают:

1.TextView: Отображает текстовую информацию на экране.

2.EditText: Позволяет пользователю вводить и редактировать текст.

3.Button: Элемент управления, который реагирует на нажатие пользователя.

4.ImageView: Отображает изображение на экране.

5.ListView: Отображает список элементов, которые могут прокручиваться вертикально.

6.RecyclerView: Улучшенная версия ListView, позволяющая эффективно отображать большие наборы данных.

7.CheckBox: Позволяет пользователю выбирать один или несколько элементов из списка.

8.RadioButton: Позволяет пользователю выбирать один элемент из списка.

9.ProgressBar: Отображает индикатор прогресса, например, при загрузке данных.

10.WebView: Отображает веб-страницу внутри приложения.

Группы компонентов в Android позволяют группировать и организовывать несколько View вместе. Некоторые распространенные группы компонентов включают:

1.LinearLayout: Это группа компонентов, которая располагает своих потомков в одну линию, либо горизонтально, либо вертикально.

2.RelativeLayout: Это группа компонентов, в которой расположение каждого компонента определяется относительно других компонентов или родительской группы.

3.ConstraintLayout: Это группа компонентов, которая использует ограничения для определения позиции и выравнивания компонентов относительно друг друга.

4.FrameLayout: Это группа компонентов, которая может содержать только один видимый компонент, который находится на переднем плане.

5.GridLayout: Это группа компонентов, которая располагает своих потомков в виде таблицы с заданным количеством строк и столбцов.

**2. Файл layout. Расположение. Назначение. Формат. Содержание: набор View по умолчанию, виды компоновки компонентов, присвоение идентификаторов, обращение к компонентам через код. Виды разметок. Комбинирование компонентов. Подключение разметки.**

Файл layout в Android представляет собой XML-файл, который определяет расположение и содержание компонентов пользовательского интерфейса в Android-приложении. Он используется для создания макета или макетов, которые определяют, как компоненты должны быть размещены на экране устройства.

Назначение файла layout:

* Определение структуры пользовательского интерфейса: Файл layout определяет, как компоненты View должны быть размещены и сгруппированы на экране устройства, чтобы создать нужный пользовательский интерфейс.

Формат файла layout:

* Файл layout представляет собой XML-файл с расширением .xml.
* Обычно файлы layout находятся в папке res/layout в структуре проекта Android-приложения.

Содержание файла layout:

* Набор View по умолчанию: Файл layout может содержать набор компонентов View, таких как TextView, EditText, Button и другие, которые определяют, что должно отображаться на экране приложения.
* Виды компоновки компонентов: В файле layout вы можете использовать различные контейнеры и виды компоновки, такие как LinearLayout, RelativeLayout, ConstraintLayout и другие, чтобы определить, как компоненты должны быть размещены и организованы на экране.
* Присвоение идентификаторов: Каждый компонент View в файле layout может иметь уникальный идентификатор, который задается с помощью атрибута "android:id". Это позволяет обращаться к компонентам через код и выполнять с ними операции и манипуляции.
* Обращение к компонентам через код: После определения компонентов в файле layout вы можете обращаться к ним из кода вашего приложения, используя методы, такие как findViewById(), чтобы получить ссылку на компонент и взаимодействовать с ними, например, установить текст, добавить обработчик событий и т.д.

Виды разметок:

* Линейная разметка (LinearLayout): Размещает компоненты в одной строке или столбце.
* Относительная разметка (RelativeLayout): Определяет позицию компонентов относительно других компонентов или родительского контейнера.
* Ограничительная разметка (ConstraintLayout): Определяет ограничения между компонентами, позволяя создавать гибкие и сложные макеты.
* Сеточная разметка (GridLayout): Размещает компоненты в виде сетки с заданным количеством строк и столбцов.
* Фреймовая разметка (FrameLayout): Размещает компоненты один над другим, внутри одного контейнера.

Комбинирование компонентов:

* Вы можете комбинировать компоненты в файле layout, добавляя их в различные контейнеры и устанавливая правильные атрибуты компоновки и ограничений, чтобы достичь нужного расположения и внешнего вида пользовательского интерфейса.

Подключение разметки:

* Разметку можно подключить к активности или фрагменту с помощью методов, таких как setContentView() или LayoutInflater.inflate(). Это позволяет отображать созданный макет на экране устройства во время выполнения приложения.

1. **Структура Android приложения. Список каталогов, назначение файлов. Папки src, gen, assets, res, файлы AndroidManifest.xml, build.gradle.**

Структура Android-приложения обычно следующая:

1.Каталог "src": Этот каталог содержит исходный код вашего приложения. Он обычно разделен на подкаталоги, соответствующие различным версиям и конфигурациям вашего приложения (например, "main", "debug", "release" и т. д.).

2.Каталог "gen": Этот каталог содержит автоматически сгенерированный код, включая классы R.java, которые содержат ссылки на ресурсы вашего приложения (например, макеты, изображения, строки и т. д.).

3.Каталог "assets": Этот каталог используется для хранения файлов, которые должны быть включены в ваше приложение и доступны через объект AssetManager. Например, файлы баз данных, шрифты, звуки и другие ресурсы могут быть помещены в этот каталог.

4.Каталог "res": Этот каталог содержит ресурсы вашего приложения, такие как макеты (layout), изображения (drawable), строки (string), значения (values), стили (styles) и другие. Он разделен на подкаталоги, соответствующие различным типам ресурсов.

5.Файл "AndroidManifest.xml": Этот файл является основным файлом манифеста вашего приложения. Он содержит информацию о вашем приложении, такую как название, иконка, разрешения, компоненты приложения (активности, службы, приемники и провайдеры) и другую конфигурацию.

6.Файлы "build.gradle": В Android-проекте обычно есть два файла build.gradle. Один находится в корневом каталоге проекта и другой внутри каталога модуля приложения. Они содержат настройки сборки проекта, включая зависимости, конфигурации компилятора, версии SDK, подпись приложения и другие настройки.

1. **Activity: назначение, три основных состояния, переход между состояниями. Перечень методов, вызываемых системой для изменения Activity.**

Activity в Android представляет собой компонент приложения, который предоставляет пользовательский интерфейс и служит для взаимодействия пользователя с приложением. Он отображает окно на экране устройства и обрабатывает события пользовательского ввода.

Назначение Activity:

* Отображение пользовательского интерфейса: Activity отвечает за отображение пользовательского интерфейса приложения, включая размещение компонентов View и реагирование на пользовательский ввод.

Три основных состояния Activity:

1.Resumed (активное состояние): Activity находится в активном состоянии, когда оно находится на переднем плане и пользователь может взаимодействовать с ним.

2.Paused (приостановленное состояние): Activity находится в приостановленном состоянии, когда оно частично скрыто другой Activity, но остается видимым частично. В это состояние Activity может перейти, например, при открытии диалогового окна или наложении другого Activity поверх него.

3.Stopped (остановленное состояние): Activity находится в остановленном состоянии, когда оно полностью скрыто другими Activity. Например, когда пользователь переключается на другое приложение или возвращается на главный экран устройства.

Переход между состояниями Activity:

* Activity может переходить между состояниями в зависимости от жизненного цикла приложения и действий пользователя. Например, при запуске приложения Activity проходит через последовательность состояний, таких как onCreate(), onStart(), onResume(). При сворачивании приложения или переключении на другое Activity, Activity может перейти в состояние onPause(), onStop(), а затем при возобновлении возвращаться к состоянию onResume().

Перечень методов, вызываемых системой для изменения Activity:

* onCreate(): Метод вызывается при создании Activity и используется для инициализации компонентов и настройки пользовательского интерфейса.
* onStart(): Метод вызывается, когда Activity становится видимым для пользователя. В этом методе можно выполнять подготовительные операции перед отображением.
* onResume(): Метод вызывается, когда Activity становится активным и на переднем плане. Здесь вы можете начать выполнение анимаций, запустить процессы, которые должны быть активными во время работы приложения и т. д.
* onPause(): Метод вызывается, когда Activity теряет фокус, но остается видимым частично. Здесь вы можете сохранить изменения данных или приостановить выполнение задач.
* onStop(): Метод вызывается, когда Activity перестает быть видимым. Здесь можно освободить ресурсы или сохранить состояние приложения.
* onDestroy(): Метод вызывается перед закрытием Activity. Здесь можно освободить все ресурсы и завершить незавершенные операции.
* onRestart(): Метод вызывается, когда Activity было остановлено и затем снова запущено.

1. **Жизненный цикл Activity. Содержание основных методов Activity.**

Жизненный цикл Activity описывает различные состояния и события, которые происходят при создании, запуске, остановке и уничтожении Activity. Вот основные методы Activity, которые используются для управления его жизненным циклом:

1.onCreate(): Метод вызывается при создании Activity. Здесь происходит инициализация компонентов пользовательского интерфейса, связывание данных и настройка обработчиков событий.

2.onStart(): Метод вызывается, когда Activity становится видимым для пользователя. В этом методе вы можете начать выполнение необходимых операций, которые должны быть активными, когда Activity видимо.

3.onResume(): Метод вызывается, когда Activity становится активным и на переднем плане. Здесь вы можете запустить анимации, запустить службы, которые должны работать во время работы приложения, и возобновить другие операции.

4.onPause(): Метод вызывается, когда Activity теряет фокус, но остается видимым частично. В этом методе можно сохранить изменения данных, остановить анимации или приостановить выполнение задач.

5.onStop(): Метод вызывается, когда Activity перестает быть видимым. Здесь можно освободить ресурсы, сохранить состояние или выполнить другие операции, необходимые при переходе в состояние остановки.

6.onRestart(): Метод вызывается, когда Activity было остановлено и затем снова запущено. Здесь вы можете выполнить дополнительные действия перед возобновлением работы Activity.

7.onDestroy(): Метод вызывается перед уничтожением Activity. Здесь можно освободить все ресурсы, отменить регистрацию слушателей и завершить незавершенные операции.

8.onSaveInstanceState(): Метод вызывается перед уничтожением Activity, когда оно может быть восстановлено позднее. Здесь вы можете сохранить состояние Activity в объекте Bundle для последующего восстановления.

9.onRestoreInstanceState(): Метод вызывается после создания Activity для восстановления сохраненного ранее состояния. Здесь вы можете восстановить сохраненное состояние Activity из объекта Bundle.

1. **Activity: запуск второй activity через intent, получение результата работы второй activity. Создание класса на основе суперкласса android.app.Activity, дополнительные записи в манифесте.**

Чтобы запустить вторую Activity через Intent, потребуется выполнить следующие шаги:

1.Создать новый экземпляр Intent и указывать текущий контекст и целевую Activity:

Intent intent = new Intent(context, SecondActivity.class);

2.(Опционально) Если необходимо передать дополнительные данные во вторую Activity, используются методы putExtra() или другие методы Intent для добавления этих данных. Например:

intent.putExtra("key", value);

3.Для запуска второй Activity вызывается метод startActivity() с вашим Intent:

startActivity(intent);

Вторая Activity будет запущена и отображена пользователю.

Чтобы получить результат работы второй Activity, вам нужно использовать метод startActivityForResult() вместо startActivity() и обработать результат в методе onActivityResult() в первой Activity:

1.В первой Activity запустите вторую Activity с использованием startActivityForResult().

Здесь "requestCode" - это целочисленный код, который вы определяете для идентификации этого запроса.

2.Во второй Activity, когда вы хотите вернуть результат обратно в первую Activity, создайте новый Intent и добавьте данные, если необходимо.

3.Завершите вторую Activity вызовом метода finish().

4.В первой Activity переопределите метод onActivityResult() для обработки полученного результата.

Чтобы создать класс на основе суперкласса android.app.Activity, нужно создать новый Java-класс и расширить его от Activity.

Когда вы создаете новую Activity, нужно добавить запись в файле AndroidManifest.xml, чтобы зарегистрировать вторую Activity. В разделе манифеста <application> добавьте следующую строку:

<activity android:name=".SecondActivity" />

1. **Назначение и применение Intent. Конструктор Intent и его параметры. Явный и неявный вызовы Activity.**

Intent в Android используется для выполнения различных действий, таких как запуск компонентов приложения, передача данных между компонентами, вызов внешних приложений и т.д. Он представляет собой объект, который описывает намерение выполнить определенное действие.

Назначение и применение Intent:

* Запуск компонентов приложения: Intent позволяет запускать различные компоненты Android, такие как Activity, Service, BroadcastReceiver и ContentProvider.
* Передача данных: С помощью Intent можно передавать данные между компонентами приложения. Например, вы можете передать данные из одной Activity в другую, отправить широковещательное сообщение или получить данные от внешнего приложения.
* Вызов внешних приложений: Intent может быть использован для вызова действий внешних приложений, таких как открытие веб-страницы, отправка электронной почты или показ карты с определенными координатами.

Конструктор Intent и его параметры:

* Intent имеет несколько конструкторов, но наиболее распространенный конструктор принимает два параметра: контекст и класс целевой компоненты. Например:

Intent intent = new Intent(context, TargetActivity.class);

Здесь "context" - это текущий контекст, а "TargetActivity.class" - класс целевой Activity.

Кроме того, Intent может принимать дополнительные параметры, такие как:

* Action (действие): указывает намерение, которое должно быть выполнено. Например, Intent.ACTION\_VIEW для открытия веб-страницы или Intent.ACTION\_SEND для отправки данных.
* Data (данные): представляет собой URI или URL, связанные с намерением. Например, URI файла, который будет открыт или URL веб-страницы.
* Extra (дополнительные данные): позволяет передавать дополнительные данные в Intent. Это может быть любой тип данных, такой как строки, числа, булевые значения и другие. Дополнительные данные добавляются с помощью метода putExtra().

Явный и неявный вызовы Activity:

* Явный вызов Activity: При явном вызове Activity указывается явно имя класса целевой Activity в конструкторе Intent. Например:

Intent intent = new Intent(context, TargetActivity.class);

* Неявный вызов Activity: При неявном вызове Activity указывается действие (Action) или фильтр (Filter), и система выбирает подходящую Activity для выполнения этого действия. Неявный вызов Activity основан на объявленных фильтрах Intent в манифесте приложения. Например:

Intent intent = new Intent(Intent.ACTION\_VIEW);

intent.setData(Uri.parse("http://www.example.com"));

В этом примере мы указываем действие ACTION\_VIEW и URI внешней веб-страницы. Система выберет подходящую Activity для открытия этой веб-страницы.

1. **Task – группа. Назначение Task. Экран Home. Состояния Activity из Task. Набор back stack.**

В контексте Android, Task представляет собой группу связанных Activity, которые выполняются в определенном порядке. Task обеспечивает организацию и управление активностями в приложении. Он отслеживает стек вызовов Activity и позволяет пользователям перемещаться между различными состояниями приложения.

Назначение Task:

* Организация работы с Activity: Task позволяет связать и управлять несколькими Activity в рамках одного задания. Он определяет порядок выполнения и отображения Activity в приложении.
* Управление переходами между Activity: Task обеспечивает удобный способ переключения между различными Activity в приложении. Пользователи могут перемещаться по стеку вызовов Activity, нажимая кнопку "назад" или с помощью других навигационных элементов.
* Сохранение состояния: Task позволяет сохранять состояние каждой Activity в стеке вызовов. При переходе между Activity в рамках Task можно восстановить сохраненное состояние.

Экран Home:

Экран Home в Android представляет собой основной экран устройства, на котором расположены значки и виджеты приложений. Когда пользователь нажимает кнопку "Home" на устройстве, он возвращается на этот экран. Здесь пользователь может запустить другие приложения или вернуться к уже запущенным приложениям.

Состояния Activity в Task:

Activity в Task могут находиться в различных состояниях в зависимости от их видимости и взаимодействия с пользователем. Основные состояния Activity в Task:

* Active (активное): Activity находится в переднем плане и взаимодействует с пользователем.
* Paused (приостановленное): Activity становится невидимым, но остается в состоянии приостановки, сохраняя свое состояние и ресурсы.
* Stopped (остановленное): Activity становится полностью невидимым и может быть уничтожено системой, если необходимо освободить память.

Набор back stack:

Back stack представляет собой стек, который хранит информацию о вызванных Activity в рамках Task. Каждый раз, когда новая Activity запускается в Task, она добавляется в вершину стека. Пользователь может перемещаться по стеку, нажимая кнопку "назад", что вызывает переход к предыдущей Activity в стеке.

Back stack позволяет сохранять порядок вызова Activity и восстанавливать предыдущие состояния при переходах между Activity в рамках Task.

1. **Service: назначение, виды, основные методы. Жизненный цикл сервисов. Запуск и остановка сервиса. Методы onCreate, onDestroy. Необходимые правки манифеста.**

Сервисы в Android Studio представляют собой компоненты приложения, выполняющие длительные операции в фоновом режиме без непосредственного взаимодействия с пользователем. Давайте рассмотрим основные аспекты, связанные с сервисами в Android Studio.

Назначение и виды сервисов:

Сервисы в Android Studio используются для выполнения операций, которые могут продолжаться даже после того, как пользователь свернул приложение или перешел к другому приложению. Они могут выполняться в фоновом режиме и использоваться для различных задач, таких как загрузка файлов, воспроизведение музыки, сетевые операции и т. д.

Существует два основных вида сервисов:

Started (запущенные) сервисы: запускаются приложением с помощью метода startService(). Они выполняют задачу в фоновом режиме, независимо от активности, которая его вызвала, и продолжают работу, пока явно не будут остановлены или пока задача не будет выполнена.

Bound (привязанные) сервисы: устанавливают связь между компонентами приложения, такими как активности, и сервисом. Они позволяют компонентам взаимодействовать с сервисом путем вызова его методов. Когда все связанные компоненты отсоединены, система автоматически уничтожает сервис.

Основные методы сервисов:

Некоторые из основных методов, связанных с жизненным циклом сервисов, включают:

onCreate(): вызывается при создании сервиса. Здесь можно выполнить инициализацию, настройку ресурсов и другие подготовительные действия.

onStartCommand(): вызывается при запуске сервиса. Здесь можно определить, какое действие должен выполнить сервис.

onBind(): вызывается при привязке компонента (например, активности) к сервису. Возвращает объект IBinder, который позволяет компоненту взаимодействовать с сервисом.

onDestroy(): вызывается перед уничтожением сервиса. Здесь можно освободить ресурсы, завершить выполнение операций и т. д.

Жизненный цикл сервисов:

Жизненный цикл сервиса включает следующие состояния:

1. Создание (onCreate()): сервис создается при вызове метода startService() или bindService().
2. Запуск (onStartCommand()): сервис переходит в это состояние, когда вызывается метод startService(). Здесь сервис выполняет свою задачу в фоновом режиме.
3. Привязка (onBind()): сервис переходит в это состояние, когда вызывается метод bindService(). Компонент, привязанный к сервису, может взаимодействовать с ним.
4. Остановка и уничтожение (onDestroy()): сервис останавливается и уничтожается, когда вызывается метод stopService() или все привязанные компоненты отсоединены.

Запуск и остановка сервиса:

Для запуска сервиса используется метод startService(), который принимает в качестве параметра Intent. Например:

Intent serviceIntent = new Intent(context, MyService.class);

startService(serviceIntent);

Для остановки сервиса используется метод stopService(). Например:

Intent serviceIntent = new Intent(context, MyService.class);

stopService(serviceIntent);

Необходимые правки манифеста:

Чтобы зарегистрировать сервис в манифесте приложения, необходимо добавить соответствующий элемент <service> в блок <application>.

Важно указать правильное имя сервиса в атрибуте android:name. Также обратите внимание на значения атрибутов android:enabled и android:exported, которые определяют доступность сервиса из других приложений и компонентов.

1. **Передача данных в сервис. Параметры для старта сервиса onStartCommand (Intent, флаги, счетчик вызовов) и остановки сервиса stopSelf, stopSelfResult. Необходимые правки манифеста.**

Передача данных в сервис в Android Studio осуществляется с помощью объекта Intent. Давайте рассмотрим различные аспекты передачи данных и параметры для метода onStartCommand() для запуска сервиса, а также методы stopSelf() и stopSelfResult() для остановки сервиса.

Передача данных в сервис:

Чтобы передать данные в сервис, вы можете использовать метод putExtra() класса Intent для добавления дополнительной информации в объект Intent перед его запуском. Например, для передачи строки в сервис вы можете сделать следующее:

Intent serviceIntent = new Intent(context, MyService.class);

serviceIntent.putExtra("key", "value");

startService(serviceIntent);

Здесь "key" - это ключ, по которому данные могут быть извлечены из объекта Intent внутри сервиса.

Внутри сервиса вы можете получить переданные данные следующим образом:

String value = getIntent().getStringExtra("key");

Здесь "key" - это тот же ключ, который вы использовали при передаче данных.

Параметры для метода onStartCommand():

Метод onStartCommand() используется для запуска сервиса и имеет несколько параметров, которые могут быть использованы для управления запуском сервиса:

Intent: объект Intent, содержащий данные, переданные сервису при его запуске. Вы можете извлечь эти данные, как описано выше.

Флаги (flags): флаги, определяющие способ запуска сервиса и его поведение. Некоторые из распространенных флагов включают START\_FLAG\_REDELIVERY (если сервис должен быть перезапущен в случае аварийного завершения) и START\_FLAG\_RETRY (если система должна повторить попытку запуска сервиса, если его запуск не удался).

Счетчик вызовов (startId): каждый раз, когда сервис запускается с помощью startService(), он получает уникальный идентификатор вызова. Этот идентификатор может использоваться для учета вызовов сервиса и его управления.

Методы stopSelf() и stopSelfResult():

Метод stopSelf() используется для остановки сервиса изнутри самого сервиса. Он прекращает выполнение сервиса и вызывает его метод onDestroy(). Пример использования:

stopSelf();

Метод stopSelfResult() также используется для остановки сервиса изнутри сервиса, но он также возвращает результат. Он позволяет сервису передать результат обратно тому компоненту, который запустил сервис. Пример использования:

int result = getResultCode();

stopSelfResult(result);

Необходимые правки манифеста:

Для зарегистрированного сервиса в манифесте не требуется специальных дополнительных правок для передачи данных или использования методов onStartCommand(), stopSelf() и stopSelfResult().

Однако важно убедиться, что соответствующий элемент <service> с указанием имени и свойств сервиса присутствует в блоке <application> вашего манифеста, как было описано в предыдущем ответе.

1. **Получение обратной связи из сервиса с помощью PendingIntent. Метод createPendingResult. Вызов PendingIntent. Необходимые правки манифеста.**

Для получения обратной связи из сервиса с использованием PendingIntent вам понадобится метод createPendingResult() и вызов PendingIntent. Давайте рассмотрим эти шаги и необходимые правки в манифесте.

Метод createPendingResult():

Метод createPendingResult() используется в активности для создания PendingIntent, который будет использоваться для получения обратной связи из сервиса. Он принимает следующие параметры:

* requestCode: уникальный код запроса, который идентифицирует этот PendingIntent.
* data: объект Intent, который будет использоваться для передачи данных обратно в активность из сервиса.

Пример использования метода createPendingResult() в активности:

// Создание PendingIntent

PendingIntent pendingIntent = createPendingResult(REQUEST\_CODE, new Intent(), 0);

Здесь REQUEST\_CODE - это уникальный код запроса.

Вызов PendingIntent из сервиса:

Чтобы отправить обратную связь из сервиса обратно в активность с использованием PendingIntent, вам необходимо создать объект Intent с данными, которые вы хотите передать, и вызвать метод send() на объекте PendingIntent.

Пример вызова PendingIntent из сервиса:

Intent resultIntent = new Intent();

resultIntent.putExtra("key", "value");

pendingIntent.send(this, RESULT\_CODE, resultIntent);

Здесь RESULT\_CODE - это код результата, который указывает, что вызов является успешным.

Необходимые правки манифеста:

Для использования PendingIntent с обратной связью из сервиса в манифесте приложения не требуется дополнительных правок.

Однако, если ваш сервис использует PendingIntent, который должен быть доступен из других приложений или компонентов, убедитесь, что в элементе <service> вашего манифеста есть соответствующий атрибут android:exported="true". Это позволит другим компонентам получать доступ к сервису через PendingIntent.

1. **Получение обратной связи из сервиса с помощью BroadcastReceiver. Метод BroadcastReceiver.. Создание и настройка IntentFilter. Регистрация IntentFilter и Action. Получение Intent-ов BroadcastReciver-ом. Поиск и извлечение данных.**

Для получения обратной связи из сервиса с использованием BroadcastReceiver вам понадобится создать и настроить IntentFilter, зарегистрировать BroadcastReceiver, а затем получить Intent-ы и извлечь данные. Давайте рассмотрим эти шаги подробнее:

**Создание и настройка IntentFilter:** IntentFilter определяет, какие типы сообщений (Intent-ов) ваш BroadcastReceiver будет прослушивать. Вы можете настроить IntentFilter, чтобы фильтровать сообщения по определенным действиям (action), категориям (category) или типам данных (data).

Пример создания и настройки IntentFilter:

IntentFilter intentFilter = new IntentFilter();

intentFilter.addAction("com.example.myapp.MY\_ACTION");

intentFilter.addCategory("com.example.myapp.MY\_CATEGORY");

Здесь "com.example.myapp.MY\_ACTION" и "com.example.myapp.MY\_CATEGORY" - это пользовательские значения действия и категории. Вы можете использовать свои собственные значения.

**Регистрация IntentFilter и BroadcastReceiver:** Чтобы ваш BroadcastReceiver начал прослушивать сообщения, вы должны зарегистрировать его в коде вашей активности или в манифесте приложения.

* **Регистрация в коде активности:**

BroadcastReceiver myReceiver = new MyReceiver();

registerReceiver(myReceiver, intentFilter);

* **Регистрация в манифесте:** В файле манифеста вашего приложения, внутри элемента <application>, добавьте следующую запись:

<receiver

android:name=".MyReceiver"

android:enabled="true">

<intent-filter>

<action android:name="com.example.myapp.MY\_ACTION" />

<category android:name="com.example.myapp.MY\_CATEGORY" />

</intent-filter>

</receiver>

**Получение Intent-ов и извлечение данных:** Когда сервис отправляет Broadcast, ваш BroadcastReceiver получит соответствующий Intent, и вы сможете извлечь данные из этого Intent.

Пример BroadcastReceiver для получения Intent-ов и извлечения данных:

public class MyReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

if ("com.example.myapp.MY\_ACTION".equals(intent.getAction())) {

String data = intent.getStringExtra("key");

// Обработка полученных данных

}

}

}

Здесь "com.example.myapp.MY\_ACTION" - это действие, на которое ваш BroadcastReceiver реагирует. Вы также можете проверить другие свойства Intent, такие как категории, если они были заданы.

**Пример отправки Broadcast из сервиса:**

Intent intent = new Intent("com.example.myapp.MY\_ACTION");

intent.putExtra("key", "value");

sendBroadcast(intent);

Здесь "com.example.myapp.MY\_ACTION" - это действие Broadcast, которое будет использоваться для фильтрации BroadcastReceiver.

1. **Использование биндинга для синхронного подключения к сервису. ServiceConnection. Объект ServiceConnection. Методы onServiceConnection, bindService, unbindService, onServiceDisconnected.**

Для синхронного подключения к сервису в Android вы можете использовать биндинг (binding). Биндинг позволяет активности или другому компоненту приложения установить прямое соединение с сервисом и взаимодействовать с ним. Для этого вам потребуется объект ServiceConnection и вызов соответствующих методов. Давайте рассмотрим эти шаги подробнее:

**Объект ServiceConnection:** ServiceConnection - это интерфейс, который вам необходимо реализовать, чтобы установить связь с сервисом. Он предоставляет методы, которые вызываются при установке и разрыве соединения с сервисом.

Пример реализации ServiceConnection:

private ServiceConnection serviceConnection = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

// Метод вызывается при успешном подключении к сервису

MyService.LocalBinder binder = (MyService.LocalBinder) service;

MyService myService = binder.getService();

// Выполнение операций с сервисом

}

@Override

public void onServiceDisconnected(ComponentName name) {

// Метод вызывается при разрыве соединения с сервисом

}

};

**Методы для установки и разрыва соединения:**

* bindService(): Метод bindService() используется для установки связи с сервисом. Он принимает объект Intent, указывающий на сервис, и объект ServiceConnection. Этот метод позволяет вам синхронно подключиться к сервису и получить доступ к нему. Пример использования:

Intent serviceIntent = new Intent(context, MyService.class);

bindService(serviceIntent, serviceConnection, Context.BIND\_AUTO\_CREATE);

* unbindService(): Метод unbindService() используется для разрыва связи с сервисом. Он принимает объект ServiceConnection, который был использован при вызове bindService(). Пример использования:

unbindService(serviceConnection);

**Методы обратного вызова в ServiceConnection:**

* onServiceConnected(): Метод onServiceConnected() вызывается при успешном подключении к сервису. Он предоставляет объект IBinder, который вы можете использовать для получения экземпляра сервиса и выполнения операций с ним.
* onServiceDisconnected(): Метод onServiceDisconnected() вызывается при разрыве соединения с сервисом. Это может произойти, например, при аварийном завершении сервиса. В этом методе вы можете выполнить необходимые действия по обработке разрыва соединения.

**Примечание:** При использовании биндинга необходимо учитывать, что подключение и отключение от сервиса должны быть выполнены в соответствующих методах жизненного цикла вашей активности или фрагмента.

1. **Обмен данными в локальном биндинге. Процесс для сервиса и приложения. Интерфейс IBinder.**

В локальном биндинге (Local Binding) сервис и клиент (приложение) работают в одном процессе и обмениваются данными через интерфейс IBinder. Давайте рассмотрим процесс для сервиса и приложения в локальном биндинге и роль интерфейса IBinder.

**Процесс для сервиса и приложения в локальном биндинге:**

1. Создание и настройка сервиса: Вам необходимо создать сервис, который будет содержать методы для взаимодействия с клиентом. Эти методы могут быть определены в пользовательском классе-наследнике Service.
2. Реализация интерфейса IBinder: В классе-наследнике Service вы должны реализовать метод onBind(), который возвращает объект IBinder, предоставляющий клиенту интерфейс для взаимодействия с сервисом. IBinder является мостом между сервисом и клиентом.
3. Привязка клиента к сервису: В клиентской активности или фрагменте вы должны выполнить привязку (binding) к сервису с помощью метода bindService(). В результате успешной привязки вы получите объект IBinder из метода onBind() сервиса.
4. Взаимодействие с сервисом: Используя объект IBinder, полученный после привязки к сервису, клиент может вызывать методы, предоставляемые сервисом, и передавать данные между клиентом и сервисом.

**Интерфейс IBinder:** IBinder - это интерфейс, предоставляющий клиенту способ взаимодействия с сервисом. Он определяет методы, которые клиент может вызывать для выполнения операций с сервисом.

Пример реализации onBind() и IBinder:

public class MyService extends Service {

private final IBinder binder = new LocalBinder();

@Nullable

@Override

public IBinder onBind(Intent intent) {

return binder;

}

public class LocalBinder extends Binder {

public MyService getService() {

return MyService.this;

}

// Дополнительные методы, предоставляемые сервисом

}

}

Пример взаимодействия с сервисом из клиента:

private MyService myService;

private ServiceConnection serviceConnection = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

MyService.LocalBinder binder = (MyService.LocalBinder) service;

myService = binder.getService();

// Выполнение операций с сервисом

}

@Override

public void onServiceDisconnected(ComponentName name) {

myService = null;

}

};

// Привязка к сервису

Intent serviceIntent = new Intent(this, MyService.class);

bindService(serviceIntent, serviceConnection, Context.BIND\_AUTO\_CREATE);

// Вызов методов сервиса

if (myService != null) {

myService.someMethod();

}

Таким образом, интерфейс IBinder позволяет клиенту получить доступ к методам, предоставляемым сервисом, и осуществлять обмен данными между клиентом и сервисом в локальном биндинге.

1. **Многопоточность и процессы: основные принципы в Android. Класс Thread. Методы start, run объекта Runnable. Требования к реализации. Пример реализации.**

В Android многопоточность позволяет выполнять асинхронные операции, чтобы не блокировать основной пользовательский интерфейс. Основными принципами многопоточности в Android являются использование класса Thread, методов start() и run() объекта Runnable, а также соблюдение требований к реализации многопоточности. Рассмотрим эти аспекты подробнее:

**Класс Thread:** Класс Thread является основным инструментом для работы с потоками в Android. Он позволяет создавать и управлять потоками выполнения операций.

**Методы start() и run() объекта Runnable:**

* start(): Метод start() вызывается для запуска потока выполнения, созданного на основе объекта класса Thread. Он автоматически вызывает метод run() объекта Runnable в отдельном потоке.
* run(): Метод run() определяет основную логику выполнения операций в потоке. Он должен быть переопределен в объекте Runnable и содержать код, который будет выполняться в отдельном потоке.

**Требования к реализации многопоточности:** При реализации многопоточности в Android важно учитывать следующие требования:

1. Необходимо избегать обращений к элементам пользовательского интерфейса из основного потока, кроме случаев, когда это явно разрешено.
2. Работа с данными, общими для нескольких потоков, должна быть синхронизирована для избежания состояния гонки (race conditions) и других проблем многопоточности.
3. Длительные операции, такие как сетевые запросы или операции с базой данных, должны выполняться в фоновых потоках, чтобы не блокировать основной поток пользовательского интерфейса.

**Пример реализации:** Вот пример простой реализации многопоточности с использованием класса Thread и объекта Runnable:

java

Thread thread = new Thread(new Runnable() {

@Override

public void run() {

// Код операций, выполняемых в фоновом потоке

}

});

thread.start(); // Запуск потока

В этом примере мы создаем новый объект Thread, передавая ему объект Runnable в конструкторе. Затем мы вызываем метод start(), который запускает поток выполнения и автоматически вызывает метод run() объекта Runnable.

Внутри метода run() вы можете определить необходимые операции, которые должны выполняться в фоновом потоке. Например, это может быть выполнение длительной операции, загрузка данных из сети или обработка данных в фоновом режиме.

1. **Работа с файловой системой. Чтение и сохранение файлов. Класс android.content.Context, его назначение и методы: deleteFile, fileList, getCacheDir, getDir, getExternalCacheDir, getExternalFilesDir, getFileStreamPath, openFileInput, openFileOutput. Режимы MODE\_PRIVATE, MODE\_APPEND. Примеры.**

Работа с файловой системой в Android включает чтение и сохранение файлов. Для этого используется класс android.content.Context, который предоставляет методы для работы с файлами. Давайте рассмотрим назначение и основные методы класса Context, а также режимы работы с файлами и примеры их использования.

**Класс android.content.Context:** android.content.Context - это класс, представляющий контекст выполнения приложения Android. Он предоставляет доступ к ресурсам приложения, таким как файлы, базы данных, настройки и другие системные службы.

**Основные методы класса Context для работы с файлами:**

1. deleteFile(String name): Удаляет файл с указанным именем из внутренней памяти приложения.
2. fileList(): Возвращает массив имен файлов, доступных во внутренней памяти приложения.
3. getCacheDir(): Возвращает каталог для кэширования файлов во внутренней памяти приложения.
4. getDir(String name, int mode): Возвращает или создает каталог с указанным именем во внутренней памяти приложения.
5. getExternalCacheDir(): Возвращает каталог для кэширования файлов на внешнем хранилище.
6. getExternalFilesDir(String type): Возвращает каталог для сохранения файлов на внешнем хранилище.
7. getFileStreamPath(String name): Возвращает путь к файлу с указанным именем во внутренней памяти приложения.
8. openFileInput(String name): Открывает входной поток для чтения файла с указанным именем из внутренней памяти приложения.
9. openFileOutput(String name, int mode): Открывает выходной поток для записи файла с указанным именем во внутреннюю память приложения.

**Режимы MODE\_PRIVATE и MODE\_APPEND:** Параметр mode, используемый в методах openFileOutput() и getDir(), определяет режим доступа к файлу. Два распространенных режима:

* MODE\_PRIVATE: Создает файл, доступный только вашему приложению. Если файл уже существует, он будет перезаписан.
* MODE\_APPEND: Добавляет данные в конец существующего файла, если он существует. Если файл не существует, он будет создан.

**Примеры использования:**

1. Чтение файла:

try {

FileInputStream fis = openFileInput("myfile.txt");

InputStreamReader isr = new InputStreamReader(fis);

BufferedReader br = new BufferedReader(isr);

String line;

StringBuilder sb = new StringBuilder();

while ((line = br.readLine()) != null) {

sb.append(line);

}

br.close();

isr.close();

fis.close();

String fileContent = sb.toString();

// Используйте fileContent по вашему усмотрению

} catch (IOException e) {

e.printStackTrace();

}

1. Сохранение файла:

try {

FileOutputStream fos = openFileOutput("myfile.txt", Context.MODE\_PRIVATE);

String fileContent = "Hello, world!";

fos.write(fileContent.getBytes());

fos.close();

} catch (IOException e) {

e.printStackTrace();

}

1. Удаление файла:

boolean fileDeleted = deleteFile("myfile.txt");

if (fileDeleted) {

// Файл успешно удален

} else {

// Не удалось удалить файл

}

1. Получение каталога для кэширования файлов:

File cacheDir = getCacheDir();

1. Получение каталога для сохранения файлов на внешнем хранилище:

File externalFilesDir = getExternalFilesDir(Environment.DIRECTORY\_DOCUMENTS);

Обратите внимание, что некоторые методы могут требовать разрешений в манифесте приложения, например, для доступа к внешнему хранилищу

1. **Работа с базами данных SQLite. Подключение к базе данных SQLite. Классы SQLiteDatabase, SQLiteCursor, SQLiteQueryBuilder, SQLiteStatement, SQLiteOpenHelper пакета android.database.sqlite. Система типов: INTEGER, REAL, TEXT, BLOB. Создание и открытие базы данных. Примеры.**

Работа с базами данных SQLite в Android включает использование различных классов из пакета android.database.sqlite. Вот некоторые из основных классов, которые вы можете использовать:

1. SQLiteDatabase: Этот класс предоставляет методы для управления базой данных SQLite, такие как создание, открытие, закрытие, выполнение SQL-запросов и транзакций. Вот примеры создания и открытия базы данных:

*// Создание базы данных или открытие существующей*

*SQLiteDatabase database = context.openOrCreateDatabase("mydatabase.db", Context.MODE\_PRIVATE, null);*

*// Закрытие базы данных*

*database.close();*

1. SQLiteCursor: Этот класс предоставляет методы для работы с результатами SQL-запросов, возвращающих наборы строк. Вы можете использовать его для извлечения данных из результирующего набора. Вот пример:

*// Выполнение SQL-запроса и получение курсора*

*Cursor cursor = database.rawQuery("SELECT \* FROM mytable", null*

*// Перебор строк в курсоре*

*while (cursor.moveToNext()) {*

*String name = cursor.getString(cursor.getColumnIndex("name"));*

*int age = cursor.getInt(cursor.getColumnIndex("age"));*

*// Делайте что-то с данными*

*}*

*// Закрытие курсора*

*cursor.close();*

1. SQLiteQueryBuilder: Этот класс помогает создавать сложные SQL-запросы с помощью методов-строителей. Он предоставляет удобные методы для создания условий, сортировки и группировки. Вот пример:

*SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();*

*queryBuilder.setTables("mytable");*

*queryBuilder.appendWhere("age > 18");*

*Cursor cursor = queryBuilder.query(database, null, null, null, null, null, null);*

*// Обработка результата*

1. SQLiteStatement: Этот класс используется для компиляции и выполнения предварительно подготовленных SQL-инструкций. Он может быть полезен, если вам нужно многократно выполнять один и тот же запрос с различными параметрами. Вот пример:

*SQLiteStatement statement = database.compileStatement("INSERT INTO mytable (name, age) VALUES (?, ?)");*

*statement.bindString(1, "John");*

*statement.bindInt(2, 25);*

*statement.execute();*

*// Очистка и освобождение ресурсов*

*statement.clearBindings();*

1. SQLiteOpenHelper: Этот класс помогает в управлении версиями базы данных SQLite и обеспечивает удобные методы для создания, обновления и открытия базы данных. Вот пример:

*// Использование SQLiteOpenHelper для создания и открытия базы данных*

*MyDatabaseHelper dbHelper = new MyDatabaseHelper(context);*

*SQLiteDatabase database = dbHelper.getWritableDatabase();*

Система типов в SQLite включает:

* INTEGER: Целочисленный тип данных.
* REAL: Тип данных с плавающей точкой.
* TEXT: Текстовый тип данных.
* BLOB: Двоичный тип данных для хранения необработанных байтов.

Вы можете использовать эти типы данных для определения схемы таблицы при создании таблицы.

1. **Класс SQLiteOpenHelper для получения данных из SQLite. Применение SQLiteOpenHelper: методы onCreate(), onUpgrade(). объект SQLiteDatabase. Запросы к SQLite средствами метода execSQL().**

Класс SQLiteOpenHelper в Android используется для создания и обновления базы данных SQLite. Он предоставляет удобные методы для управления версиями базы данных и выполнения операций с базой данных. Вот пример использования SQLiteOpenHelper для получения данных из SQLite:

*// Создание таблицы при создании базы данных*

*db.execSQL("CREATE TABLE mytable (id INTEGER PRIMARY KEY, name TEXT)");*

*// Обновление таблицы при обновлении базы данных*

*db.execSQL("DROP TABLE IF EXISTS mytable");*

*onCreate(db);*

В приведенном выше примере В методе onCreate() определена схема таблицы mytable, которая будет создана при создании базы данных.

Теперь, чтобы получить данные из SQLite, вы можете использовать объект SQLiteDatabase. Вот пример:

*SQLiteDatabase database = dbHelper.getReadableDatabase();*

*Cursor cursor = database.rawQuery("SELECT \* FROM mytable", null);*

*while (cursor.moveToNext()) {*

*String name = cursor.getString(cursor.getColumnIndex("name"));*

*int id = cursor.getInt(cursor.getColumnIndex("id"));*

В приведенном выше примере мы вызываем метод getReadableDatabase(), чтобы получить экземпляр SQLiteDatabase. Затем мы выполняем SQL-запрос с помощью метода rawQuery(), который возвращает Cursor, содержащий результаты запроса. Мы перебираем строки в Cursor и извлекаем значения столбцов по их именам с помощью метода getColumnIndex().

Кроме того, вы можете использовать метод execSQL() для выполнения других SQL-запросов без возвращения результата. Вот пример:

*INSERT INTO mytable (name) VALUES ('John')*

*UPDATE mytable SET name = 'Jane' WHERE id = 1*

*DELETE FROM mytable WHERE id = 1*

В приведенном выше примере мы используем execSQL() для выполнения различных SQL-запросов: вставки, обновления и удаления данных.

1. **Объект Cursor для запросов к объекту SQLiteDatabase. Методы getCount(), moveToFirst(), get\*(columnIndex). Класс CursorAdapter и подкласс SimpleCursorAdapter. Параметры конструктора SimpleCursorAdapter.**

Объект Cursor в Android используется для работы с результатами запросов к объекту SQLiteDatabase. Он предоставляет методы для перемещения по результатам запроса и извлечения данных из отдельных столбцов. Некоторые из основных методов Cursor:

* getCount(): Возвращает количество строк в результате запроса.
* moveToFirst(): Перемещает курсор на первую строку результата запроса.
* getString(columnIndex): Возвращает значение столбца по указанному индексу в виде строки.
* getInt(columnIndex): Возвращает значение столбца по указанному индексу в виде целого числа.
* getFloat(columnIndex): Возвращает значение столбца по указанному индексу в виде числа с плавающей точкой.

Пример использования Cursor:

*Cursor cursor = database.rawQuery("SELECT \* FROM mytable", null);*

*int count = cursor.getCount(); // Получение количества строк в результате*

*if (cursor.moveToFirst()) {*

*do {*

*String name = cursor.getString(cursor.getColumnIndex("name"));*

*int age = cursor.getInt(cursor.getColumnIndex("age"));*

*// Делайте что-то с данными*

*} while (cursor.moveToNext());*

*}*

*cursor.close();*

В приведенном выше примере мы используем Cursor для получения результатов запроса SELECT из таблицы mytable. Мы используем метод getCount() для получения количества строк в результате. Затем мы используем moveToFirst() для перемещения курсора на первую строку и moveToNext() для перехода к следующей строке (цикл do-while используется для перебора всех строк). Для каждой строки мы используем getColumnIndex() для получения индекса столбца по его имени, а затем соответствующие методы getString() или getInt() для извлечения данных. Класс CursorAdapter и его подкласс SimpleCursorAdapter используются для связывания данных из Cursor с элементами пользовательского интерфейса, например, ListView или RecyclerView. SimpleCursorAdapter является подклассом CursorAdapter и предоставляет простую реализацию для связывания данных из Cursor с представлениями элементов пользовательского интерфейса.

Параметры конструктора SimpleCursorAdapter включают:

* Context: Контекст приложения.
* Layout: Ресурс макета представления элемента пользовательского интерфейса.
* Cursor: Объект Cursor с данными.
* from: Массив имен столбцов в Cursor, из которых будут извлекаться данные.
* to: Массив ресурсов ID представлений элементов пользовательского интерфейса, в которые будут связываться данные.

Пример использования SimpleCursorAdapter:

*Cursor cursor = database.rawQuery("SELECT \* FROM mytable", null);*

*int[] to = { R.id.nameTextView, R.id.ageTextView }; // Ресурсы ID представлений элементов пользовательского интерфейса*

*String[] from = { "name", "age" }; // Имена столбцов в Cursor*

*SimpleCursorAdapter adapter = new SimpleCursorAdapter(context, R.layout.list\_item, cursor, from, to);*

*ListView listView = findViewById(R.id.listView);*

*listView.setAdapter(adapter);*

В приведенном выше примере мы создаем объект SimpleCursorAdapter для связывания данных из Cursor с представлениями элементов пользовательского интерфейса. Мы передаем контекст, ресурс макета элемента пользовательского интерфейса, Cursor, массив имен столбцов в Cursor (from) и массив ресурсов ID представлений элементов пользовательского интерфейса (to). Затем мы устанавливаем адаптер на ListView для отображения данных.

1. **Библиотека Room. Назначение, применение, минимальная реализация. Обзор аннотаций Entity, Dao Database. Класс RoomDatabase. Примеры.**

Библиотека Room - это часть Android Architecture Components и предоставляет абстракцию поверх базы данных SQLite для более удобной и надежной работы с базами данных в приложении Android. Room упрощает создание, управление и выполнение запросов к базе данных SQLite. Вот общий обзор основных компонентов и аннотаций в библиотеке Room:

1. Entity: Аннотация, применяемая к классу, который представляет таблицу в базе данных. Каждое поле класса, отмеченное аннотацией @ColumnInfo, будет представлять столбец в таблице. Для указания первичного ключа используется аннотация @PrimaryKey.

*@Entity(tableName = "users")*

*public class User {*

*@PrimaryKey*

*@ColumnInfo(name = "user\_id")*

*public int userId;*

*@ColumnInfo(name = "user\_name")*

*public String userName;*

*}*

1. Dao (Data Access Object): Аннотация, применяемая к интерфейсу, который предоставляет методы для выполнения операций с базой данных, таких как вставка, обновление, удаление и запросы. Аннотации @Insert, @Update, @Delete и @Query используются для определения соответствующих операций базы данных.

*@Dao*

*public interface UserDao {*

*@Insert*

*void insert(User user);*

*@Update*

*void update(User user);*

*@Delete*

*void delete(User user);*

*@Query("SELECT \* FROM users")*

*List<User> getUsers();*

*}*

1. Database: Абстрактный класс, представляющий базу данных SQLite. Аннотация @Database применяется к этому классу и определяет версию базы данных, а также список классов-сущностей и доступных Dao.

*@Database(entities = {User.class}, version = 1)*

*public abstract class MyAppDatabase extends RoomDatabase {*

*public abstract UserDao userDao();*

*}*

1. RoomDatabase: Базовый класс для доступа к базе данных. Вы можете создать собственный подкласс RoomDatabase, который расширяет этот класс и предоставляет методы для получения объектов Dao.

*MyAppDatabase database = Room.databaseBuilder(context, MyAppDatabase.class, "my-database")*

*.build();*

*UserDao userDao = database.userDao();*

*userDao.insert(user);*

*List<User> users = userDao.getUsers();*

В приведенном выше примере мы создаем экземпляр базы данных MyAppDatabase с помощью Room.databaseBuilder(). Затем мы получаем объект UserDao с помощью метода userDao(), чтобы выполнить операции с базой данных.

1. **Аннотация (класс) Entity из библиотеки Room: создание таблицы, описание полей, типов, модификаторов доступа, ключей. Методы Setter, Getter, конструктор, индексы: назначение, описание, примеры.**

Аннотация @Entity в библиотеке Room используется для определения класса, который представляет таблицу в базе данных SQLite. Она предоставляет возможность определения структуры таблицы, полей, их типов, модификаторов доступа и ключей. Давайте рассмотрим подробнее возможности, предоставляемые аннотацией @Entity:

1. Создание таблицы: Аннотация @Entity позволяет указать имя таблицы с помощью параметра tableName.
2. Описание полей и их типов: Поля класса, отмеченные аннотацией @ColumnInfo, будут представлять столбцы в таблице. Вы можете указать имя столбца с помощью параметра name, а также указать его тип.
3. Модификаторы доступа: Вы можете использовать модификаторы доступа, такие как public, private, protected, для определения видимости полей.
4. Ключи: В дополнение к первичному ключу (@PrimaryKey), вы также можете использовать аннотацию @Index для определения индексов в таблице. Индексы могут повысить производительность при выполнении запросов.
5. Методы Setter и Getter, конструктор: Room не требует явного определения методов Setter и Getter для полей класса. Однако, если вам необходимо использовать эти методы для доступа к полям класса, вы можете их определить вручную. Также вы можете определить конструктор для инициализации объектов класса.

*@Entity(tableName = "users")*

*public class User {*

*@PrimaryKey*

*@ColumnInfo(name = "user\_id")*

*private int userId;*

*@ColumnInfo(name = "user\_name")*

*public String userName;*

*public User(int userId, String userName) {*

*this.userId = userId;*

*this.userName = userName; }*

*// Методы Setter и Getter }*

В приведенном выше примере мы определяем конструктор для класса User, который принимает userId и userName.

1. **Интерфейс Dao. Аннотации Insert, Update: добавление возврат данных. Режимы вставки. Метод Delete, аннотация @Transaction. Примеры.**

Интерфейс Dao (Data Access Object) в библиотеке Room используется для определения методов, которые выполняют операции с базой данных, такие как вставка, обновление и удаление данных. Давайте рассмотрим некоторые из этих методов и соответствующие аннотации:

1. Аннотация @Insert: Используется для определения метода вставки данных в базу данных. По умолчанию, метод вставки не возвращает никаких данных. Однако, вы можете указать флаг OnConflictStrategy для обработки конфликтов при вставке данных.

*…*

*void insert(User user);*

*void insertOrUpdate(User user);*

*void insertAll(User... users);*

В приведенном выше примере у нас есть метод insert, который вставляет одного пользователя в базу данных. Метод insertOrUpdate также выполняет вставку, но в случае конфликта (например, если пользователь уже существует), он обновляет существующую запись вместо того, чтобы просто вставлять новую. Метод insertAll позволяет вставить несколько пользователей одновременно.

1. Аннотация @Update: Используется для определения метода обновления данных в базе данных.

…

@Update

void update(User user);

@Update

void updateAll(User... users);

В приведенном выше примере у нас есть метод update, который обновляет данные о пользователе в базе данных. Метод updateAll позволяет обновить несколько пользователей одновременно.

1. Метод Delete: Используется для определения метода удаления данных из базы данных.

…

@Delete

void delete(User user);

@Query("DELETE FROM users")

void deleteAll();

1. Аннотация @Transaction: Используется для определения метода, который будет выполнен внутри одной транзакции базы данных. Это полезно, когда вы хотите выполнить несколько операций базы данных как единое атомарное действие.

@Transaction

@Query("SELECT \* FROM users")

List<User> getUsersWithOrders();

В приведенном выше примере у нас есть метод getUsersWithOrders, который выполняет выборку пользователей из базы данных внутри одной транзакции. Это гарантирует согласованность данных и предотвращает проблемы с параллельным доступом к базе данных.

1. **Получение данных с помощью запросов Query библиотеки Room. List, массив, Cursor. Метод с аннотацией Query: параметры запроса, возвращаемые типы. LiveData, Subsets, Insert, Update и Delete. Примеры.**

Для работы с базой данных в Android с использованием библиотеки Room можно использовать запросы Query. Room предоставляет удобные аннотации и методы для выполнения различных операций с базой данных. Рассмотрим основные аспекты работы с запросами Query:

**Получение данных с помощью запросов Query:** С помощью аннотации @Query в Room можно выполнять запросы к базе данных. Вы можете определить параметры запроса, возвращаемый тип и выполнить запрос для получения данных. Room поддерживает различные типы возвращаемых данных, такие как List, массивы и Cursor.

**Примеры запросов с использованием аннотации @Query:**

Получение списка всех записей из таблицы:

@Query("SELECT \* FROM table\_name")

List<YourEntity> getAllRecords();

Получение отфильтрованного списка записей с использованием параметров запроса:

@Query("SELECT \* FROM table\_name WHERE column\_name = :value")

List<YourEntity> getFilteredRecords(String value);

Получение одиночной записи из таблицы:

@Query("SELECT \* FROM table\_name WHERE id = :id")

YourEntity getRecordById(int id);

**LiveData, Subsets, Insert, Update и Delete:** Room также поддерживает различные операции, такие как вставка (Insert), обновление (Update) и удаление (Delete) данных в базе данных. Он также предоставляет поддержку LiveData для автоматического обновления пользовательского интерфейса при изменении данных в базе.

**Примеры использования операций в Room:**

1. Вставка данных в таблицу:

@Insert

void insertRecord(YourEntity record);

1. Обновление данных в таблице:

@Update

void updateRecord(YourEntity record);

1. Удаление данных из таблицы:

@Delete

void deleteRecord(YourEntity record);

1. Использование LiveData для получения данных в режиме реального времени:

@Query("SELECT \* FROM table\_name")

LiveData<List<YourEntity>> getAllRecordsLiveData();

1. Использование Subset (часть результатов запроса) для получения определенных полей:

@Query("SELECT id, name FROM table\_name")

List<YourSubset> getRecordSubset();

В этих примерах YourEntity представляет собой модель данных, соответствующую таблице в базе данных, а YourSubset представляет собой модель данных, содержащую только определенные поля. С помощью запросов Query в Room вы можете получать данные из базы данных, вставлять, обновлять и удалять записи, а также использовать LiveData и Subsets для удобной работы с данными.

1. **Класс Fragment. Фрагменты в Android. Сущность, назначение, описание. Элементы управления фрагментом. Библиотека AndroidX Fragment library и её подключение. Добавление фрагмента в Activity: элемент FragmentContainerView, атрибут android:name. Добавление шаблона и логики фрагмента.**

Класс Fragment является основным строительным блоком пользовательского интерфейса в Android. Фрагменты представляют собой независимые модули, которые могут быть добавлены, удалены и заменены внутри активности (Activity). Рассмотрим основные аспекты фрагментов:

**Сущность и назначение фрагментов:** Фрагменты представляют отдельные части пользовательского интерфейса, которые могут быть комбинированы и переиспользованы в разных активностях. Они помогают разбить пользовательский интерфейс на модули, что упрощает его управление и создание гибкой архитектуры приложения.

**Элементы управления фрагментом:** Фрагменты могут содержать различные элементы управления, такие как кнопки, текстовые поля, изображения и другие виджеты. Они могут реагировать на пользовательские события и взаимодействовать с активностью, которая содержит фрагмент.

**Библиотека AndroidX Fragment и её подключение:** AndroidX Fragment library предоставляет улучшенные возможности для работы с фрагментами в Android. Для подключения библиотеки AndroidX Fragment в проект необходимо выполнить следующие действия:

1. В файле build.gradle приложения добавьте зависимость:

implementation 'androidx.fragment:fragment:1.3.0'

1. Синхронизируйте проект, чтобы загрузить зависимости библиотеки.

**Добавление фрагмента в Activity:** Чтобы добавить фрагмент в активность, используйте элемент FragmentContainerView в макете активности и атрибут android:name для указания класса фрагмента.

Пример использования FragmentContainerView в макете активности:

<androidx.fragment.app.FragmentContainerView

android:id="@+id/fragment\_container"

android:name="com.example.myapp.MyFragment"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" />

**Добавление шаблона и логики фрагмента:** Чтобы создать шаблон фрагмента, расширьте класс Fragment из пакета androidx.fragment.app и переопределите методы, такие как onCreateView() для создания пользовательского интерфейса фрагмента и onActivityCreated() для выполнения логики после присоединения фрагмента к активности.

Пример создания фрагмента:

// Создание пользовательского интерфейса фрагмента

View rootView = inflater.inflate(R.layout.fragment\_layout, container, false);

// Настройка элементов управления фрагмента

Button button = rootView.findViewById(R.id.button);

button.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

// Обработка события нажатия кнопки

}

});

return rootView;

}

@Override

public void onActivityCreated(Bundle savedInstanceState) {

super.onActivityCreated(savedInstanceState);

// Выполнение логики после присоединения фрагмента к активности

}

}

В этом примере fragment\_layout - это макет, содержащий пользовательский интерфейс фрагмента.

1. **Жизненный цикл фрагмента. Константы перечисления Lifecycle.State: сушность, методы реализации.**

Жизненный цикл фрагмента определяет различные состояния и события, которые происходят во время работы фрагмента. Фрагменты в Android имеют следующие основные состояния жизненного цикла:

1.Initialized (Инициализирован): Фрагмент был создан, но еще не связан с активностью или не добавлен в иерархию пользовательского интерфейса.

2.Created (Создан): Фрагмент создан и связан с активностью. В этом состоянии фрагмент может выполнять свою инициализацию и подготовку к отображению.

3.Started (Запущен): Фрагмент видим для пользователя. Активность, содержащая фрагмент, была запущена и фрагмент становится видимым на экране. В этом состоянии фрагмент может начать выполнять анимацию, запускать асинхронные задачи и подобное.

4.Resumed (Возобновлен): Фрагмент активен и получает фокус пользовательского взаимодействия. В этом состоянии фрагмент готов к обработке пользовательских действий и может взаимодействовать с пользователем.

5.Paused (Приостановлен): Фрагмент больше не в фокусе пользовательского взаимодействия. Это происходит, когда другой фрагмент или активность получает фокус. Фрагмент может сохранять состояние и приостанавливать выполнение задач.

6.Stopped (Остановлен): Фрагмент полностью скрыт для пользователя. Это происходит, когда фрагмент или активность, содержащая фрагмент, становятся невидимыми. Фрагмент может сохранять данные и освобождать ресурсы.

7.Destroyed (Уничтожен): Фрагмент полностью уничтожен и освобождает все ресурсы. Это происходит, когда фрагмент больше не нужен или когда активность, содержащая фрагмент, завершается.

Константы перечисления Lifecycle.State представляют эти состояния и предоставляют методы для реализации жизненного цикла фрагмента. Некоторые из основных констант состояний:

* INITIALIZED: Состояние инициализации фрагмента.
* CREATED: Состояние создания фрагмента.
* STARTED: Состояние запуска фрагмента.
* RESUMED: Состояние возобновления фрагмента.
* DESTROYED: Состояние уничтожения фрагмента.

Для реализации жизненного цикла фрагмента можно использовать класс LifecycleObserver и аннотацию @OnLifecycleEvent. Класс, реализующий интерфейс LifecycleObserver, может наблюдать за изменениями состояния фрагмента и реагировать на соответствующие события.

Примеры методов реализации состояний фрагмента:

* onCreate(): Вызывается при создании фрагмента. Здесь можно выполнять инициализацию и настройку фрагмента.
* onStart(): Вызывается, когда фрагмент становится видимым для пользователя. Можно запускать анимации или асинхронные задачи.
* onResume(): Вызывается, когда фрагмент получает фокус пользовательского взаимодействия. Здесь можно начинать взаимодействие с пользователем.
* onPause(): Вызывается, когда фрагмент теряет фокус пользовательского взаимодействия. Здесь можно сохранять данные или остановить выполнение задач.
* onStop(): Вызывается, когда фрагмент становится полностью невидимым. Здесь можно освобождать ресурсы или сохранять состояние.
* onDestroy(): Вызывается при уничтожении фрагмента. Здесь можно освобождать все ресурсы и завершать задачи.

1. **Класс RecyclerView.Adapter: назначение, особенности, минимальная реализации собственного адаптера списка. Методы onCreateViewHolder, onBindViewHolder, getItemCount.**

Компонент RecyclerView появился в Android 5.0 Lollipop и находится в разделе Containers. Раньше для отображения прокручиваемого списка использовался ListView. Со временем у него обнаружилось множество недостатков, которые было трудно исправить.

Схематично работу RecyclerView можно представить следующим образом. На экране отображаются видимые элементы списка. Когда при прокрутке списка верхний элемент уходит за пределы экрана и становится невидимым, его содержимое очищается. При этом сам "чистый" элемент помещается вниз экрана и заполняется новыми данными, иными словами переиспользуется, отсюда название Recycle.

Компонент RecyclerView не является родственником ListView и относится к семейству ViewGroup. Он часто используется как замена ListView, но его возможности шире. Внешний вид можно представить не только в виде обычного списка, но и в виде сетки. При необходимости можно быстро переключиться между разными типами отображения.

Для размещения своих дочерних элементов используется специальный менеджер макетов LayoutManager. Он может быть трёх видов.

* LinearLayoutManager - дочерние элементы размещаются вертикально (как в ListView) или горизонтально
* GridLayoutManager - дочерние элементы размещаются в сетке, как в GridView
* StaggeredGridLayoutManager - неравномерная сетка

Можно создать собственный менеджер на основе RecyclerView.LayoutManager.

RecyclerView.ItemDecoration позволяет работать с дочерними элементами: отступы, внешний вид.

ItemAnimator - отвечает за анимацию элементов при добавлении, удалении и других операций.

RecyclerView.Adapter связывает данные с компонентом и отслеживает изменения.

* notifyItemInserted(), notifyItemRemoved(), notifyItemChanged() - методы, отслеживающие добавление, удаление или изменение позиции одного элемента
* notifyItemRangeInserted(), notifyItemRangeRemoved(), notifyItemRangeChanged() - методы, отслеживающие изменение порядка элеметов

Стандартный метод notifyDataSetChanged() поддерживается, но он не приводит к внешнему изменению элементов на экране.

Программисты со стажем знают, что для создания "правильного" ListView нужно было создавать класс ViewHolder. В старых списках его можно было игнорировать. Теперь это стало необходимым условием.

### getItemCount()

Возвращает количество элементов списка. Как правило данные являются однотипными, например, список или массив строк. Адаптеру нужно знать, сколько элементов нужно предоставить компоненту, чтобы распределить ресурсы и подготовиться к отображению на экране. При работе с коллекциями или массивом мы можем просто вычислить его длину и передать это значение методу адаптера getItemCount(). В простых случаях мы можем записать код в одну строчку.

// Длинный вариант

override fun getItemCount(): Int {

return names.size

}

// Короткий вариант

override fun getItemCount() = names.size

### onCreateViewHolder

Нам нужно указать идентификатор макета для отдельного элемента списка, созданный нами ранее в файле recyclerview\_item.xml. А также вернуть наш объект класса ViewHolder. Допустим, устройство может отобразить на экране 9 элементов списка. RecyclerView создаст 11-12 элементов (с запасом). Неважно, каким большим будет ваш список, но все данные будут размещаться в тех же 11 элементах, автоматически меняя содержимое при прокрутке.

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {

val itemView =

LayoutInflater.from(parent.context)

.inflate(R.layout.recyclerview\_item, parent, false)

return MyViewHolder(itemView)

}

### onBindViewHolder()

В методе адаптера onBindViewHolder() связываем используемые текстовые метки с данными - в одном случае это значения из списка, во втором используется одна и та же строка. Параметр position отвечает за позицию в списке (индекс), по которой можно получить нужные данные.

override fun onBindViewHolder(holder: MyViewHolder, position: Int) {

holder.largeTextView.text = names[position]

holder.smallTextView.text = "кот"

}

1. **Паттерн MVP: описание, назначение, пример использования. Преимущества и недостатки. Библиотека moxy-community: схема, возможности. Аннотации Moxy: @InjectPresenter, @InjectViewState, @StateStrategyType, @GenerateViewState. Компоненты Moxy: MoxyPresenter, MvpView, MvpViewState.**

*MVP (Model-View-Presenter)* – это шаблон проектирования, который используется для разработки пользовательского интерфейса в приложении.

MVP разделяет ответственность за бизнес логику и логику отображения.

MVP состоит из следующих частей:

**• Model** представляет собой данные, которые необходимо показать пользователю. В большинстве Android-приложений моделью выступает слой, отвечающий за получение данных с бэкэнда.

**• View** – это класс, отвечающий за отображение данных. В Android-приложениях View – это обычно Activity или Fragment.

Кроме того View слушает пользовательские ивенты и делегирует их обработку в Presenter.

Например View может иметь такой код:

loginButton.setOnClickListener { presenter.onLoginClicked() }

**• Presenter** – это класс, который имеет ссылки и на View, и на Model, и расположен между ними. Presenter отвечает за обработку ивентов, приходящих из View, получение данных из Model и обновление View c полученными данными.

В Android-приложениях хорошей практикой считается делать Presenter независимым от Android SDK. Другими словами Presenter не имеет доступа к Android классам напрямую и может быть использован в plain java приложении.

Плюсы MVP:

1. Сильно упрощается написание тестов к коду;
2. Легко менять какую-то часть, не ломая при этом другую;
3. Код разбивается на мелкие кусочки, за счёт чего он становится более понятным и читабельным.

Минусы MVP:

1. Кода становится больше;
2. К этому подходу нужно привыкать;
3. На данный момент не сильно распространённый(*но известный*) подход, поэтому приходится всем рассказывать о нём.

Как работает Моху:

1. Во *View* происходит событие , которое передаётся в *Presenter*
2. *Presenter* передаёт команду во *ViewState*
3. *Presenter* стартует асинхронный запрос в *Model*
4. *ViewState* складывает команду в очередь команд, после чего передаёт её во *View*
5. *View* приводит себя в состояние, указанное в команде
6. *Presenter* получает результат запроса из *Model*
7. *Presenter* передаёт во *ViewState* две команды
8. *ViewState* сохраняет команды в очередь команд и передаёт их во *View*
9. *View* приводит себя в состояние, указанное в командах
10. Новая/пересозданная *View* присоединяется к уже имеющемуся *Presenter*
11. *ViewState* передаёт сохранённый список команд в новую/пересозданную *View*
12. Новая/пересозданная *View* приводит себя в состояние, указанное в командах

## Moxy – возможности

У Moxy есть несколько весомых преимущества перед другими решениями:

* *Presenter* не пересоздаётся при пересоздании Activity(это в разы упрощает работу с многопоточьностью)
* Автоматизация полного восстановления того, что видит пользователь при пересоздании Activity(в том числе при динамическом добавлении элементов Android View)
* Возможность из одного *Presenter* менять сразу несколько *View*(на практике оказалось чрезвычайно удобно)

Для этого в Moxy есть несколько механизмов, которые можно комбинировать между собой так, как вам будет угодно. Самыми весомыми механизмами являются аннотации, на основании которых генерируется код. А во время исполнения программы, инструмент под название MvpDelegate начинает полноценно использовать сгенерированный код.

Доступны следующие аннотации:

* @InjectPresenter – аннотация для управления жизненным циклом *Presenter*
* @InjectViewState – аннотация для привязывания *ViewState* к *Presenter*
* @StateStrategyType – аннотация для управления стратегией добавления и удаления команды из очереди команд во *ViewState*
* @GenerateViewState – аннотация для генерации кода *ViewState* для определенного интерфейса *View*

#### Moxy – MvpPresenter

В концепции *MVP*, вся бизнес-логика располагается в *Presenter* и в *Model*. Для *Presenter* в Moxy заведен класс MvpPresenter<View extends MvpView>. В MvpPresenter содержится экземпляр *ViewState*, который в тоже время должен реализовывать тот самый тип View, который пришёл в MvpPresenter. Доступ к этому экземпляру *ViewState* можно получить из метода public View getViewState(). Так же есть методы для привязывания/отвязывания *View* от *Presenter*(public void attachView(View view) и public void detachView(View view)).

В классе MvpPresenter так же есть интересный метод protected void onFirstViewAttach(). Этот метод вызывается тогда, когда к конкретному экземпляру *Presenter* первый раз будет привязана любая *View*. А когда к этому *Presenter* будет привязана *другая* *View*, к ней уже будет применено состояние из *ViewState*.

#### Moxy – MvpView и MvpViewState

Самым простым компонентом *MVP* является *View*. В дополнение ко *View*, наша библиотека имеет сущность *ViewState*, которая непосредственно связана со *View*. *ViewState* является наследником MvpViewState<View extends MvpView>. Он управляет одним, или несколькими, *View*(все одного типа View). И каждый раз, когда во *ViewState* приходит команда из *Presenter*, *ViewState* отправляет её всем *View*, о которых он знает. Также у MvpViewState есть метод protected abstract void restoreState(View view), который будет вызван когда какая-нибудь *View* будет пересоздана.

1. **Сенсоры. Ускорение, ориентация. Классы и методы: SensorManager, getSensorList, getDefaultSensor. registerListener. Интерфейс SensorEventListener и методы onAccuracyChanged, onSensorChanged. Константы свойств сенсора. Примеры.**

В Android существуют различные сенсоры, которые позволяют приложениям получать данные о различных физических величинах. Два распространенных типа сенсоров - это сенсор ускорения и сенсор ориентации:

1.Класс SensorManager:

* Назначение: Класс SensorManager предоставляет доступ к системным сенсорам и управляет их регистрацией и отменой регистрации для получения обновлений данных.
* Методы:
* getSensorList(int type): Возвращает список сенсоров заданного типа.
* getDefaultSensor(int type): Возвращает объект сенсора заданного типа, который является сенсором по умолчанию для данного типа на устройстве.
* registerListener(SensorEventListener listener, Sensor sensor, int rate): Регистрирует слушателя сенсора для получения обновлений данных с указанным сенсором и частотой обновления.

2.Интерфейс SensorEventListener:

* Назначение: Интерфейс SensorEventListener определяет методы, которые должны быть реализованы слушателем сенсора для получения обновлений данных.
* Методы:
* onAccuracyChanged(Sensor sensor, int accuracy): Вызывается при изменении точности сенсора.
* onSensorChanged(SensorEvent event): Вызывается при получении новых данных от сенсора.

3.Константы свойств сенсора (класс Sensor):

* TYPE\_ACCELEROMETER: Сенсор ускорения, предоставляющий данные о трехмерном ускорении устройства.
* TYPE\_ORIENTATION: Сенсор ориентации, предоставляющий данные о вращении устройства в пространстве.

1. **Определение местоположения. GPS координаты. Интерфейсы, классы, методы и константы: LocationManager, requestLocationUpdates, GPS\_PROVIDER, NETWORK\_PROVIDER, LocationListener, onLocationChanged, showLocation, onProviderDisabled, checkEnabled, onProviderEnabled, checkEnabled, getLastKnownLocation, onStatusChanged, OUT\_OF\_SERVICE, TEMPORARILY\_UNAVAILABLE, AVAILABLE. Назначение, примеры использования.**

Определение местоположения в Android осуществляется с помощью LocationManager, который управляет получением данных о местоположении от различных провайдеров, таких как GPS\_PROVIDER и NETWORK\_PROVIDER. Давайте рассмотрим интерфейсы, классы, методы и константы, связанные с определением местоположения:

1.Класс LocationManager:

* Назначение: Класс LocationManager предоставляет доступ к системным службам определения местоположения и управляет получением обновлений о местоположении от провайдеров.
* Методы:
* requestLocationUpdates(String provider, long minTime, float minDistance, LocationListener listener): Запрашивает обновления о местоположении от указанного провайдера с указанной минимальной частотой времени и минимальным изменением расстояния.
* getLastKnownLocation(String provider): Возвращает последнее известное местоположение от указанного провайдера.

2.Константы провайдеров:

* GPS\_PROVIDER: Провайдер GPS, который использует GPS для определения местоположения.
* NETWORK\_PROVIDER: Провайдер сети, который использует информацию о сети для определения местоположения.

3.Интерфейс LocationListener:

* Назначение: Интерфейс LocationListener определяет методы, которые должны быть реализованы слушателем местоположения для получения обновлений о местоположении.
* Методы:
* onLocationChanged(Location location): Вызывается при получении нового местоположения.
* onProviderDisabled(String provider): Вызывается, когда провайдер отключен пользователем.
* onProviderEnabled(String provider): Вызывается, когда провайдер включен пользователем.
* onStatusChanged(String provider, int status, Bundle extras): Вызывается при изменении статуса провайдера.

4.Константы статуса провайдера:

* OUT\_OF\_SERVICE: Провайдер недоступен.
* TEMPORARILY\_UNAVAILABLE: Провайдер временно недоступен.

AVAILABLE: Провайдер доступен.

1. **Google maps. Создание и настройка проекта. Карта и события. API ключ, Google Play services. Разрешения manifest. Метод getMap получения объекта GoogleMap. Метод setMapType. Управление картой методом getUiSetting. Интерфейсы, слушетели и методы: setOnMapClick, OnMapClickListener интерфейса, OnMapClick.**

Для работы с Google Maps в Android необходимо создать и настроить проект, получить API-ключ, добавить соответствующие разрешения в файл манифеста и использовать методы для управления картой и обработки событий. Давайте рассмотрим основные шаги и методы для работы с Google Maps:

1. Создание и настройка проекта:
   * Создайте проект в консоли разработчика Google (<https://console.developers.google.com/>).
   * Включите API Google Maps для своего проекта и получите API-ключ.
2. Добавление зависимости Google Play services:
   * Откройте файл build.gradle (Module: app) в вашем проекте.
   * В блоке dependencies добавьте строку: implementation 'com.google.android.gms:play-services-maps:latest\_version'
3. Разрешения в файле манифеста:
   * Добавьте следующие разрешения в файл манифеста Android:

<uses-permission android:name="android.permission.ACCESS\_FINE\_LOCATION" />

<uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION" />

<uses-permission android:name="android.permission.INTERNET" />

1. Получение объекта GoogleMap:

* В вашей активности или фрагменте получите объект GoogleMap с помощью метода getMapAsync:

SupportMapFragment mapFragment = (SupportMapFragment) getSupportFragmentManager().findFragmentById(R.id.map);

mapFragment.getMapAsync(this);

* Реализуйте интерфейс OnMapReadyCallback в вашей активности или фрагменте и переопределите метод onMapReady:

@Override

public void onMapReady(GoogleMap googleMap) {

// Ваш код для работы с объектом GoogleMap

}

1. Управление картой:

* Изменение типа карты:

googleMap.setMapType(GoogleMap.MAP\_TYPE\_NORMAL); // Обычный вид карты

googleMap.setMapType(GoogleMap.MAP\_TYPE\_SATELLITE); // Спутниковый вид карты

googleMap.setMapType(GoogleMap.MAP\_TYPE\_HYBRID); // Гибридный вид карты

* Управление интерфейсом карты (кнопки управления, компас, зум и т. д.):

UiSettings uiSettings = googleMap.getUiSettings();

uiSettings.setZoomControlsEnabled(true); // Показать кнопки зума

uiSettings.setCompassEnabled(true); // Показать компас

1. Обработка событий клика на карту:

* Реализуйте интерфейс GoogleMap.OnMapClickListener в вашей активности или фрагменте и переопределите метод onMapClick:

@Override

public void onMapClick(LatLng latLng) {

// Обработка события клика на карту

}

* Установите слушатель событий клика на карту:

googleMap.setOnMapClickListener(this);