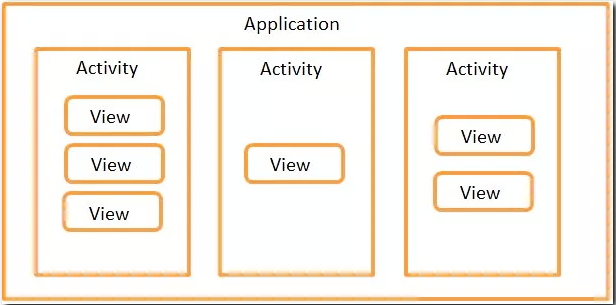
1. Основные компоненты Android-приложения. View – компоненты. Группы компонентов.
2. Файл layout. Расположение. Назначение. Формат. Содержание: набор View по умолчанию, виды компоновки компонентов, присвоение идентификаторов, обращение к компонентам через код. Виды разметок. Комбинирование компонентов. 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.

**1) Основные компоненты Android-приложения. View – компоненты. Группы компонентов.**

Если проводить аналогию с Windows, то приложение состоит из окон, называемых Activity. В конкретный момент времени обычно отображается одно Activity и занимает весь экран, а приложение переключается между ними. В качестве примера можно рассмотреть почтовое приложение. В нем одно Activity – список писем, другое – просмотр письма, третье – настройки ящика. При работе вы перемещаетесь по ним. Содержимое Activity формируется из различных компонентов, называемых View. Самые распространенные View - это кнопка, поле ввода, чекбокс и т.д.

Примерно это можно изобразить так:

Необходимо заметить, что View обычно размещаются в ViewGroup. Самый распространенный пример ViewGroup – это Layout. Layout бывает различных типов и отвечает за то, как будут расположены его дочерние View на экране (таблицей, строкой, столбцом …)

Группы компонентов:

1) Design и Text

Desing - это графическое представление экрана. Оно сейчас открыто. А Text - это текстовое представление.

2) Режимы отображения экрана

Design - в нем мы видим View компоненты так, как они обычно выглядят на экране.

Blueprint - отображаются только контуры View компонентов

3) Экран

Сюда можно добавлять различные компоненты.

4) Палитра

Это список всех View компонентов, которые можно добавлять на экран: кнопки, поля ввода, чекбоксы, прогрессбары и прочее.

5) Дерево компонентов

Здесь представлена иерархия View компонентов вашего экрана.

6) Свойства

**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**

При разработке, каждому Activity сопоставляется одноименный java-класс (наследник класса android.app.Activity). При запуске приложения, когда система должна показать Activity и дальше работать с ним, она будет вызывать методы этого класса. И от того, что мы в этих методах накодим, зависит поведение Activity.

метод onCreate – он вызывается, когда приложение создает и отображает Activity Код реализации onCreate.

super.onCreate(savedInstanceState); - это вызов метода родительского класса, выполняющий необходимые процедуры, его мы не трогаем.

setContentView(R.layout.activity\_main); Метод setContentView(int) – устанавливает содержимое Activity из layout-файла. Но в качестве аргумента мы указываем не путь к layout-файлу (res/ layout/ activity\_main.xml), а константу, которая является ID файла. Эта константа генерируется автоматически в файле R.java, который мы пока не трогаем. В этом классе будут храниться сгенерированные ID для всех ресурсов проекта (из папки res/\*), чтобы мы могли к ним обращаться. Имена этих ID-констант совпадают с именами файлов ресурсов (без расширений).

Файл res/layout/activity\_main.xml был создан средой разработки вместе с Activity. Его название запрашивалось на том же экране, где и название Activity

***Layout-файл в виде XML***

Открыв в Android Studio layout файл activity\_main или myscreen, вы видите его визуальное представление. Т.е. некий предпросмотр, как это будет выглядеть на экране. Снизу вы можете видеть две вкладки – Design и Text.

На вкладке Text есть достаточно читабельное xml-описание всех View нашего layout-файла. Названия xml-элементов - это классы View-элементов, xml-атрибуты - это параметры View-элементов, т.е. все те параметры, что мы меняем через вкладку Properties. Также вы можете вносить изменения прямо сюда и изменения будут отображаться во вкладке Design. Например, изменим текст у TextView. Вместо «new layout file myscreen for activity», напишем текст «some new text»

Обычно авторы учебников дают содержание layout-файлов именно в xml виде. Это удобно – вы можете просто скопировать фрагмент и использовать, и не надо вручную добавлять View-элементы, бегать по Properties и настраивать все руками. Я буду делать в своих проектах так же.

***Layout-файл при смене ориентации экрана***

По умолчанию мы настраиваем layout-файл под вертикальную ориентацию экрана.

У нас есть возможность создать layout-файл, который будет использоваться приложением, когда устройство находится в горизонтальной ориентации.

Создание такого файла почти не отличается от создания обычного layout-файла. Становимся на папку res/layout и создаем новый Layout resource file. Название файла указываем то же самое: myscreen. Осталось добавить спецификатор, который даст приложению понять, что этот layout-файл надо юзать в горизонтальной ориентации. Для этого в списке спецификаторов слева снизу находим Orientation

И жмем кнопку со стрелкой вправо. Тем самым мы включили использование спецификатора ориентации. Нам надо указать, что нас интересует горизонтальная ориентация: Landscape.

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

*src* – каталог исходного кода приложения. Включает подкаталоги. *gen* – каталог файлов, сгенерированных ADT в процессе разработки. *assets* – собрание произвольных файлов и каталогов. *bin* – скомпилированные файлы проекта. *libs* – дополнительные библиотеки. *res* – каталог хранения ресурсов проекта

*/res/drawable* – каталог содержащий изображения или файлы цветов. */res/layout* – содержит файлы разметки. */res/menu* – меню, заданное через XML. */res/values* – содержит файлы строковых ресурсов. */res/raw* – любые данные не в формате XML. */res/xml* – произвольные XML-файлы

*build.gradle* – система автоматической сборки, построенная на принципах Apache Ant и Apache Maven.

*Манифест* - структурный ХМL-файл, всегда имеет название AndroidManifest.xml для всех приложений. Он задает конфигурацию приложения: объявляет компоненты приложения, перечисляет любые библиотеки, связанные с приложением (помимо библиотек Android, связанных по умолчанию), и объявляет разрешения, которые приложение предоставляет.

Корневой, обязательный элемент – может быть только один элемент в файле. Должен обязательно содержать элемент. Обязательные атрибуты:

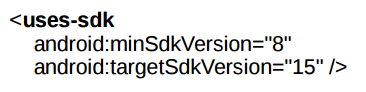
– *xmlns:android* – должен всегда содержать значение «http://schemas.android.com/apk/res/android»

– *package* – полное имя пакета для приложения.

– *android:installLocation* – место установки приложения по-умолчан.

Возможные варианты:

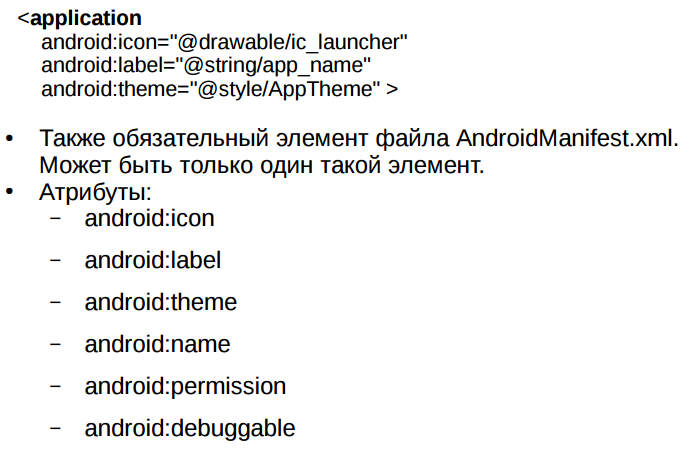
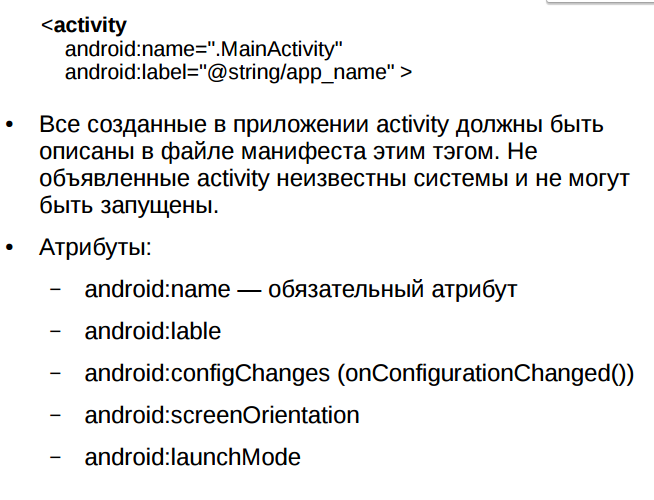
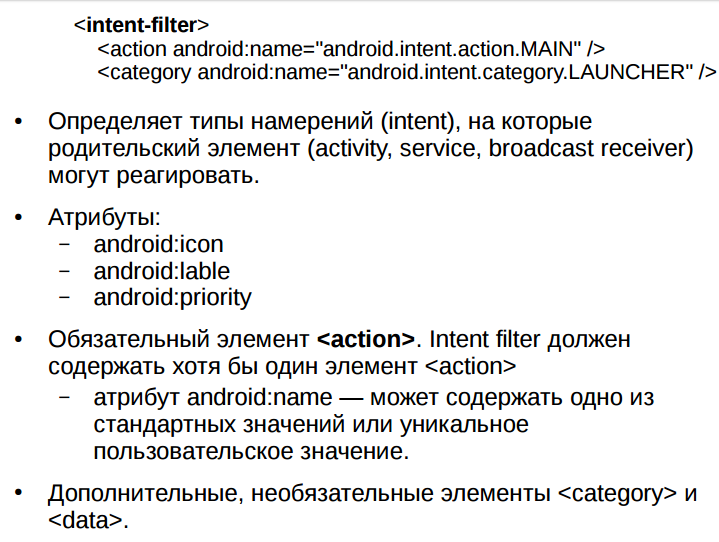
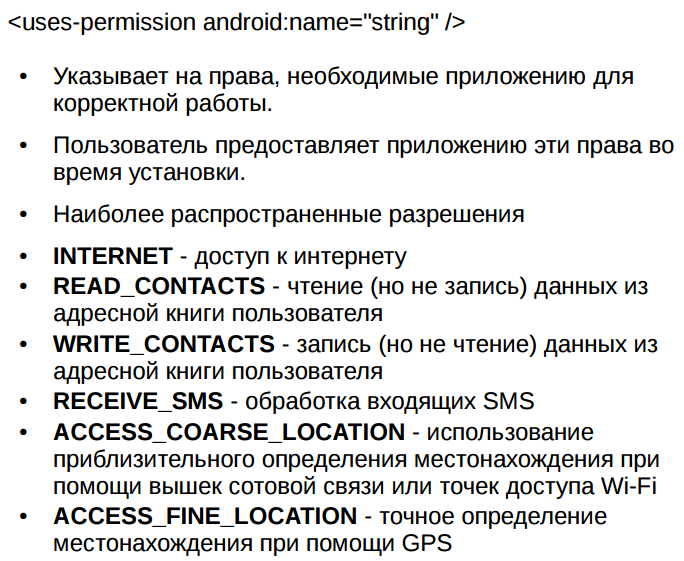
internalOnly; auto; preferExternal

  
– Несмотря на имя, указывает на версию API, а не SDK

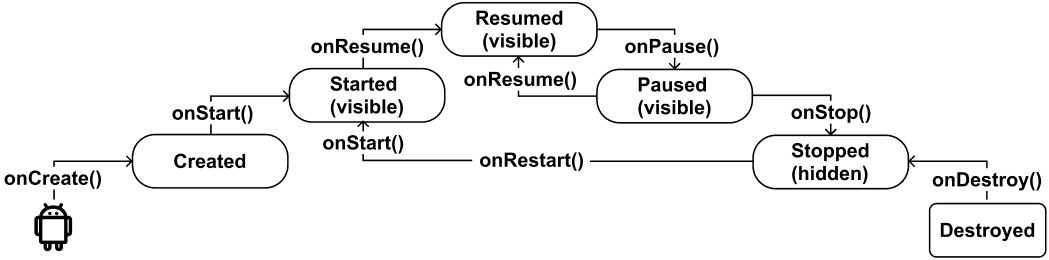
– *android:minSdkVersion* – Android не установит приложение, если версия API устройства ниже указанного значения. Если не задано, то по-умолчанию присваивается значение 1.

– *android:targetSdkVersion* – указывает на версию API, на которой приложение тестировалось и система не должна выполнять какие-либо действия для совместимости.

– *android:maxSdkVersion* – максимальная версия API. Приложение не установиться и может быть удалено после обновления системы (не рекомендуется использовать). Версии Android старше 2.0.1 (Level 6) не проверяют этот параметр.

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

Activity – компонент, представление кототого видит пользователь.  


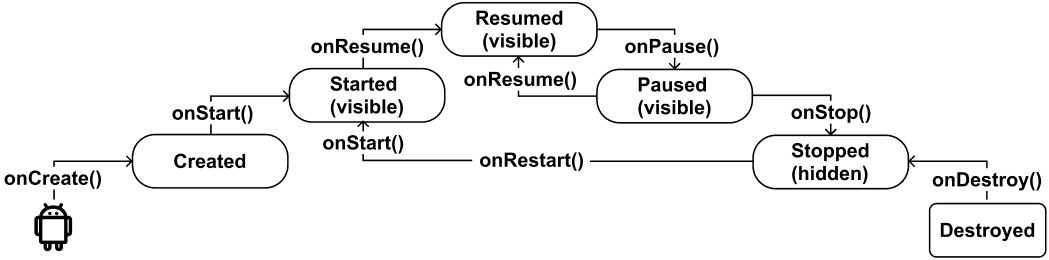
Основные callback-методы:

- *onCreate()* - вызывается единожды, при создании activity. Создает интерфейс и инициализирует поля класса activty.  
- *onPause()* - первый признак, что activity мб закрыта. Нужно остановить все ресурсоёмкие процессы и освободить ресурсы.  
- *onResume()* - вызывается всегда, когда activity становится видна пользователю. Нужно инициализировать переменные, которые освободили в onPause() и другие инициализации.  
- *onStop()* - вызывается после onPause(), если activity должна быть остановлена. Нужно освободить все ресурсы, т.к. activity мб выгружена из памяти (уничтожена) в любой момент.   
- *onRestart()* - показывает, что activity восстанавливается после того, как была остановлена (после вызова onStop()).

- *onStart()* - вызывается всегда при отображении activity. Необходимо инициализировать переменные, освобожденные в методе onStop().  
- *onDestroy()* - вызывается, когда activity полностью удаляется из памяти. Вызов - после вызова методов onPause() и onStop()  
- Вызов *finish()* из onCreate() не вызывает onPause() и onStop().  
Основные состояния:  
- *visible* – Activity запущена, видна пол-лю (взаимодействует с Activity)   
- *partially visible* – Activity частичто скрыта от пол-ля: отображается диалоговое окно, перекрывается полупрозрачной Activity   
- *hidden* – Activity полностью скрыта от пользователя. Никакой код Activity не исполняется.

Задача – набор из нескольких activity. Порядок вызова activity сохраняется в стек – стек вызова или «обратный стек».

**5)** **Жизненный цикл Activity. Содержание основных методов Activity.**

Activity – компонент, представление кототого видит пользователь.  


Основные callback-методы:

- *onCreate()* - вызывается единожды, при создании activity. Создает интерфейс и инициализирует поля класса activty.  
- *onPause()* - первый признак, что activity мб закрыта. Нужно остановить все ресурсоёмкие процессы и освободить ресурсы.  
- *onResume()* - вызывается всегда, когда activity становится видна пользователю. Нужно инициализировать переменные, которые освободили в onPause() и другие инициализации.  
- *onStop()* - вызывается после onPause(), если activity должна быть остановлена. Нужно освободить все ресурсы, т.к. activity мб выгружена из памяти (уничтожена) в любой момент.Не сохранять состояние элементов интерфейса.  
- *onRestart()* - показывает, что activity восстанавливается после того, как была остановлена (после вызова onStop()).

- *onStart()* - вызывается всегда при отображении activity. Необходимо инициализировать переменные, освобожденные в методе onStop().  
- *onDestroy()* - вызывается, когда activity полностью удаляется из памяти. Вызов - после вызова методов onPause() и onStop()  
- Вызов *finish()* из onCreate() не вызывает onPause() и onStop().  
Основные состояния:  
- *visible* – Activity запущена, видна пол-лю (взаимодействует с Activity)   
- *partially visible* – Activity частичто скрыта от пол-ля: отображается диалоговое окно, перекрывается полупрозрачной Activity   
- *hidden* – Activity полностью скрыта от пользователя. Никакой код Activity не исполняется.

Задача – набор из нескольких activity. Порядок вызова activity сохраняется в стек – стек вызова или «обратный стек».

**6) Activity: запуск 2-ой activity через intent, получение рез-та работы второй activity. Создание класса на основе суперкласса android.app.Activity, дополнительные записи в манифесте.**

[Intent](http://developer.android.com/reference/android/content/Intent.html) – объект, который обеспечивает связывание отдельных компонент во время выполнения. Intent представляет «намерение что-то сделать». Можно использовать интернет для широкого круга задач, но чаще они используются, чтобы начать другую активити.

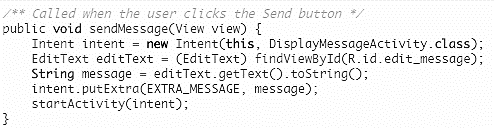
Внутри метода sendMessage(), создайте Intent для запуска активити с именем DisplayMessageActivity: *Intent intent = new Intent(this, DisplayMessageActivity.class);* Для этого требуется импортировать Intent класс: *import android.content.Intent;*

Конструктор, используемый здесь принимает два параметра:

* [*Context*](http://developer.android.com/reference/android/content/Context.html) в качестве первого параметра (this используется, поскольку [Activity](http://developer.android.com/reference/android/app/Activity.html) класс является подклассом Context)
* [*Class*](http://developer.android.com/reference/java/lang/Class.html) компонента приложения, в который система должна доставить Intent (активность, которая должна быть запущена)

Для запуска активити, вызоваем [startActivity()](http://developer.android.com/reference/android/app/Activity.html#startActivity(android.content.Intent)) и передаем в него Intent. Система получает вызов и запускает экземпляр Activity, указанный в Intent.

С помощью кода, полный sendMessage() метод, который вызывается кнопкой Отправить теперь выглядит следующим образом:

  
**Для создания новой Activity в Android Studio:**

В главном меню выберем **File>New>Activity>Blank Activity.**

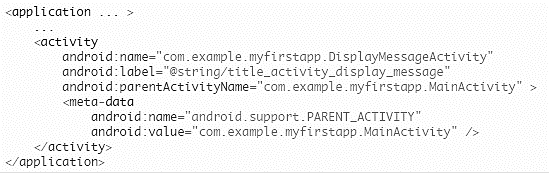
Заполним поля в окне мастера создания активити: **Activity Name**: DisplayMessageActivity; **Layout Name**: activity\_display\_message; **Title**: Сообщение; **Hierarchial Parent**: com.example.myfirstapp.MainActivity

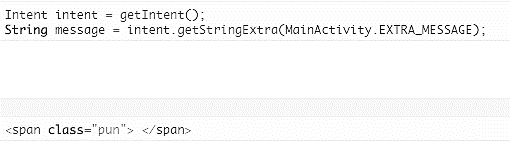
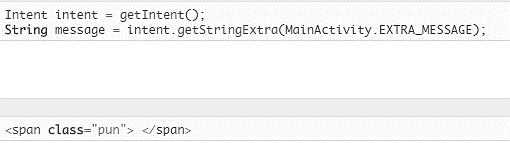
Остальные поля оставим по умолчанию. Нажмем **Finish**.

Все Activity должны быть объявлены в файле манифеста, AndroidManifest.xml, c использованием тега [<activity>](http://developer.android.com/guide/topics/manifest/activity-element.html).

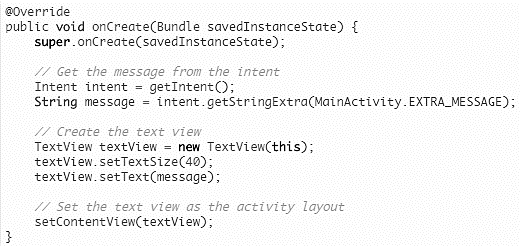
При использовании Android Studio (Eclipse) для создания своей Activity, запись в манифесте создается по умолчанию.

В других случаях можно добавить запись в манифест:

  
Атрибут [*android:parentActivityName*](http://developer.android.com/guide/topics/manifest/activity-element.html#parent) объявляет имя родительской Activity в рамках логической иерархии приложения. Система использует это значение для реализации поведения навигации по умолчанию [(Вверх](http://www.fandroid.info/navigatsiya-s-pomoshhyu-nazad-i-vverh-v-android-prilozheniyah/) на Android 4.1 (Уровень API 16) и выше).  
Каждый [Activity](http://developer.android.com/reference/android/app/Activity.html) вызывается с помощью [Intent](http://developer.android.com/reference/android/content/Intent.html), независимо от того, как пользователь между ними переходит. Можно получить [Intent](http://developer.android.com/reference/android/content/Intent.html), который запустил активити, вызвав [getIntent()](http://developer.android.com/reference/android/app/Activity.html#getIntent()) и получить данные, содержащиеся в нем. В *DisplayMessageActivity* [onCreate()](http://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)), получает интент и извлекает сообщение доставленное MainActivity:

  
  
Для отображения сообщения на экране, создадим [TextView](http://developer.android.com/reference/android/widget/TextView.html) виджет и установим текст с помощью [setText()](http://developer.android.com/reference/android/widget/TextView.html#setText(char[],%20int,%20int)). Добавим [TextView](http://developer.android.com/reference/android/widget/TextView.html) как корневое представление макета активити, передав его в [setContentView()](http://developer.android.com/reference/android/app/Activity.html#setContentView(android.view.View)).

Полный [onCreate()](http://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) метод для DisplayMessageActivity теперь выглядит следующим образом:



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

[Intent](http://developer.android.com/reference/android/content/Intent.html) – объект, который обеспечивает связывание отдельных компонент во время выполнения. Intent представляет «намерение что-то сделать». Можно использовать интернет для широкого круга задач, но чаще они используются, чтобы начать другую активити.

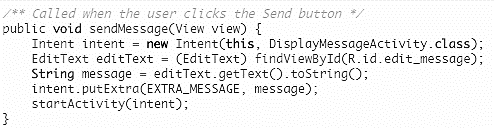
Внутри метода sendMessage(), создайте Intent для запуска активити с именем DisplayMessageActivity: *Intent intent = new Intent(this, DisplayMessageActivity.class);* Для этого требуется импортировать Intent класс: *import android.content.Intent;*

Конструктор, используемый здесь принимает два параметра:

* [*Context*](http://developer.android.com/reference/android/content/Context.html) в качестве первого параметра (this используется, поскольку [Activity](http://developer.android.com/reference/android/app/Activity.html) класс является подклассом Context)
* [*Class*](http://developer.android.com/reference/java/lang/Class.html) компонента приложения, в который система должна доставить Intent (активность, которая должна быть запущена)

Для запуска активити, вызоваем [startActivity()](http://developer.android.com/reference/android/app/Activity.html#startActivity(android.content.Intent)) и передаем в него Intent. Система получает вызов и запускает экземпляр Activity, указанный в Intent.

С помощью кода, полный sendMessage() метод, который вызывается кнопкой Отправить теперь выглядит следующим образом:

  
**Для создания новой Activity в Android Studio:**

В главном меню выберем **File>New>Activity>Blank Activity.**

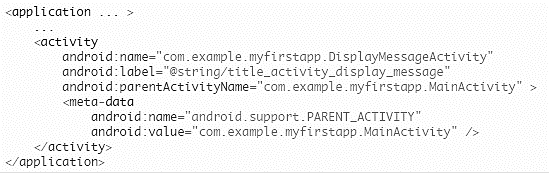
Заполним поля в окне мастера создания активити: **Activity Name**: DisplayMessageActivity; **Layout Name**: activity\_display\_message; **Title**: Сообщение; **Hierarchial Parent**: com.example.myfirstapp.MainActivity

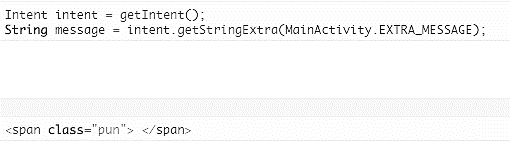
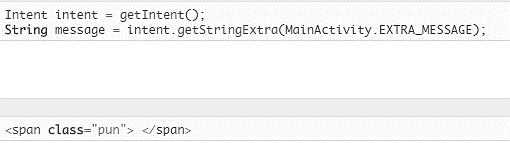
Остальные поля оставим по умолчанию. Нажмем **Finish**.

Все Activity должны быть объявлены в файле манифеста, AndroidManifest.xml, c использованием тега [<activity>](http://developer.android.com/guide/topics/manifest/activity-element.html).

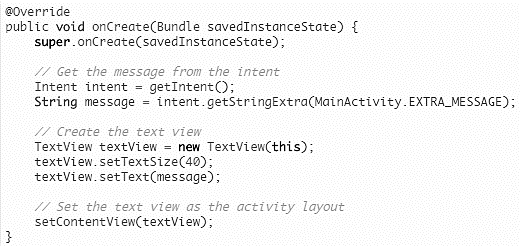
При использовании Android Studio (Eclipse) для создания своей Activity, запись в манифесте создается по умолчанию.

В других случаях можно добавить запись в манифест:

  
Атрибут [*android:parentActivityName*](http://developer.android.com/guide/topics/manifest/activity-element.html#parent) объявляет имя родительской Activity в рамках логической иерархии приложения. Система использует это значение для реализации поведения навигации по умолчанию [(Вверх](http://www.fandroid.info/navigatsiya-s-pomoshhyu-nazad-i-vverh-v-android-prilozheniyah/) на Android 4.1 (Уровень API 16) и выше).  
Каждый [Activity](http://developer.android.com/reference/android/app/Activity.html) вызывается с помощью [Intent](http://developer.android.com/reference/android/content/Intent.html), независимо от того, как пользователь между ними переходит. Можно получить [Intent](http://developer.android.com/reference/android/content/Intent.html), который запустил активити, вызвав [getIntent()](http://developer.android.com/reference/android/app/Activity.html#getIntent()) и получить данные, содержащиеся в нем. В *DisplayMessageActivity* [onCreate()](http://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)), получает интент и извлекает сообщение доставленное MainActivity:

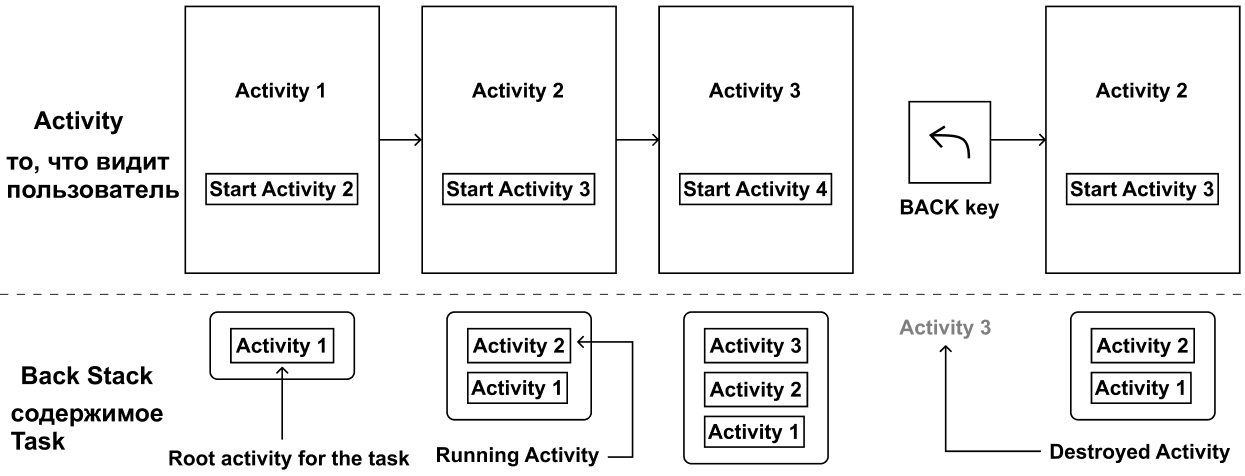
  
  
Для отображения сообщения на экране, создадим [TextView](http://developer.android.com/reference/android/widget/TextView.html) виджет и установим текст с помощью [setText()](http://developer.android.com/reference/android/widget/TextView.html#setText(char[],%20int,%20int)). Добавим [TextView](http://developer.android.com/reference/android/widget/TextView.html) как корневое представление макета активити, передав его в [setContentView()](http://developer.android.com/reference/android/app/Activity.html#setContentView(android.view.View)).

Полный [onCreate()](http://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) метод для DisplayMessageActivity теперь выглядит следующим образом:



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

Task – группа из нескольких Activity, с помощью которых пользователь выполняет определенную операцию. Обычно стартовая позиция для создания Task – экран Домой (Home). Находясь в Home, вызов происходит из списка приложений или через ярлык. Создается Task и Activity приложения помещается в этот Task как корневое. Task выходит на передний фон. Если система обнаружила, что в фоне уже существует Task этого приложения, то она выведет его на передний план и создавать ничего не будет. Когда Activity A вызывает Activity B, то Activity B помещается на верх Task и получает фокус. Activity A остается в Task, но находится в состоянии Stopped (его не видно и оно не в фокусе). Если пользователь жмет Back находясь в Activity B, то Activity B удаляется из Task и уничтожается, а Activity A оказывается теперь на верху Task и получает фокус. В каком порядке добавлялись в Task Activity, в таком порядке они содержатся в Task. Они не сортируются и не упорядочиваются внутри. Набор Activity в Task еще называют back stack.



Допустим у нас есть Task с несколькими Activity. Он на переднем фоне, мы с ним работаем сейчас. Если мы нажмем кнопку Home, то ничего не будет удалено, все Activity сохранятся в этом Task-е, а сам Task просто уйдет на задний фон и его всегда можно будет вызвать оттуда, снова вызвав приложение. Если в активном Task-е несколько раз нажимать кнопку Назад, то в итоге в стэке не останется Activity, пустой Task будет удален и пользователь увидит экран Home.

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

[Service](https://developer.android.com/reference/android/app/Service.html?hl=ru) – компонент приложения, который может выполнять длительные операции в фоновом режиме и не содержит пользовательского интерфейса. Другой компонент приложения может запустить службу, которая продолжит работу в фоновом режиме даже тогда, когда пользователь перейдет в другое приложение. Компонент может привязаться к службе для взаимодействия с ней и выполнять межпроцессное взаимодействие (IPC).

Фактически служба может принимать две формы:

- *Запущенная*. Когда компонент приложения запускает ее вызовом [startService()](https://developer.android.com/reference/android/content/Context.html?hl=ru#startService(android.content.Intent)). После запуска служба может работать в фоновом режиме в течение неограниченного времени, даже если уничтожен компонент, который ее запустил. Обычно выполняет одну операцию и не возвращает результатов вызывающему компоненту.

- *Привязанная*. Когда компонент приложения привязывается к ней вызовом [bindService()](https://developer.android.com/reference/android/content/Context.html?hl=ru#bindService(android.content.Intent,%20android.content.ServiceConnection,%20int)). Привязанная служба предлагает интерфейс клиент-сервер, который позволяет компонентам взаимодействовать со службой, отправлять запросы, получать результаты и делать это между разными процессами посредством межпроцессного взаимодействия (IPC). Привязанная служба работает пока к ней привязан другой компонент приложения. К службе могут быть привязаны несколько компонентов одновременно, но когда все они отменяют привязку, служба уничтожается.

Cлужба может работать обеими способами – может быть запущенной и допускать привязку. Это зависит от реализации пары методов обратного вызова: [onStartCommand()](https://developer.android.com/reference/android/app/Service.html?hl=ru#onStartCommand(android.content.Intent,%20int,%20int)) позволяет компонентам запускать службу, [onBind()](https://developer.android.com/reference/android/app/Service.html?hl=ru#onBind(android.content.Intent)) – выполнять привязку.

Независимо от состояния приложения, любой компонент может использовать службу так, как любой компонент может использовать операцию – запустив ее с помощью [Intent](https://developer.android.com/reference/android/content/Intent.html?hl=ru). Но можно объявить закрытую службу в файле манифеста и заблокировать доступ к ней из других приложений.

У сервиса так же, как и у Activity есть методы onCreate и onDestroy – которые срабатывают при создании и уничтожении сервиса.

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

Входящие параметры метода [onStartCommand](http://developer.android.com/reference/android/app/Service.html#onStartCommand(android.content.Intent,%20int,%20int)):

1. **Intent**. Тот, который отправляется в путь, когда мы запускаем сервис с помощью метода startService. То есть его можно использовать для передачи данных в сервис. Тут все также, как при вызове другого Activity – тоже можно передать данные с помощью intent.

2. **Флаги** запуска. Он нам не нужен, пропускаем.

3. **StartId** – **счетчик** вызовов startService, пока сервис запущен. Т.е. мы запустили сервис методом startService, сработал метод onStartCommand и получил на вход startId = 1. Вызываем еще раз метод startService, сработал метод onStartCommand и получил на вход startId = 2 и т.д. Счетчик сбрасывается, когда сервис будет остановлен методами stopService, stopSelf и пр. После этого вызовы снова идут с единицы.

Именно **startId** нужен на вход методу stopSelf(startId): этот метод дает системе понять, что конкретный вызов сервиса был успешно **обработан**. Логично, что сервис можно останавливать, когда все вызовы выполнили метод stopSelf(startId) и система видит, что не осталось необработанных вызовов, Т.е. поступила куча вызовов, они все обработались и когда последний обработанный (но при этом не обязательно последний поступивший) выполняет stopSelf(startId) – сервис останавливается.

Сервис останавливается, когда последний полученный (а не последний обработанный) вызов выполняет метод stopSelf(startId). Ранее полученные вызовы могут продолжать работать.

Метод stopSelfResult(startId) аналогичен методу stopSelf(startId), но при этом еще возвращает boolean значение – остановил он сервис или нет.

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

Как с помощью PendingIntent можно в Activity получать результаты работы сервиса:

* в Activity создаем PendingIntent с помощью метода createPendingResult
* кладем этот PendingIntent в обычный Intent, который используем для старта сервиса и вызываем startService
* в сервисе извлекаем PendingIntent из полученного в методе onStartCommand объекта Intent
* когда нам необходимо передать результаты работы из сервиса в Activity, вызываем метод send для объекта PendingIntent
* эти результаты из сервиса ловим в Activity в методе onActivityResult

Фишка PendingIntent здесь в том, что он содержит некую связь с Activity (в котором он был создан) и, когда вызывается метод send, он идет в это Activity и несет данные, если необходимо.

**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.**

Как с помощью BroadcastReceiver можно в Activity получать результаты работы сервиса:

* в Activity создаем BroadcastReceiver и IntentFilter, настроенный на определенный Action, регистрируем (включаем) эту пару. Теперь BroadcastReceiver будет получать Intent-ы подходящие под условия IntentFilter
* в сервисе, когда нам понадобится передать данные в Activity, мы создаем Intent (с Action из предыдущего пункта), кладем в него данные, которые хотим передать, и посылаем его на поиски BroadcastReceiver
* BroadcastReceiver в Activity ловит этот Intent и извлекает из него данные

Все аналогично вызовам Activity с использованием Action и IntentFilter. Если Action в Intent (отправленном из сервиса) и в IntentFilter (у BroadcastReceiver в Activity) совпадут, то BroadcastReceiver получит этот Intent и сможет извлечь данные для Activity.

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

Существует синхронный способ взаимодействия с сервисом. Он достигается с помощью биндинга (binding, подключение). Мы подключаемся к сервису и можем взаимодействовать с ним путем обычного вызова методов с передачей данных и получением результатов.

Для запуска и остановки сервиса используются методы startService и stopService. Для биндинга используются методы bindService и unbindService.

В onCreate мы создаем Intent, который позволит нам добраться до сервиса.

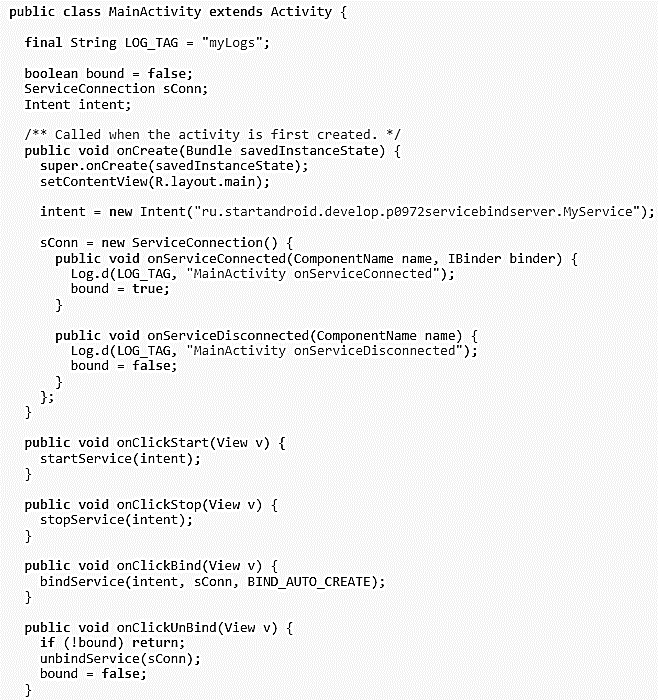
Объект ServiceConnection позволит определить, когда мы подключились к сервису и когда связь с сервисом потеряна (если сервис был убит системой при нехватке памяти). При подключении к сервису сработает метод onServiceConnected. На вход он получает имя компонента-сервиса и объект Binder для взаимодействия с сервисом. При потере связи сработает метод onServiceDisconnected.

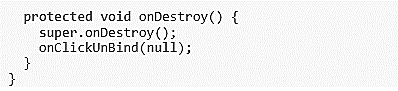
Переменную bound используем для того, чтобы знать – подключены мы в данный момент к сервису или нет. Соответственно при подключении мы переводим ее в true, а при потере связи в false.

В onClickStart мы запускаем сервис, в onClickStop – останавливаем.

В onClickBind – соединяемся с сервисом, используя метод bindService. На вход передаем Intent, ServiceConnection и флаг BIND\_AUTO\_CREATE, означающий, что, если сервис, к которому мы пытаемся подключиться, не работает, то он будет запущен.

В onClickUnBind с помощью bound проверяем, что соединение уже установлено. Далее отсоединяемся методом unbindService, на вход передавая ему ServiceConnection. И в bound пишем false, т.к. мы сами разорвали соединение. Метод onServiceDisconnected не сработает при явном отключении.





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

Метод onBind возвращает объект, наследующий интерфейс IBinder. Проще –объект Binder расширить необходимыми методами: создаем класс MyBinder с предком Binder и рисуем в нем свои методы. При биндинге, в onServiceConnected мы получаем объект Binder. Можно привести его к типу MyBinder (из сервиса) и вызывать его методы, а реализация будет срабатывать в сервисе, где мы описывали этот класс. Это сработает, если сервис и приложение выполняются в одном процессе. Поэтому такой биндинг называется локальным.

Timer – повторение действия через заданный промежуток времени. Принцип действия: TimerTask – задача, которую Timer периодически выполняет. Метод run – действия задачи. Для объекта Timer вызываем метод schedule, куда передаем задачу TimerTask; время, через которое начнется выполнение, и период повтора. Отмена выполнения задачи – метод cancel для TimerTask. Отмененную задачу нельзя запланировать. Если снова надо включить – нужно создать новый экземпляр TimerTask и скормить его таймеру.

Метод onCreate – создаем таймер и выполняем метод schedule, в котором стартует задача. Метод schedule проверяет, что задача уже создана и отменяет ее. Далее планирует новую с отложенным на 1000 мс запуском и периодом = interval. Перезапускает задачу, используя текущий интервал повтора; если задача не создана, то создает. Сама задача просто выводит в лог текст run. Если interval = 0, то ничего не делаем.

Метод upInterval получает на вход значение, увелич. interval на это значение и перезапускает задачу. Задача будет повторяться реже. Метод downInterval получает на вход значение, умен. interval на это значение и перезапускает задачу. Задача будет повторяться чаще.

onBind возвращает binder. Это объект класса MyBinder. MyBinder расширяет стандартный Binder, добавляем в него метод getService – возвращает MyService. В onClickUnBind с помощью bound проверяем, что соединение установлено. Отсоединяемся методом unbindService, на вход передаем ему ServiceConnection. И в bound пишем false, т.к. соединение разорвано. Метод onServiceDisconnected не сработает при явном отключении.

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

**Treads**

● Каждое приложение работает в «главном» потоке  
(интерфейсном потоке – UI thread).  
● Callback-методы выполняются в интерфейсном потоке.  
● Диалог "application not responding" (ANR) («приложение не  
отвечает») при блокировке интерфейсного потока.  
● 2 правила работы с потоками Android:  
1. Не блокировать UI потоки;  
2. Не получать доступ к элементам UI из вне UI потока.

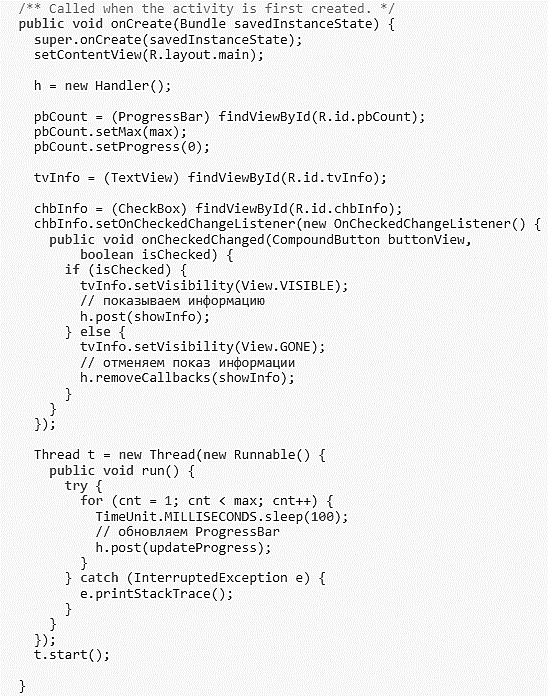
Методы для получения доступа к UI потоку из рабочего потока:  
● Activity.runOnUiThread(Runnable)  
● View.post(Runnable)  
● View.postDelayed(Runnable, long)

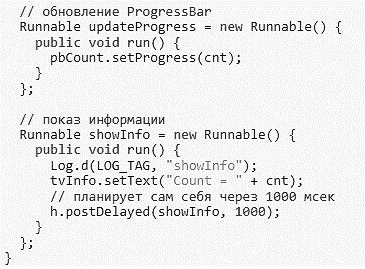
Для создания нового потока нужно реализовать интерфейс Runnable. Можно создать поток из любого объекта, реализующего интерфейс Runnable и объявить метод run(). Внутри метода run() размещается код для нового потока. Этот поток завершится, когда метод вернёт управление.

Когда объявлен новый класс с интерфейсом Runnable, нужно использовать конструктор: *Thread(Runnable объект\_потока, String имя\_потока)*

В первом параметре указывается экземпляр класса, реализующего интерфейс. Он определяет, где начнётся выполнение потока. Во втором параметре передаётся имя потока.

После создания нового потока, его нужно запустить с помощью метода start(), который выполняет вызов метода run().





**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.**

Работа с настройками уровня activity и приложения позволяет сохранить небольшие данные отдельных типов (string, int). Для работы с большими массивами данных (графические файлы, файлы мультимедиа и т.д.) придется обращаться к файловой системе.

Для работы с файлами абстрактный класс android.content.Context определяет ряд методов:

* *boolean deleteFile (String name):* удаляет определенный файл
* *String[] fileList ():* получает все файлы, которые содержатся в подкаталоге /files в каталоге приложения (далее - КП)
* *File getCacheDir():* получает ссылку на подкаталог cache в КП
* *File getDir(String dirName, int mode):* получает ссылку на подкаталог в КП, если такого подкаталога нет, то он создается
* *File getExternalCacheDir():* получает ссылку на папку /cache внешней файловой системы устройства
* *File getExternalFilesDir(String type):* получает ссылку на каталог /files внешней файловой системы устройства
* *File getFileStreamPath(String filename):* возвращает абсолютный путь к файлу в файловой системе
* *FileInputStream openFileInput(String filename):* откр. файл для чтения
* *FileOutputStream openFileOutput (String name, int mode):* открывает файл для записи

Все файлы, которые создаются и редактируются в приложении хранятся в подкаталоге /files в КП.

Для непосредственного чтения и записи файлов применяются также стандартные классы Java из пакета java.io.

Итак, применим функционал чтения-записи файлов в приложении. Пусть у нас будет примитивная разметка layout. {1 код} Поле EditText – для ввода текста, а TextView - для вывода сохраненного текста. Для сохранения и восстановления текста добавляются две кнопки. Теперь в коде Activity пропишем обработчики кнопок с сохранением и чтением файла {2 код} При нажатии на кнопку сохранения будет создаваться поток вывода *FileOutputStream fos = openFileOutput (FILE\_NAME, MODE\_PRIVATE)*. При этом будет использоваться режим MODE\_PRIVAT (если файл существует, то он будет перезаписан). Система позволяет создавать файлы с двумя разными режимами: 1) *MODE\_PRIVATE*: файлы доступны только владельцу приложения (режим по умолчанию); 2) *MODE\_APPEND*: данные могут быть добавлены в конец файла. Если надо было дописать файл, тогда надо было использовать режим MODE\_APPEND: *FileOutputStream fos = openFileOutput (FILE\_NAME, MODE\_APPEND).* Для чтения файла применяется поток ввода FileInputStream: *FileInputStream fin = openFileInput (FILE\_NAME);*

{1 код}

<?xml version="1.0" encoding="utf-8"?>

<androidx.constraintlayout.widget.ConstraintLayout

xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

android:layout\_width="match\_parent"

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

<EditText

android:id="@+id/editor"

android:layout\_width="0dp"

android:layout\_height="0dp"

android:textSize="18sp"

android:gravity="start"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintBottom\_toTopOf="@id/save\_text"

app:layout\_constraintTop\_toTopOf="parent" />

<Button

android:id="@+id/save\_text"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:onClick="saveText"

android:text="Сохранить"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintBottom\_toTopOf="@id/text"

app:layout\_constraintTop\_toBottomOf="@id/editor" />

<TextView

android:id="@+id/text"

android:layout\_width="0dp"

android:layout\_height="0dp"

android:gravity="start"

android:textSize="18sp"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintBottom\_toTopOf="@+id/open\_text"

app:layout\_constraintTop\_toBottomOf="@+id/save\_text" />

<Button

android:id="@+id/open\_text"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:onClick="openText"

android:text="Открыть"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintTop\_toBottomOf="@+id/text" />

</androidx.constraintlayout.widget.ConstraintLayout>

{2 код}

public class MainActivity extends AppCompatActivity {

private final static String FILE\_NAME = "content.txt";

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

} // сохранение файла

public void saveText(View view){

FileOutputStream fos = null;

try {

EditText textBox = findViewById(R.id.editor);

String text = textBox.getText().toString();

fos = openFileOutput(FILE\_NAME, MODE\_PRIVATE);

fos.write(text.getBytes());

Toast.makeText(this, "Файл сохранен", Toast.LENGTH\_SHORT).show(); }

catch(IOException ex) {

Toast.makeText(this, ex.getMessage(), Toast.LENGTH\_SHORT).show(); }

finally{

try{

if(fos!=null)

fos.close(); }

catch(IOException ex){

Toast.makeText(this, ex.getMessage(), Toast.LENGTH\_SHORT).show();

} } }

// открытие файла

public void openText(View view){

FileInputStream fin = null;

TextView textView = findViewById(R.id.text);

try {

fin = openFileInput(FILE\_NAME);

byte[] bytes = new byte[fin.available()];

fin.read(bytes);

String text = new String (bytes);

textView.setText(text); }

catch(IOException ex) {

Toast.makeText(this, ex.getMessage(), Toast.LENGTH\_SHORT).show();

}

finally{

try{

if(fin!=null)

fin.close();

}

catch(IOException ex){

Toast.makeText(this, ex.getMessage(), Toast.LENGTH\_SHORT).show();

} } }}

**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 БД. Примеры.**

**SQLite –** БД с открытым исходным кодом.Поддерживает SQL, транзакции и процедуры.Не требует много ресурсов (примерно 250 кБ памяти). Встраиваемая библитека. 3 основных типа данных: TEXT, INTEGER, REAL.Всё содержимое БД хранится в одном файле. **Типы данных: 1)** INTEGER: целое число, аналог типу int в java; 2) REAL: число с плавающей точкой, аналог float и double в java; 3) TEXT: набор символов, аналог String и char в java; 4) BLOB: массив бинарных данных, аналог типу int в java**. Особенности:** SQLite интегрирован во все Android устройства; не требуется установка или настройка БД; нужно определить только процедуры создания и обновления БД; выполняется обращение к файлам устройства; желательно выполнять запросы асинхронно (в отдельном потоке); файл БД хранится в каталоге приложения: DATA/data/ИМЯ\_ПРИЛОЖЕНИЯ/databases/ИМЯ\_ФАЙЛА

DATA/data/com.bstu.test/databases/test.db

**Класс SQLiteOpenHelper**. onCreate() – вызывается при первом создании БД; onUpgrade() – вызывается при модификации БД **Класс SQLiteDataBase.** Базовый класс для работы с БД SQLite. Предоставляет методы для выполнения запросов к БД, открытия, обновл., закрытия БД. insert(), update(), delete(). execSQL(). Запросы к БД выпол-ся через rawQuery(), query(), SQLiteQueryBuilder класс

**Класс Cursor**. Запрос возвращает объект типа Cursor. Cursor указывает на одну строку результата. getCount(). moveToFirst(), moveToNext(), isAfterLast(). Типизированные методы get\*() для извлечения данных (getLong(columnIndex), getString(columnIndex)). getColumnIndex(columnName). close()

**Класс** **SQLiteQueryBuilder.** Создает SQL-запросы.

Сами sql-выражения представлены классом **SQLiteStatement**, которые позволяют с помощью плейсхолдеров вставлять в выражения динамические данные.

Основную функциональность по работе с БД предоставляет пакет **android.database**. Функциональность для работы с SQLite находится в пакете **android.database.sqlite**.

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

**Класс SQLiteOpenHelper** – создание БД. Содержит два абстрактных метода: 1) **onCreate()** – вызывается при первом создании БД; 2) **onUpgrade(**) – вызывается при модификации БД.

В приложении создается класс, наследуемый от ***SQLiteOpenHelper***. В нем нужно реализовать вышеперечисленные методы, описав в них логику создания и модификации вашей базы. Здесь принято объявлять открытые строковые константы для названия таблиц и полей создаваемой БД, которые клиенты могут использовать для определения столбцов при выполнении запросов к БД. Тщательно документируйте тип данных каждого столбца, т. к. клиенту требуется эта информация для обращения к данным. Объявление констант для таблицы Contact:

*public static final String TAbLE\_NAME = "contact";*

*public static final String NAME "first\_name";*

*public static final String PHONE "phone";*

Класс, расширяющий SQLiteOpenHelper, неявно наследует интерфейс BaseColumns, в котором определена строковая константа \_id, представляющая имя поля для идентификаторов записей. В создаваемых таблицах БД тип поля \_id – *integer primary key autoincrement*. Описатель autoincrement является необязательным.

В методе обратного вызова **onCreate()** нужно реализовать логику создания таблиц и заполнить их начальными данными (если нужно):

*@Override*

*public void onCreate(SQLiteDatabase db){*

*db.execSQL("CREATE TAbLE + TAbLE\_NAME + (\_id INTEGER PRIMARY KEY AUTOINCREMENT, + COL\_NAME + TEXT, + COL\_PHONE + TEXT);");*

Метод **onUpdate()** вызывается при установке обновлений программы с измененной структурой таблиц. В методе обратного вызова onUpgrade() можно реализовать запрос в БД на уничтожение таблицы (drop table), а после вызвать метод **onCreate()** для создания версии таблицы с обновленной структурой:

*@Override*

*public void onUpgrade(SQLiteDatabase db,int oldVersion,int newVersion)*

*{db.execSQL("DROP TAbLE IF EXISTS + TAbLE\_NAME);onCreate(db);*

**Класс SQLiteDatabase** – управление БД SQLite. В немопределены методы для чтения, добавления, удаления, изменения данных. Для чтения данных используют вызов метода query():

*Cursor query(*

*String table,*

*String[] columns,*

*String selection,*

*String[] selectionargs,*

*String groupBy,*

*String having,*

*String sortOrder)*

В **метод query()** передают семь параметров:

1. **table** – имя таблицы, к которой передается запрос;

2. **columns** – список имен возвращаемых полей. При передаче null возвращаются все столбцы;

3. **selection** – параметр, формирующий выражение where (исключая сам оператор where). Значение null возвращает все строки;

4. **selectionArgs** – значения аргументов фильтра;

5. **groupBy** – параметр, формирующий выражение group ву (исключая сам оператор group ву). Если group ву не нужен, передается null;

6. **having**– параметр, формирующий выражение having (исключая сам оператор having). Если не нужен, передается null;

7. **sortOrder**– параметр, форматирующий выражение order ву (исключая сам оператор order ву). При сортировке по умолчанию передается null.

**19) Объект Cursor для запросов к объекту SQLite Database. Методы getCount(), moveToFirst(), get\* (columnIndex). Класс CursorAdapter и подкласс Simple CursorAdapter. Пар-ры констр-ра SimpleCursorAdapter.**

**Объект cursor** используется для перемещений назад или вперед по выборке данных. У экземпляров типа cursor есть встроенное понятие позиции, похожей на Java-интерфейс ***Iterator***. Чтобы получить требуемые записи из выборки, можно использовать несколько методов для позиционирования курсора:

* **moveToFirst()** – перемещает курсор в первую запись в выборке;
* **moveToLast()** – перемещает курсор в последн. запись в выборке;
* **moveToNext()** – перемещ. курсор в след. запись и определяет, существует ли эта запись. Возвращает true – курсор указывает на др. строку после перемещения, false – текущая запись была последней;
* **moveToPrevious()** – перемещает курсор в предыдущую запись;
* **moveToPosition()** – перемещает курсор в указанную позицию;
* **getPosition()** – возвращает текущий индекс позиции курсора.

Есть еще методы: **isAfterLast()** показывает, что курсор находится в конце массива данных; **moveToFirst()** переводит курсор на первый элемент массива данных; **getColumnIndex()** позволяет получить индекс столбца по его имени;

**Объект Cursor** обеспечивает доступ к записям результирующей выборки. Для обработки возвращаемых данных у него есть методы для чтения каждого типа данных – **getString(), getlnt(), getFloat(), getDouble().** Для вставки новой записи используется **метод insert()**:

*long insert (String table, String nullColumnHack, ContentValues values)*

* **table** – имя таблицы, в которую будет вставлена запись;
* **nullColumnHack**– в SQLite нельзя вставлять полностью пустую строку, и если полученная строка будет пустой, то этому столбцу явно будет назначено значение null;
* **values** – карта отображений (***класс Мар*** и его наследники), переда- ваемая клиентом, которая содержит пары ключ-значение. Ключи в карте – названия столбцов табл, значения – вставляемыми данными.

**insert()** возвращает идентиф-тор вставленной строки или -1(ошибка)

Для обновления – **метод update(),** удаления – **метод delete(),** возвращают число модифицированных или удаленных строк**:**

*int update (String table,ContentValues values, String where Clause, String[] whereArgs)*

*int delete (String table, String where Clause, String[] whereArgs)*

Класс SimpleCursorAdapter позволяет назначать данные курсора, используемые источником данных, для компонентов. Подкласс класса CursorAdapter, который был разработан для облегчения отображения столбцов класса Cursor непосредственно на компоненты TextView или ImagesView, определенные в XML-разметке. Конструктор класса получает следующие параметры:

* Context, в нем выполняется компонент ListView или ему подобный
* идентификатор ресурса разметки, который используется для отображения каждого элемента в ListView
* класс Cursor. Обеспечивает доступ к данным. Можно присвоить значение null, если класс Cursor определяется позже
* массив String, включающий отображаемые названия столбцов
* массив типа int. Включает идентификаторы ресурсов интерфейса
* параметр типа int - флаг

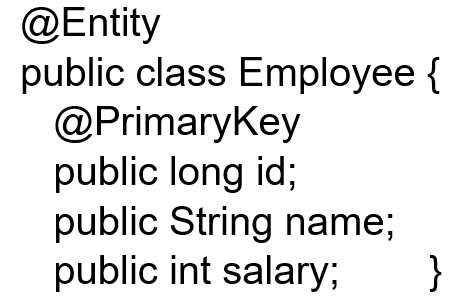
Для создания класса SimpleCursorAdapter нужно определить массивы, включающие имена столбцов, для отображения на компоненты GUI; ID ресурсов для компонентов GUI, которые отображают данные из именованных столбцов. В коде создается массив String, показывающий, что может отображаться лишь именованный столбец. Затем создается параллельный массив типа int, содержащий ID ресурсов для соответствующих компонентов GUI. Потом создается класс SimpleCursorAdapter.

Размеры массивов fromColumns и toViews должные совпадать.

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

**Room** – высокоуровневый интерфейс для низкоуровневых привязок SQLite, встроенных в Android, о которых вы можете узнать больше в документации. Выполняет большую часть своей работы во время компиляции, создавая API-интерфейс поверх встроенного SQLite API, поэтому не нужно работать с Cursor или ContentResolver. Room имеет три основных компонента: Entity, Dao и Database.

Аннотацией **Entity** необходимо пометить объект, который мы хотим хранить в БД. Для этого создадим класс Employee, который будет представлять собой данные сотрудника: id, имя, зарплата.

 Класс помечается аннотацией Entity. Объекты класса Employee будут использоваться при работе с БД. Employee также будет использован для создания таблицы в базе. В качестве имени таблицы будет использовано имя класса. А поля таблицы будут созданы в соответствии с полями класса. Аннотацией PrimaryKey мы помечаем поле, которое будет ключом в таблице. В объекте **Dao** будем описывать методы для работы с БД. Нужны методы для получения списка сотрудников и для добавления/ изменения/ удаления сотрудников. Опишем их в интерфейсе с аннотацией Dao.

*@Dao*

*public interface EmployeeDao {*

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

*List<Employee> getAll();*

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

*Employee getById(long id);*

*@Insert*

*void insert(Employee employee);*

*@Update*

*void update(Employee employee);*

*@Delete*

*void delete(Employee employee); }*

Методы getAll и getById позволяют получить полный список сотрудников или конкретного сотрудника по id. В аннотации Query нам необходимо прописать соответствующие SQL-запросы, которые будут использованы для получения данных. Для вставки/ обновления/ удаления используются методы insert/ update/ delete с соответствующими аннотациями. Тут никакие запросы указывать не нужно. Названия методов могут быть любыми. Главное - аннотации.

Аннотацией **Database** помечаем основной класс по работе с БД. Этот класс должен быть абстрактным и наследовать **RoomDatabase**.

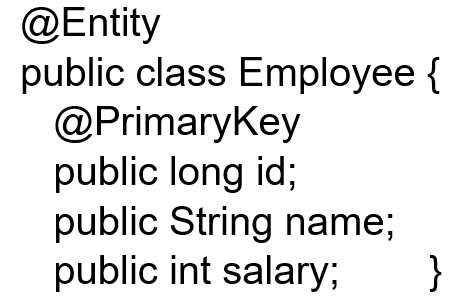
*@Database(entities = {Employee.class}, version = 1)*

*public abstract class AppDatabase extends RoomDatabase {*

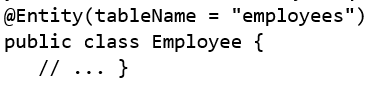
*public abstract EmployeeDao employeeDao(); }*

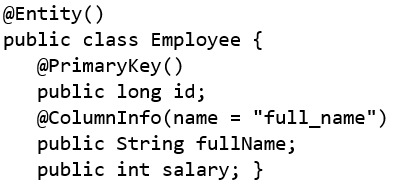
В параметрах аннотации Database указываем, какие Entity будут использоваться, и версию базы. Для каждого Entity класса из списка entities будет создана таблица. В Database классе необходимо описать абстрактные методы для получения Dao объектов, которые вам понадобятся.

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

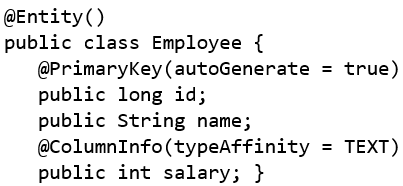
Аннотацией **Entity** необходимо пометить объект, который мы хотим хранить в БД. Для этого создадим класс Employee, который будет представлять собой данные сотрудника: id, имя, зарплата.

Класс помечается аннотацией Entity. Объекты класса Employee будут использоваться при работе с БД. Employee также будет использован для создания таблицы в базе. Будем получать их от базы при запросах данных и отправлять их в базу при вставке данных. В качестве имени таблицы будет использовано имя класса. А поля таблицы будут созданы в соответствии с полями класса. Аннотацией PrimaryKey мы помечаем поле, которое будет ключом в таблице. Entity класс используется для создания таблицы.

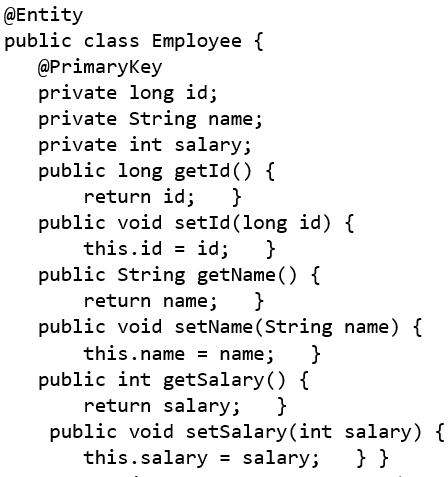
По умолчанию в качестве имени таблицы используется имя этого класса. Можно указать свое имя, используя пар-р tableName.



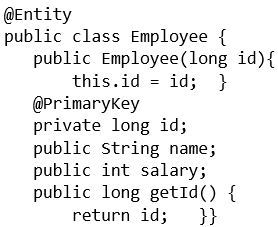
По умолчанию в качестве имени полей в таблице используются имена полей Entity класса. Но можно указать свое имя, используя параметр name в аннотации ColumnInfo.

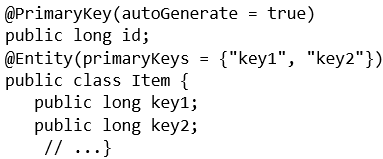


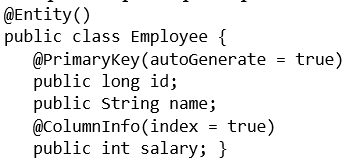
По умолчанию Room определяет тип данных для поля в таблице по типу данных поля в Entity классе. Можно явно указать свой тип.

***Модификаторы доступа.*** Чтобы Room мог добраться до полей класса Entity, мы делаем их public. Но есть возможность использовать private поля. Для этого надо добавить set/get методы.

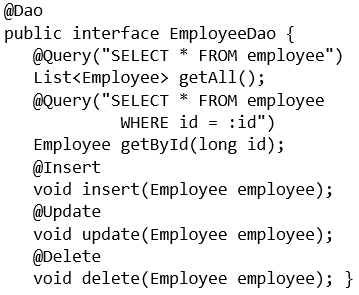
Все поля - private. Но каждое имеет set/get методы. Чтобы добавить методы: ALT+INSERT, пункт **Getter and Setter**, затем выбираем поля, для которых надо создать методы.

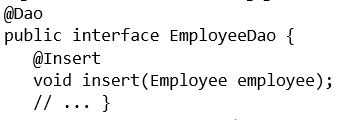
Вместо set-методов можно использовать конструктор. Поле id здесь - private и имеет get-метод. А вместо set-метода, Room будет использовать конструктор. Параметр конструктора должен иметь тот же тип и имя, что и поле Entity класса. Можно использовать конструктор для всех полей или только для некоторых.

***Первичный ключ.*** Параметр autoGenerate позволяет включить для поля режим autoincrement, в котором БД сама будет генерировать значение, если он не будет указан. Чтобы создать составной ключ, используем пар-р primaryKeys.

***Индекс.*** В аннотации Entity есть параметр indicies, который позволяет задавать индексы. Индекс дает возможность установить проверку на уникальность. Это делается параметром unique = true. *@Entity (indices = {@Index(value = {"first\_name", "last\_name"}, unique = true)})* Индекс для одного поля может быть настроен через параметр index аннотации ColumnInfo

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

***Dao.*** В объекте Dao описываются методы для работы с БД. Методы getAll и getById позволяют получить полный список сотрудников или конкретного сотрудника по id. В аннотации Query нужно прописать соответствующие SQL-запросы, которые используются для получения данных. Для вставки/ обновления/ удаления используются методы insert/ update/delete с соответствующ. аннотациями. Запросы указывать не нужно. Названия методов - любые,главное - аннотации.

***Insert.*** Аннотация Insert - это простой способ вставить объект в БД. В Dao интерфейсе описываем метод, который на вход принимает Entity объект. К методу добавляем аннотацию Insert и Room сгенерирует необходимый код в реализации этого интерфейса.

*Вставка нескольких объектов****.*** Можно передавать в метод не один, а несколько объектов, используя varargs

*Получение id.* Можно возвращать id только что добавленной записи. Для этого надо описать метод так, чтобы он возвращал long (*long insert(Employee employee);*). В случае добавления нескольких записей, необходимо использовать long[] или List<Long>

*Режимы вставки.* В режиме REPLACE старая запись будет заменена новой. Этот режим хорош, когда надо вставить запись, если ее нет в таблице или обновить запись, если она есть. В режиме IGNORE будет оставлена старая запись и операция вставки не будет выполнена.

***Update.*** Аналогична Insert, но не вставляет, а обновляет объекты в БД. @Update void update(Employee employee);

Так же можно использовать коллекции и varargs, чтобы обновлять несколько объектов сразу. Update ищет в бд запись по ключу. Если не найдет, то ничего не произойдет. Если найдет, то обновит все поля, а не только те, которые мы заполнили в Entity объекте. Можно получить количество обновленных записей. Для этого опишите метод так, чтобы он возвращал int (int update(List<Employee> employee);) Update поддерживает параметр onConflict.

***Delete.*** Методы с аннотацией Delete будут удалять объекты. Можно использовать коллекции и varargs, чтобы удалять несколько объектов сразу. @Delete void delete(Employee employee);

Delete ищет в бд запись по ключу. Можно получить количество удаленных записей. Для этого необходимо описать метод так, чтобы он возвращал int. (int delete(List<Employee> employee);)

***Transaction*** (Транзакции). Аннотация @Transaction позволяет выполнять несколько методов в рамках одной транзакции.

@Dao

public abstract class EmployeeCarDao {

@Insert

public abstract void insertEmployee(Employee employee);

@Insert

public abstract void insertCar(Car car);

@Transaction

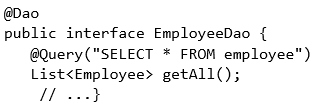
public void insertCarAndEmployee(Car car, Employee employee) {

insertCar(car);

insertEmployee(employee); } }

EmployeeCarDao - отдельный Dao объект для работы с комбинацией Car и Employee. В нем описываются методы для вставки объектов по отдельности, а затем оба этих метода вызываем в одном методе с аннотацией Transaction. В итоге вставятся либо оба объекта, либо, в случае возникновения ошибки, ни один из них. В этом случае Dao - не интерфейс, а абстрактный класс.

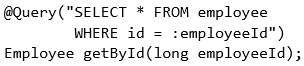
**23) Получение данных с помощью запросов Query библиотеки Room. List, массив, Cursor. Метод с аннотацией Query: параметры запроса, возвращаемые типы. LiveData, Subsets, Insert, Update и Delete. Примеры.**

***List, массив, Cursor.*** Чтобы запросить из базы Employee-объекты, нужно в Dao создать метод с аннотацией Query.

В Query прописывается запрос, который должен вернуть данные. В качестве возвращаемого типа указываем List<Employee>. При вызове этого метода, Room сделает запрос в таблицу employee, конвертирует полученные данные в Employee объекты и упакует их в List. Запрос в Query проверяется на правильность синтаксиса во время компиляции. Вместо List, можно использовать массив или Cursor, если это необходимо.

***LiveData.*** Room умеет возвращать данные в LiveData обертке.

Использование LiveData имеет огромное преимущество перед использование списка или массива. При добавлении новых, удалении старых или обновлении текущих данных в таблице employee, Room снова выполнит Query запрос, и мы получим в onChanged методе актуальные данные с учетом последних изменений.

***Передача параметров.*** В Query можно передавать параметры, чтобы сделать запросы более конкретными. Перед параметром employeeId в запросе должно стоять двоеточие. Room возьмет значение этого параметра из метода и подставит его в запрос.

***Subsets.*** Часто при запросе данных нужно получить из таблицы не все поля, а только некоторые. Такие запросы быстрее и легче, чем тянуть все поля. В этом случае мы можем использовать отдельный объект - обычный класс. С помощью ColumnInfo мы настраиваем имена полей, чтобы они совпадали с полями таблицы. Используем этот класс в методе запроса.

@Query("SELECT first\_name, last\_name FROM employee")

List<Name> getNames();

В этих не Entity классах можно использовать вложенные классы с аннотацией @Embedded.

***Insert, Update и Delete запросы.*** Аннотации Insert, Update и Delete позволяют модифицировать данные, но их возможности слишком ограниченны. Часто возникает необходимость обновить только некоторые поля или удалить записи по определенному условию. Это можно сделать запросами с помощью Query.Обновление зарплат у сотрудников по списку id:

@Query("UPDATE employee SET salary = :newSalary WHERE id IN (:idList)")

int updateSalaryByIdList(List<Long> idList, int newSalary);

Опционально метод может возвращать int значение, в котором мы получим количество обновленных строк. Если это не нужно, то делайте метод void:

int updatedCount = db.employeeDao().updateSalaryByIdList(Arrays.asList(1L, 3L, 4L), 10000);

Удаление сотрудников по списку id

@Query("DELETE from employee WHERE id IN (:idList)")

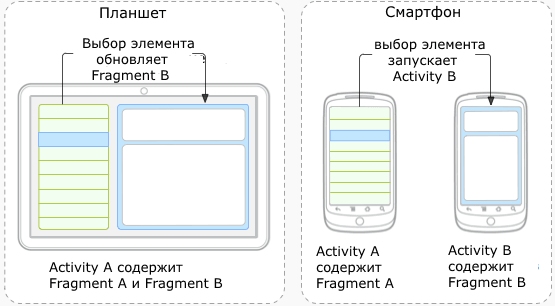
int deleteByIdList(List<Long> idList);

Запросы удаления также могут возвращать int значение, в котором мы получим количество удаленных строк:

int deletedCount = db.employeeDao().deleteByIdList(Arrays.asList(1L, 3L, 4L));

**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.**

Фрагмент представляет кусочек визуального интерфейса приложения, который может использоваться повторно и многократно. У фрагмента может быть собственный файл layout, есть свой собственный жизненный цикл. Фрагмент существует в контексте activity и имеет свой жизненный цикл, вне activity обособлено он существовать не может. Каждая activity может иметь несколько фрагментов.



Для начала работы с фрагментами создадим новый проект с пустой MainActivity. И вначале создадим первый фрагмент. Но сразу стоит отметить, что не вся функциональность фрагментов по умолчанию может быть доступна в проекте, поскольку располагается в отдельной библиотеке - AndroidX Fragment library. И вначале необходимо подключить к проекту эту библиотеку в файле build.gradle. Найдем в нем секцию dependencies. В ее начало добавим строку implementation "androidx.fragment:fragment:1.3.6"

И затем нажмем на появившуюся ссылку Sync Now.

Фактически фрагмент - это обычный класс Java, который наследуется от класса Fragment. Однако как и класс Activity, фрагмент может использовать xml-файлы layout для определения графического интерфейса. И таким образом, мы можем добавить по отдельности класс Java, который представляет фрагмент, и файл xml для хранения в нем разметки интерфейса, который будет использовать фрагмент.

***Добавление фрагмента в Activity***

Для использования фрагмента добавим его в MainActivity. Для этого изменим файл activity\_main.xml, которая определяет интерфейс для MainActivity:

<?xml version="1.0" encoding="utf-8"?>

<androidx.fragment.app.FragmentContainerView

xmlns:android="http://schemas.android.com/apk/res/android"

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:name="com.example.fragmentapp.ContentFragment" />

Для добавления фрамента применяется элемент FragmentContainerView. По сути FragmentContainerView представляет объект View, который расширяет класс FrameLayout и предназначен специально для работы с фрагментами. Собственно кроме фрагментов он больше ничего содержать не может.

Его атрибут android:name указывает на имя класса фрагмента, который будет использоваться. В моем случае полное имя класса фрагмента с учетов пакета com.example.fragmentapp.ContentFragment.

Код класса MainActivity остается тем же, что и при создании проекта:

package com.example.fragmentapp;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

}

Стоит отметить, что Android Studio представляет готовый шаблон для добавления фрагмента. Собственно воспользуемся этим способом.

Для этого нажмем на папку, где находится класс MainActivity, правой кнопкой мыши и в появившемся меню выберем New -> Fragment -> Fragment(Blank)

***Добавление логики к фрагменту***

Фрагмент определяет кнопку. Теперь добавим к этой кнопки некоторое действие. Для этого изменим класс ContentFragment:

package com.example.fragmentapp;

import android.os.Bundle;

import android.view.View;

import android.widget.Button;

import android.widget.TextView;

import androidx.annotation.NonNull;

import androidx.annotation.Nullable;

import androidx.fragment.app.Fragment;

import java.util.Date;

public class ContentFragment extends Fragment {

public ContentFragment(){

super(R.layout.fragment\_content);

}

@Override

public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {

super.onViewCreated(view, savedInstanceState);

Button updateButton = view.findViewById(R.id.updateButton);

TextView updateBox = view.findViewById(R.id.dateTextView);

updateButton.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

String curDate = new Date().toString();

updateBox.setText(curDate);

}

});

}

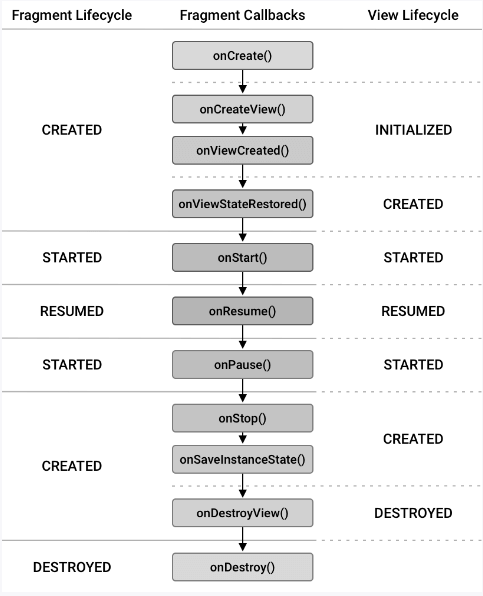
}

Здесь переопределен метод onViewCreated класса Fragment, который вызывается после создания объекта View для визуального интерфейса, который представляет данный фрагмент. Созданный объект View передается в качестве первого параметра. И далее мы можем получить конкретные элементы управления в рамках этого объекта View, в частности, TextView и Button, и выполнить с ними некоторые действия. В данном случае в обработчике нажатия кнопки в текстовом поле выводится текущая дата.

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

***Жизненный цикл фрагментов***

Каждый класс фрагмента наследуется от базового класса Fragment и имеет свой жизненный цикл, состоящий из ряда этапов:



Каждый этап жизненного цикла описывается одной из констант перечисления Lifecycle.State:

* INITIALIZED
* CREATED
* STARTED
* RESUMED
* DESTROYED

Стоит отметить, что представление фрагмента (его визуальный интерфейс или View) имеет отдельный жизненный цикл.

* При создании фрагмент находится в состоянии INITIALIZED. Чтобы фрагмент прошел все остальные этапы жизненного цикла, фрагмент необходимо передать в объект FragmentManager, который далее определяет состояние фрагмента и переводит фрагмент из одного состояния в другое.
* Когда фрагмент добавляется в FragmentManager и прикрепляется к определенному классу Activity, у фрагмента вызывается метод onAttach(). Данный метод вызывается до всех остальных методов жизненного цикла. После этого фрагмент переходит в состояние CREATED
* onCreate(): происходит создание фрагмента. В этом методе мы можем получить ранее сохраненное состояние фрагмента через параметр метода Bundle savedInstanceState. (Если фрагмент создается первый раз, то этот объект имеет значение null) Этот метод вызывается после вызова соответствующего метода onCreate() у activity.

public void onCreate(Bundle savedInstanceState)

* onCreateView(): фрагмент создает представление (View или визуальный интерфейс). В этом методе мы можем установить, какой именно изуальный интерфейс будет использовать фрагмент. При выполнении этого метода представление фрагмента переходит в состояние INITIALIZED. А сам фрагмент все еще находится в состоянии CREATED

public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)

* Первый параметр - объект LayoutInflater позволяет получить содержимое ресурса layout и передать его во фрагмент.
* Второй параметр - объект ViewGroup представляет контейнер, в которой будет загружаться фрагмент.
* Третий параметр - объект Bundle представляет состояние фрагмента. (Если фрагмент загружается первый раз, то равен null)
* На выходе метод возвращает созданное с помощью LayoutInflater представление в виде объекта View - собственно представление фрагмента
* onViewCreated(): вызывается после создания представления фрагмента. public void onViewCreated (View view, Bundle savedInstanceState)
* Первый параметр - объект View - представление фрагмента, которое было создано посредством метода onCreateView.
* Второй параметр - объект Bundle представляет состояние фрагмента. (Если фрагмент загружается первый раз, то равен null)
* onViewStateRestored(): получает состояние представления фрагмента. После выполнения этого метода представление фрагмента переходит в состояние CREATED

public void onViewStateRestored (Bundle savedInstanceState)

* onStart(): вызывается, когда фрагмент становится видимым и вместе с представлением переходит в состояние STARTED public void onStart ()
* onResume(): вызывается, когда фрагмент становится активным, и пользователь может с ним взаимодействовать. При этом фрагмент и его представление переходят в состояние RESUMED public void onResume ()
* onPause(): фрагмент продолжает оставаться видимым, но уже не активен и вместе с представлением переходит в состояние STARTED

public void onPause ()

* onStop(): фрагмент больше не является видимым и вместе с представлением переходит в состояние CREATED public void onStop ()

На этом этапе жизненного цикла мы можем сохранить состояние фрагмента с помощью метода onSaveInstanceState(). Однако стоит учитывать, что вызов этого метода зависит от версии API. До API 28 onSaveInstanceState() вызывается до onStop(), а начиная API 28 после onStop().

* onDestroyView(): уничтожается представление фрагмента. Представление переходит в состояние DESTROYED
* onDestroy(): окончательно уничтожение фрагмента - он также переходит в состояние DESTROYED
* Метод onDetach() вызывается, когда фрагмент удаляется из FragmentManager и открепляется от класса Activity. Этот метод вызывается после всех остальных методов жизненного цикла.

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

Элемент RecyclerView предназначен для оптимизации работы со списками и во многом позволяет повысить производительность по сравнению со стандартным ListView.

Для представления данных добавим в проект в ту же папку, где расположен класс MainActivity, новый класс Java, который назовем State:

package com.example.listapp;

public class State {

private String name; // название

private String capital; // столица

private int flagResource; // ресурс флага

public State(String name, String capital, int flag){

this.name=name;

this.capital=capital;

this.flagResource=flag;

}

public String getName() {

return this.name;

}

public void setName(String name) {

this.name = name;

}

public String getCapital() {

return this.capital;

}

public void setCapital(String capital) {

this.capital = capital;

}

public int getFlagResource() {

return this.flagResource;

}

public void setFlagResource(int flagResource) {

this.flagResource = flagResource;

}}

Класс State содержит поля для хранения названия и столицы страны, а также ссылку на ресурс изображения флага страны. В данном случае предполагается, что в папке res/drawable будут располагаться файлы изображений флагов для используемых стран.

Допустим, мы хотим вывести список объектов State с помощью RecyclerView. Для этого добавим в папку res/layout новый файл list\_item.xml:

<?xml version="1.0" encoding="utf-8"?>

<androidx.constraintlayout.widget.ConstraintLayout

xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content">

<ImageView

android:id="@+id/flag"

android:layout\_width="70dp"

android:layout\_height="50dp"

android:layout\_marginTop="16dp"

android:layout\_marginBottom="16dp"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toLeftOf="@+id/name"

app:layout\_constraintTop\_toTopOf="parent"

app:layout\_constraintBottom\_toBottomOf="parent" />

<TextView

android:id="@+id/name"

android:layout\_width="0dp"

android:layout\_height="wrap\_content"

android:layout\_marginLeft="16dp"

android:text="Название"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintLeft\_toRightOf="@+id/flag"

app:layout\_constraintTop\_toTopOf="parent"

app:layout\_constraintBottom\_toTopOf="@+id/capital" />

<TextView

android:id="@+id/capital"

android:layout\_width="0dp"

android:layout\_height="wrap\_content"

android:layout\_marginLeft="16dp"

android:text="Столица"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintLeft\_toRightOf="@+id/flag"

app:layout\_constraintTop\_toBottomOf="@+id/name"

app:layout\_constraintBottom\_toBottomOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>

Этот файл определяет разметку для вывода одного объекта State.

Как и в случае с ListView, для вывода сложных объектов в RecyclerView необходимо определить свой адаптер. Поэтому добавим в ту же папку, где расположен класс MainActivity и State, новый класс Java, который назовем StateAdapter:

package com.example.listapp;

import android.content.Context;

import android.view.LayoutInflater;

import android.view.View;

import android.view.ViewGroup;

import android.widget.ImageView;

import android.widget.TextView;

import androidx.recyclerview.widget.RecyclerView;

import java.util.List;

public class StateAdapter extends RecyclerView.Adapter<StateAdapter.ViewHolder>{

private final LayoutInflater inflater;

private final List<State> states;

StateAdapter(Context context, List<State> states) {

this.states = states;

this.inflater = LayoutInflater.from(context);

}

@Override

public StateAdapter.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {

View view = inflater.inflate(R.layout.list\_item, parent, false);

return new ViewHolder(view);

}

@Override

public void onBindViewHolder(StateAdapter.ViewHolder holder, int position) {

State state = states.get(position);

holder.flagView.setImageResource(state.getFlagResource());

holder.nameView.setText(state.getName());

holder.capitalView.setText(state.getCapital());

}

@Override

public int getItemCount() {

return states.size();

}

public static class ViewHolder extends RecyclerView.ViewHolder {

final ImageView flagView;

final TextView nameView, capitalView;

ViewHolder(View view){

super(view);

flagView = view.findViewById(R.id.flag);

nameView = view.findViewById(R.id.name);

capitalView = view.findViewById(R.id.capital);

}

}

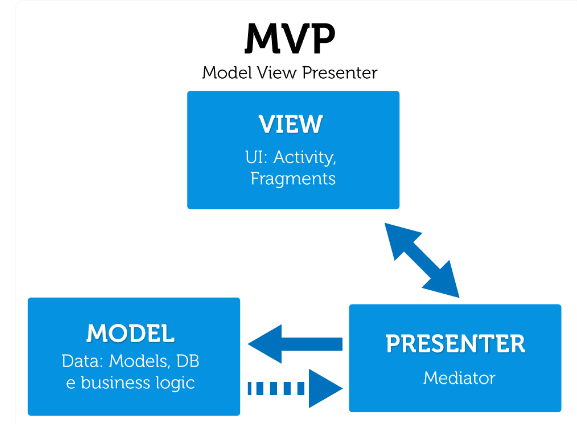
}

Адаптер, который используется в RecyclerView, должен наследоваться от абстрактного класса RecyclerView.Adapter. Этот класс определяет три метода:

* onCreateViewHolder: возвращает объект ViewHolder, который будет хранить данные по одному объекту State.
* onBindViewHolder: выполняет привязку объекта ViewHolder к объекту State по определенной позиции.
* getItemCount: возвращает количество объектов в списке

**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.**

Android не определяет зоны ответственности между компонентами приложения, поэтому вся логика работы с UI и данными приложения описана внутри одной Activity, что не позволяет сделать приложение расширяемым и легко тестируемым. Использование MVP позволяет решить эту проблему.



Есть много интересных подходов для реализации MVP, но не зависимо от выбранного решения должны сохранятся три компонента:

Presenter: Presenter выступает в качестве посредника между View и Model. Он извлекает данные из модели и передает их во View. Но в отличие от типичного MVC, он также решает, что нужно делать, когда вы взаимодействуете с View.

View :View, как правило, реализуется в Activity, которая содержит ссылку на презентер. Единственное, что делает View, это вызывает методы презентера при каком-либо действии пользователя

Model :Model рассматривается в качестве поставщика данных, которые будут отображаться во View.

Основные плюсы MVP:

* Код разбивается на мелкие, независимые кусочки
* Упрощение написания тестов к коду
* Более легкое внесение изменений в код.

Минусы: - Кода становится больше

**MOXY**

Moxy это библиотека, которая помогает использовать MVP шаблон при Android разработке.

Она имеет несколько ключевых особенностей:

- Presenter остается в живых, когда Activity пересоздается (это упрощает работу с многопоточностью).

- Автоматически восстанавливается все, что видел пользователь когда Activity пересоздавалась

- Возможность изменения многих Views из одного Presenter

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. По факту это значит, что вы практически не программируете во View. Для того, чтоб ваш Presenter не превратился в God Object, нужно разделять каждый отдельный блок бизнес-логики в отдельный Presenter. В таком случае у вас получится много Presenter, но они будут очень простыми и понятными.

***Moxy – MvpView и MvpViewState***

View - самый простой компонент MVP. Нужно завести интерфейс, который наследуется от интерфейса-маркера MvpView и описать в нём методы, которые будет уметь выполнять View. ViewState является наследником MvpViewState<View extends MvpView>. Он управляет одним, или несколькими, View. И каждый раз, когда во ViewState приходит команда из Presenter, ViewState отправляет её всем View, о которых он знает.

Стоит заметить, что MvpViewState хранит в себе список всех привязанных к нему View. И будет хорошо, если вы не будете забывать отвязывать View, которые уже уничтожены. В MvpViewState хранятся не прямые ссылки на View, а WeakReference, что всё-таки поможет GC.

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

***Список сенсоров. Сенсор света.***

MainActivity.java:

package ru.startandroid.develop.p1371sensors;

import java.util.List;

import android.app.Activity;

import android.hardware.Sensor;

import android.hardware.SensorEvent;

import android.hardware.SensorEventListener;

import android.hardware.SensorManager;

import android.os.Bundle;

import android.view.View;

import android.widget.TextView;

public class MainActivity extends Activity {

TextView tvText;

SensorManager sensorManager;

List<Sensor> sensors;

Sensor sensorLight;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

tvText = (TextView) findViewById(R.id.tvText);

sensorManager = (SensorManager) getSystemService(SENSOR\_SERVICE);

sensors = sensorManager.getSensorList(Sensor.TYPE\_ALL);

sensorLight = sensorManager.getDefaultSensor(Sensor.TYPE\_LIGHT);

}

public void onClickSensList(View v) {

sensorManager.unregisterListener(listenerLight, sensorLight);

StringBuilder sb = new StringBuilder();

for (Sensor sensor : sensors) {

sb.append("name = ").append(sensor.getName())

.append(", type = ").append(sensor.getType())

.append("\nvendor = ").append(sensor.getVendor())

.append(" ,version = ").append(sensor.getVersion())

.append("\nmax = ").append(sensor.getMaximumRange())

.append(", resolution = ").append(sensor.getResolution())

.append("\n--------------------------------------\n");

}

tvText.setText(sb); }

public void onClickSensLight(View v) {

sensorManager.registerListener(listenerLight, sensorLight,

SensorManager.SENSOR\_DELAY\_NORMAL);

}

@Override

protected void onPause() {

super.onPause();

sensorManager.unregisterListener(listenerLight, sensorLight);

}

SensorEventListener listenerLight = new SensorEventListener() {

@Override

public void onAccuracyChanged(Sensor sensor, int accuracy) {

}

@Override

public void onSensorChanged(SensorEvent event) {

tvText.setText(String.valueOf(event.values[0]));

}

};

}

В onCreate получаем SensorManager. У него запрашиваем полный список сенсоров, используя метод getSensorList и передавая туда тип сенсора TYPE\_ALL.

Чтобы получить конкретный сенсор (Sensor), вызываем метод getDefaultSensor. Передаем тип TYPE\_LIGHT и получаем сенсор света. Тут аккуратнее, т.к. если такого сенсора в девайсе нет – то метод вернет null.

В onClickSensList мы отписываем слушателя от сенсора. Об этом чуть позднее.

Далее берем список сенсоров и выводим по ним инфу на экран:

getName – имя

getType – тип

getVendor – создатель

getVersion – версия

getMaximumRange – максимальное значение, которое может вернуть сенсор

getResolution – насколько я понял, это минимальный шаг, с которым может изменяться значение

В методе onClickSensLight мы используем метод registerListener, чтобы на ранее полученный сенсор света (sensorLight) повесить своего слушателя listenerLight. Третий параметр метода – скорость получения новых данных. Т.е. насколько часто вам необходимо получать данные от сенсора. Есть 4 скорости в порядке убывания: SENSOR\_DELAY\_NORMAL, SENSOR\_DELAY\_UI, SENSOR\_DELAY\_GAME, SENSOR\_DELAY\_FASTEST.

Правда хелп пишет, что система может проигнорить это значение и выдавать данные как ей удобно. А начиная с API Level 9 можно вместо константы скорости передавать свое значение в микросекундах. Не перепутайте с миллисекундами.

В onPause мы отписываем своего слушателя от сенсора света. Тут, как обычно, рекомендуется отписываться как только данные вам не нужны, чтобы не расходовать зря батарею.

listenerLight – слушатель, реализует интерфейс SensorEventListener. У него два метода:

onAccuracyChanged – вызывается, когда меняется точность данных сенсора и в начале получения данных. Дает нам объект-сенсор и уровень точности:

SENSOR\_STATUS\_ACCURACY\_HIGH – максимально возможная точность

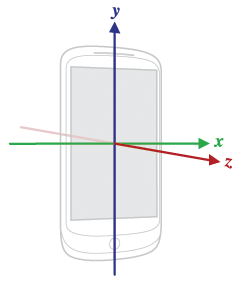
SENSOR\_STATUS\_ACCURACY\_MEDIUM – средняя точность, калибровка могла бы улучшить результат

SENSOR\_STATUS\_ACCURACY\_LOW – низкая точность, необходима калибровка

SENSOR\_STATUS\_UNRELIABLE – данные сенсора совсем ни о чем. Либо нужна калибровка, либо невозможно чтение данных.

onSensorChanged – здесь то мы и получаем данные от сенсора в объекте SensorEvent.

***Ускорение***

Далее рассмотрим сенсоры движения. Для этого нам надо будет понять, что в нашем трехмерном пространстве у устройства есть три оси.

Т.е. если держать устройство перед собой, то ось X проходит слева направо, ось Y проходит снизу вверх, ось Z проходит сквозь устройство в вашем направлении. Сенсор ускорения вернет нам массив из трех значений, каждое из которых соответствует определенной оси.

***Ориентация***

Теперь попробуем использовать данные сенсора ускорения и добавим к ним данные сенсора магнитного поля. Эти два набора данных при определенных манипуляциях дадут нам углы наклона устройства. Угла будет три, по одному для каждой оси.

За работу с сенсорами отвечает класс ***SensorManager***, содержащий несколько констант, которые характеризуют различные аспекты системы датчиков Android, в том числе:

*Тип датчика:* Ориентация, акселерометр, свет, магнитное поле, близость, температура и т.д.

*Частота измерений:* Максимальная, для игр, обычная, для пользовательского интерфейса. Когда приложение запрашивает конкретное значение частоты отсчётов, с точки зрения сенсорной подсистемы это лишь рекомендация. Никакой гарантии, что измерения будут производиться с указанной частотой, нет.

*Точность:*Высокая, низкая, средняя, ненадёжные данные.

Устройство может включать в себя несколько реализаций одного и того же типа датчиков. Чтобы найти реализацию, используемую по умолчанию, вызовите метод ***getDefaultSensor()*** из объекта ***SensorManager***, передавая ему в качестве параметра тип датчика в виде одной из констант, описанных выше.

У класса ***SensorManager*** есть метод ***getSensorList()***, позволяющий получить список доступных датчиков на устройстве через константу Sensor.TYPE\_ALL и метод getName():

Также вам понадобится интерфейс ***android.hardware.SensorListener***. Интерфейс реализован с помощью класса, который используется для ввода значений датчиков по мере их изменения в режиме реального времени. Приложение реализует этот интерфейс для мониторинга одного или нескольких имеющихся аппаратных датчиков.

Интерфейс включает в себя два необходимых метода:

* Метод ***onSensorChanged***(int sensor, float values[]) вызывается всякий раз, когда изменяется значение датчика. Этот метод вызывается только для датчиков, контролируемых данным приложением. В число аргументов метода входит целое, которое указывает, что значение датчика изменилось, и массив значений с плавающей запятой, отражающих собственно значение датчика. Некоторые датчики выдают только одно значение данных, тогда как другие предоставляют три значения с плавающей запятой. Датчики ориентации и акселерометр дают по три значения данных каждый.
* Метод ***onAccuracyChanged***(int sensor,int accuracy) вызывается при изменении точности показаний датчика. Аргументами служат два целых числа: одно указывает датчик, а другое соответствует новому значению точности этого датчика.

**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..**

Android устройства могут предоставить нам данные по нашему текущему местоположению. Это, конечно, очень удобно и вовсю используется для, например, пользования картой, получения актуальной для вашей местности информации (прогноз погоды), всевозможных чекинов и пр.

Реализация этого всего вполне проста. Мы вешаем слушателя на провайдера и получаем данные. На данный момент есть два провайдера: GPS и Network.

GPS – это данные с GPS-спутников.

Network – это координаты, которые можно получить через сотовую связь или WiFi. Для этого провайдера нужен инет.

MainActivity.java:

package ru.startandroid.develop.p1381location;

import java.util.Date;

import android.app.Activity;

import android.content.Intent;

import android.location.Location;

import android.location.LocationListener;

import android.location.LocationManager;

import android.os.Bundle;

import android.view.View;

import android.widget.TextView;

public class MainActivity extends Activity {

  TextView tvEnabledGPS;

  TextView tvStatusGPS;

  TextView tvLocationGPS;

  TextView tvEnabledNet;

  TextView tvStatusNet;

  TextView tvLocationNet;

  private LocationManager locationManager;

  StringBuilder sbGPS = new StringBuilder();

  StringBuilder sbNet = new StringBuilder();

  @Override

  protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);

    tvEnabledGPS = (TextView) findViewById(R.id.tvEnabledGPS);

    tvStatusGPS = (TextView) findViewById(R.id.tvStatusGPS);

    tvLocationGPS = (TextView) findViewById(R.id.tvLocationGPS);

    tvEnabledNet = (TextView) findViewById(R.id.tvEnabledNet);

    tvStatusNet = (TextView) findViewById(R.id.tvStatusNet);

    tvLocationNet = (TextView) findViewById(R.id.tvLocationNet);

    locationManager = (LocationManager) getSystemService(LOCATION\_SERVICE);

  }

  @Override

  protected void onResume() {

    super.onResume();

    locationManager.requestLocationUpdates(LocationManager.GPS\_PROVIDER,

        1000 \* 10, 10, locationListener);

    locationManager.requestLocationUpdates(

        LocationManager.NETWORK\_PROVIDER, 1000 \* 10, 10,

        locationListener);

    checkEnabled();

  }

  @Override

  protected void onPause() {

    super.onPause();

    locationManager.removeUpdates(locationListener);

  }

  private LocationListener locationListener = new LocationListener() {

    @Override

    public void onLocationChanged(Location location) {

      showLocation(location);

    }

    @Override

    public void onProviderDisabled(String provider) {

      checkEnabled();

    }

    @Override

    public void onProviderEnabled(String provider) {

      checkEnabled();

      showLocation(locationManager.getLastKnownLocation(provider));

    }

    @Override

    public void onStatusChanged(String provider, int status, Bundle extras) {

      if (provider.equals(LocationManager.GPS\_PROVIDER)) {

        tvStatusGPS.setText("Status: " + String.valueOf(status));

      } else if (provider.equals(LocationManager.NETWORK\_PROVIDER)) {

        tvStatusNet.setText("Status: " + String.valueOf(status));

      }

    }

  };

  private void showLocation(Location location) {

    if (location == null)

      return;

    if (location.getProvider().equals(LocationManager.GPS\_PROVIDER)) {

      tvLocationGPS.setText(formatLocation(location));

    } else if (location.getProvider().equals(

        LocationManager.NETWORK\_PROVIDER)) {

      tvLocationNet.setText(formatLocation(location));

    }

  }

  private String formatLocation(Location location) {

    if (location == null)

      return "";

    return String.format(

        "Coordinates: lat = %1$.4f, lon = %2$.4f, time = %3$tF %3$tT",

        location.getLatitude(), location.getLongitude(), new Date(

            location.getTime()));

  }

  private void checkEnabled() {

    tvEnabledGPS.setText("Enabled: "

        + locationManager

            .isProviderEnabled(LocationManager.GPS\_PROVIDER));

    tvEnabledNet.setText("Enabled: "

        + locationManager

            .isProviderEnabled(LocationManager.NETWORK\_PROVIDER));

  }

  public void onClickLocationSettings(View view) {

    startActivity(new Intent(

        android.provider.Settings.ACTION\_LOCATION\_SOURCE\_SETTINGS));

  };

}

В onCreate определяем TextView-компоненты и получаем ***LocationManager***, через который и будем работать.

В onResume вешаем слушателя с помощью метода ***requestLocationUpdates***. На вход передаем:

- тип провайдера: **GPS\_PROVIDER** или **NETWORK\_PROVIDER**

- минимальное время (в миллисекундах) между получением данных. Я укажу здесь 10 секунд, мне этого вполне хватит. Если хотите получать координаты без задержек – передавайте 0. Но учитывайте, что это только минимальное время. Реальное ожидание может быть дольше.

- минимальное расстояние (в метрах). Т.е. если ваше местоположение изменилось на указанное кол-во метров, то вам придут новые координаты.

- слушатель, объект ***locationListener***, который рассмотрим ниже

Также здесь обновляем на экране инфу о включенности провайдеров.

В onPause отключаем слушателя методом removeUpdates.

***locationListener*** – слушатель, реализует интерфейс **LocationListener** с методами:

* ***onLocationChanged*** – новые данные о местоположении, объект Location. Здесь мы вызываем свой метод ***showLocation***, который на экране отобразит данные о местоположении.
* ***onProviderDisabled*** – указанный провайдер был отключен юзером. В этом методе вызываем свой метод ***checkEnabled***, который на экране обновит текущие статусы провайдеров.
* ***onProviderEnabled*** – указанный провайдер был включен юзером. Тут также вызываем ***checkEnabled***. Далее методом ***getLastKnownLocation*** (он может вернуть null) запрашиваем последнее доступное местоположение от включенного провайдера и отображаем его. Оно может быть вполне актуальным, если вы до этого использовали какое-либо приложение с определением местоположения.
* ***onStatusChanged*** – изменился статус указанного провайдера. В поле status могут быть значения ***OUT\_OF\_SERVICE*** (данные будут недоступны долгое время), ***TEMPORARILY\_UNAVAILABLE*** (данные временно недоступны), ***AVAILABLE*** (все ок, данные доступны). В этом методе мы просто выводим новый статус на экран.

Провайдеры включаются и отключаются в настройках системы. Тем самым, просто определяется доступен ли провайдер для получения от него координат. Чуть позже увидим, как можно отправить юзера в эти настройки. Программное включение/выключение провайдеров через стандартные методы недоступно.

Далее идут свои методы.

* showLocation на вход берет Location, определяет его провайдера методом getProvider и отображает координаты в соответствующем текстовом поле.
* formatLocation на вход берет Location, читает из него данные и форматирует из них строку. Какие данные он берет: getLatitude – широта, getLongitude – долгота, getTime – время определения.
* checkEnabled определяет включены или выключены провайдеры методом isProviderEnabled и отображает эту инфу на экране.
* Метод onClickLocationSettings срабатывает по нажатию кнопки Location settings и открывает настройки, чтобы пользователь мог включить или выключить провайдер. Для этого используется Intent с action = ACTION\_LOCATION\_SOURCE\_SETTINGS.

Осталось в манифесте прописать разрешение на определение координат - ACCESS\_FINE\_LOCATION, которое позволит нам использовать и Network и GPS. Также существует разрешение ACCESS\_COARSE\_LOCATION, но оно дает доступ только к Network-провайдеру.

**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.**

***API ключ***

1)Начнем с первого пункта. Для этого нам надо зайти в Google APIsConsole.

* Если у вас нет проекта, гугл предложит его создать.
* Жмем Create project…
* Проект создан. Слева выбираем Services и открывается список служб.
* В этом списке ищем Google Maps Android API v2
* Жмем на тумблер включения (OFF), принимаем соглашение и снова попадаем в список служб.
* Но теперь служба должна быть включена
* Все ок. Теперь идем за ключом.
* Для этого слева сверху выбираем API Access и открывается такой экран.
* На нем жмем кнопку Create new Android key и от нас требуют ввести SHA1-значение ключа, которым будет подписано приложение.
* И даже дают инструкции, как это сделать с помощью утилиты keytool
* Если вообще непонятно о чем речь, рекомендую прочесть Урок 123. Там я все подробно описал.
* Но последние версии Eclipse позволяют получить SHA1 без всяких команд, достаточно зайти в настройки и там есть вся инфа
* Берем содержимое поля SHA1 fingerprint
* К нему через точку с запятой дописываем пакет приложения (в этом уроке это будет ru.startandroid.develop.p1391googlemaps), вставляем в поле и жмем Create.
* Появился нужный нам ключ - строка API key.
* Не закрывайте пока эту страницу, скоро этот созданный ключ нам понадобится при создании приложения.
* Не забывайте проделать эти же манипуляции, когда будете подписывать приложение для публикации на маркете другим ключом.
* Google Play services

2)Теперь второй пункт. Для работы с картой нам понадобится Google Play services.

* Открываем SDK Manager и ищем там Extras > Google Play services (если планируете поддержку Android 2.2, то Google Play services for Froyo). Если еще не установлено, то устанавливайте.
* После того, как скачалось, у себя на компе ищите папку по следующему пути: <android-sdk>/extras/google/google\_play\_services/libproject/google-play-services\_lib/. Если вдруг не знаете, где лежит SDK, то это можно подсмотреть в настройках Eclipse, пункт Android, SDK Location:
* Исходники из папки google-play-services\_lib надо будет импортнуть в Workspace. Для этого в меню Eclipse жмем File > Import, выбираем Android > Existing Android Code into Workspace, находим google-play-services\_lib папку, ставим галку, чтобы импорт был выполнен путем копирования (Copy projects into workspace) и жмем Finish.
* Проект google-play-services\_lib должен появиться в вашем Workspace
* Предварительная подготовка завершена, можем создавать проект.

***Настройка проекта***

Теперь еще немного возни с проектом.

Заходим в свойства проекта и добавляем (кнопкой Add) ссылку на ранее импортированный google-play-services\_lib проект.

В манифесте необходимо добавить следующее в тег application:

<meta-data

android:name="com.google.android.maps.v2.API\_KEY"

android:value="AIzaSyComUhEqr9BL4JjqJE05Lck4j1uABIU08Y">

</meta-data>

<meta-data

android:name="com.google.android.gms.version"

android:value="@integer/google\_play\_services\_version">

</meta-data>

Первые данные, это наш ключ из гугл-консоли. Здесь вам надо в android:value поставить ваше значение API key, которое вы чуть ранее получили и оставили открытым в браузере. Этот ключ нужен, чтобы карта работала.

Вторые данные – это версия Google Play services. Думаю, что эта инфа о версии нужна, чтобы карта четко понимала, как и что ей доступно из этих сервисов.

Также, в манифесте в тег manifest нам надо добавить следующие разрешения:

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

<uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE"></uses-permission>

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"></uses-permission>

<uses-permission android:name="com.google.android.providers.gsf.permission.READ\_GSERVICES"></uses-permission>

Это доступ в инет, проверка доступности инета, сохранение кэша карт и доступ к гугл-веб-сервисам.

Если думаете работать с определением местоположения, то не забывайте про:

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

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

И туда же, в тег manifest такое требование:

<uses-feature

android:glEsVersion="0x00020000"

android:required="true">

</uses-feature>

Гугл-карты используют OpenGL ES версии 2. На девайсах, которые это не поддерживают, карта просто не отобразится. Поэтому ставим ограничение.

Теперь все. Далее продолжаем работу, как с обычным проектом.

В strings.xml добавим строки:

<string name="test">Test</string>

Экран main.xml:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<Button

android:id="@+id/btnTest"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:onClick="onClickTest"

android:text="@string/test">

</Button>

<fragment

android:id="@+id/map"

android:name="com.google.android.gms.maps.SupportMapFragment"

android:layout\_width="match\_parent"

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

</fragment> </LinearLayout>

MainActivity.java:

package ru.startandroid.develop.p1391googlemaps;

import android.os.Bundle;

import android.support.v4.app.FragmentActivity;

import android.view.View;

import com.google.android.gms.maps.GoogleMap;

import com.google.android.gms.maps.SupportMapFragment;

public class MainActivity extends FragmentActivity {

SupportMapFragment mapFragment;

GoogleMap map;

final String TAG = "myLogs";

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

mapFragment = (SupportMapFragment) getSupportFragmentManager()

.findFragmentById(R.id.map);

map = mapFragment.getMap();

if (map == null) {

finish();

return;

}

init();

}

private void init() {

}

public void onClickTest(View view) {

map.setMapType(GoogleMap.MAP\_TYPE\_SATELLITE);

}

}

В onCreate мы находим наш фрагмент с картой и получаем от него объект GoogleMap методом getMap. Учитывайте, что этот метод может вернуть null.

Кстати, в onCreate можно также повесить проверку, что на устройстве доступно Google Play services. Для этого надо вызвать метод isGooglePlayServicesAvailable() и он должен вернуть SUCCESS.

Метод init пока пустой.

В onClickTest мы устанавливаем тип карты методом setMapType

Всего существует 5 типов:

MAP\_TYPE\_NONE – карта не будет отображаться

MAP\_TYPE\_NORMAL – обычный режим, в нем карта стартует по умолчанию.

MAP\_TYPE\_SATELLITE – снимки со спутника

MAP\_TYPE\_TERRAIN – карта рельефа местности

MAP\_TYPE\_HYBRID – снимки со спутника + инфа о улицах и транспорте

Получить текущий тип можно методом getMapType.

***Управление***

Управление картой жестами и кнопками также поддается настройке. Для этого необходимо вызвать метод getUiSettings и мы получим объект настроек интерфейса UiSettings.

У него куча методов, которые что-либо включают/выключают:

setAllGesturesEnabled – все жесты

setCompassEnabled – компас (слева сверху)

setMyLocationButtonEnabled – кнопка текущего местоположения (необходимо включить определение, метод setMyLocationEnabled(true))

setRotateGesturesEnabled – жесты вращения

setScrollGesturesEnabled – жесты пролистывания карты

setTiltGesturesEnabled – жесты смена угла обзора

setZoomControlsEnabled – кнопки зума

setZoomGesturesEnabled – жесты зума

У этих методов есть аналоги, которые начинаются с is вместо set и позволяют получить текущие значения.

На карту можно вешать слушателей. Перепишем метод init:

private void init() {

map.setOnMapClickListener(new OnMapClickListener() {

@Override

public void onMapClick(LatLng latLng) {

Log.d(TAG, "onMapClick: " + latLng.latitude + "," + latLng.longitude);

}

});

map.setOnMapLongClickListener(new OnMapLongClickListener() {

@Override

public void onMapLongClick(LatLng latLng) {

Log.d(TAG, "onMapLongClick: " + latLng.latitude + "," + latLng.longitude);

}

});

map.setOnCameraChangeListener(new OnCameraChangeListener() {

@Override

public void onCameraChange(CameraPosition camera) {

Log.d(TAG, "onCameraChange: " + camera.target.latitude + "," + camera.target.longitude);

}

});

}

* setOnMapClickListener вешает слушателя OnMapClickListener. Его метод onMapClick сработает при клике на карту и вернет объект LatLng с координатами (latitude, longitude), где было нажатие.
* setOnMapLongClickListener вешает слушателя OnMapLongClickListener. Его метод onMapLongClick сработает при длительном нажатии на карту и вернет объект LatLng с координатами (latitude, longitude), где было нажатие.
* setOnCameraChangeListener вешает слушателя OnCameraChangeListener. Его метод onCameraChange сработает при смене позиции камеры, т.е. при пролистывании карты, и вернет объект CameraPosition, который содержит информацию о текущем положении камеры.

CameraPosition имеет атрибуты:

- target, тип LatLng с полями-координатами: latitude, longitude. Это точка, на которую смотрит камера.

- bearing, угол поворота камеры от севера по часовой

- tilt, угол наклона камеры

- zoom, текущий уровень зума