FIT2081 – Mobile Development

# Week 2 – Basic Components

* Minimum SDK version determines the lowest level of Android that your app will run on

## API fragmentation

* Fragment: a modular section of an activity 🡪 can be reused
* Relate to the updates of software
* Forward compatibility
  + Old apps running on new platform versions
  + Android apps are generally forward-compatible with new versions of the Android platform 🡪 most framework API are additive
* Backward compatibility:
  + New apps running on old platform
  + Can be solved by replacing existing API parts

## Android Components

Read more: <https://developer.android.com/guide/components/fundamentals.html>

### Activities

* Entry point for interacting with the user
* Represents a single screen with a user interface
* An activity facilitates the following key interactions between system and app:
  + Keeping track of what user currently cares about (what is on the screen)
  + Knowing the previously used processes contain things the user may return to (stopped activities)
  + Helping app handle having its process killed so the user can return to activities with their previous state restored
  + Providing a way for apps to implement user flows between each other & for the system to coordinate these flows
* The same activity can be started from different apps
* Activated by intents

### Services

* A general-purpose entry point for keeping an app **running** in the **background** for all kinds of reasons (do not have an user interface)
  + Playing music in the background
* 2 lifecycle:
  + Run until it finishes
  + Run as long as its bound-to process is still running
* 2 types of started service:
  + User is aware
  + User is not aware
* Activated by intents

### Content Providers

* Implement a mechanism to share data between applications
* Can store data in the file system
* Access to data is provided via a Universal Resource Identifier (URI)
* Data can be shared in the form of a file or an entire SQLite database
* **Content Resolver:**
  + The single, global instance in your application that provide access to your (other applications’) content providers
  + Includes: CRUD (create, delete, update and delete) methods
* Activate by request from content resolver

### Broadcast Receivers

* Component that enables the system to deliver events to the app outside of regular user flow
* Can deliver to apps that are not running
* Usually used as a system notification
* Activated by intents

### Related

#### Intent

* Used for activating 3 out of 4 components: service, activities and broadcast receivers

#### Manifest file

* XML file
* Includes:
  + A declaration of all components in the application
  + If a component is not declared the system can’t see it
* Primary task: informing the system about the app’s component
* Other tasks:
  + Identify any user permission the app requires
  + Declare minimum API level required by the app
  + Declare hardware and software features used or required by the app (camera, Bluetooth service)
  + Declare API libraries the app needs to be linked against (other than the Android framework API)

#### Resources

* Include strings, images, fonts, colours that appear in the user interface together with the XML representation of the user interface layouts
* By default, these files are stored in the /res
* Using resources make it easier to update various characteristics of the app without modifying the code and by providing sets of alternative resources – enables you to optimise the app

#### Context

* Interface to global information about an application environment
* When an application is compiled, a class named R is created that contains references to the application resources

# Week 3 – Life Cycle

# Week 4 – View & Layout

* Android UIs are made of a hierarchy of View objects

## View:

* + Basic building block for UI components
  + Occupies a rectangular area and responsible for drawing and event handling
  + Base class widget, used in creating interactive UI components (button, text fields, etc…)

## View group:

* + Base class for layout and view container, contains many views (children)
  + Define **ViewGroup.LayoutParams** class which serves as the base class for layout parameters
  + Allow views to be nested 🡪 represent as a hierarchy (HTML and XML alike)
  + The tree is call **Layout** (responsible for managing the size, position and behaviour of all the Views it contains)

## ViewGroup.LayoutParams:

* + Used by views to tell their parents how they want to be layout
  + Check [ViewGroup Layout Attribute](https://developer.android.com/reference/android/R.styleable.html#ViewGroup_Layout) for all the supported child attributes
  + Base LayoutParams class just describe how big the view wants to be for width and height
    - FILL\_PARENT (MATCH\_PARRENT in API 8 and above) 🡪 wants to be as big as parent minus paddings
    - WRAP\_CONTENT 🡪 wants to be big enough to enclose its content (plus padding)
    - An exact number
  + There are subclasses for this class that adds extra attributes

## Layout Parameters

* + XML layout attributes named layout\_something define layout parameters for the views that its ViewGroup contains

## Creating UIs

* 2 ways to declare a layout:
  + XML
  + **Instantiate** layout elements at runtime (Java)
* Java or XML:
  + XML:
    - enables you to better separate the presentation of your application from the code that control its behaviours
    - External so you can modify or adapt it without having to modify your source code and recompile
    - Easier to visualise the structure of your UI
  + Java:
    - if you really want this 🡪 refer to ViewGroup and View class references
    - check out JavaLayout app
* Personal opinion: Java is more complicated and confusing when dealing with layout 🡪 have to use parameters and stuff

## Layout types

### ViewGroup

* Has several Layout direct subclasses
  + CoordinatorLayout, FrameLayout, GridLayout, LinearLayout, RelativeLayout
* Has several Layout indirect classes
  + TableLayout
* ViewGroup is also a View 🡪 can be contained in another ViewGroup
  + Should aim for optimisation
  + Google introduced ConstraintLayout for optimisation sake

### View containers

* Many direct and indirect View container subclasses that can be contained in another view ( be part of a UI’s View hierarchy). E.g. Toolbar

### ConstraintLayout

* A new direct subclass found in the support library
  + Use SDK Manager to download the Android Support Repo 🡪 “ConstraintLayout for Android” and “Solver for ConstraintLayout”
  + Edit Gradle Scripts to include the necessary dependencies
* Compatible with API level 9
* Basically like a RelativeLayout but:
  + Views are attached to the layout sides or horizontal and vertical guidelines (virtual layout sides) and other Views by software analogues of springs
  + These springs can expand and collapse depending on the viewport of the device (including current orientation)
  + The tension can be determined by a percentage
  + Hard margins can be specified

## Styles

* A collection of attributes that specify the look and format for a view or window
* A style is defined in an XML resource that is separate from the XML that specifies the layout
* Style Inheritance:
  + Parent attribute in the <style> elements lets you specify a style from which your style should inherit attributes 🡪 use this to inherit attributes from an existing style and define only the attributes that you want to change or add
  + Similar to CSS

## Themes

* Style applied to an entire Activity or app [in the app’s Manifest file]
* Apply the Style to a set of attributes

## Material Design

* Comprehensive guide for visual, motion and interaction design across platforms and devices
* To use material design 🡪 follow guidelines defined in the material design specification and use the new components and functionality available in Android 5.0 (API level > 21 )
* [Tutorial](https://developer.android.com/guide/topics/ui/look-and-feel)
* Compatibility complications: cannot use Material theme with standard Activity super class (not able for maximum backward compatibility)
  + Works for Activities that are subclass of Activity
* Use v7 for backward compatibility to API 9 (2.3)
* Deeper MD:
  + [Overview](https://developer.android.com/design/material/index.html)
  + [Getting Started](https://developer.android.com/training/material/get-started.html)
  + [Training overview](https://developer.android.com/training/material/index.html)
  + [Customising MD theme](https://developer.android.com/training/material/theme.html)
  + [Compatibility](https://developer.android.com/training/material/compatibility.html)

## Action Bar vs AppBar

* App bar was formerly known as action bar, used for branding, navigation, search and actions
* No UI components, it is a design concept 🡪 more [info](https://developer.android.com/design/patterns/actionbar.html)
* Pre-lollipop: part of default theme (not a widget)
* Post-lollipop: no longer part of default theme 🡪 work more like a toolbar widget
* No UI component but there is a UI component called Toolbar and a confusing class named ActionBar 🡪 which addresses whatever is the AppBar/Action Bar for an Activity’s UI (could be implemented as an AppBar or Toolbar)
* android.support.v7.widget.Toolbar class
  + “A Toolbar is a generalization of action bars for use within application layouts. While an action bar is traditionally part of an Activity's opaque window decor controlled by the framework, a Toolbar may be placed at any arbitrary level of nesting within a view hierarchy. An application may choose to designate a Toolbar as the action bar for an Activity using the setSupportActionBar() method.”
* setSupportActionBar(Toolbar toolbar) Method
  + When set to a non-null value the getActionBar() method will return an ActionBar object that can be used to control the given toolbar as if it were a traditional window decor action bar
  + In addition to the Toolbar’s methods

### Using an ActionBar as your App Bar (not preferred)

* Use a theme which does not contain “.NoActionBar”
* The Action bar will appear without any coding but use getSupportActionBar() to gain a reference to it so it can manipulated
* There are API level differences as features were incrementally added

### Using a Toolbar as your App Bar (preferred)

* Use a theme which DOES contain “.NoActionBar” so there is no ActionBar
  + Style item elements in a style’s XML definition
* No API differences
* Use an android.support.v7.widget.Toolbar instead and use setSupportActionBar (Toolbar toolbar) to set the Toolbar to act as the Action Bar for this Activity window.

### Relevant videos:

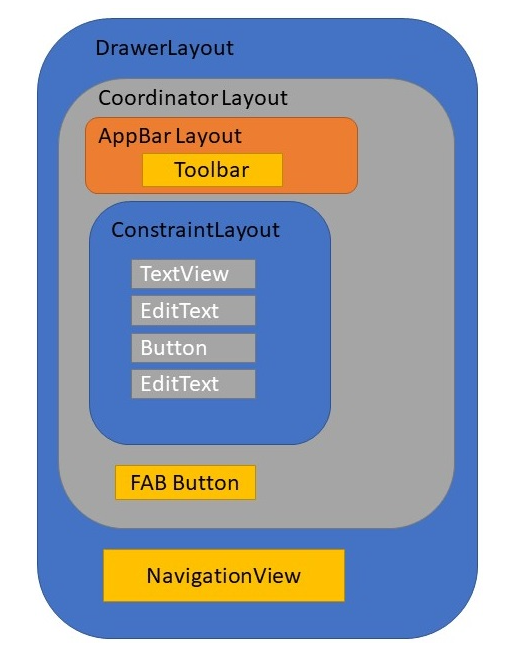
* + [Toolbar: How do they work?](https://www.youtube.com/watch?annotation_id=annotation_2860510657&feature=iv&index=5&list=PLWz5rJ2EKKc-lJo_RGGXL2Psr8vVCTWjM&src_vid=THadGrPeSJM&v=kmUGLURRPkI)
  + [AppBarLayout and scrolling gestures](https://www.youtube.com/watch?v=THadGrPeSJM&ab_channel=AndroidDevelopers)

## Tute notes

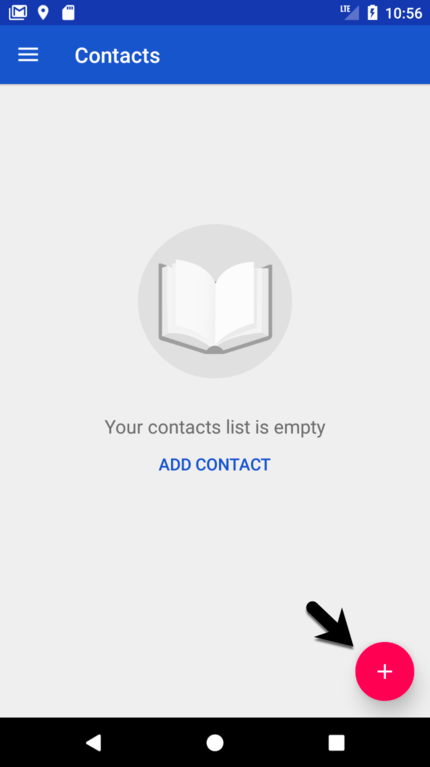
* 3 steps to use a broadcast receiver (receiving messages)
  + Declare permission (manifest file)
  + Register BR w Android OS (manifest file)
  + Tell BR what to listen on (channel/ frequency) 🡪 broadcast

# Week 5 – Advanced UI Design

## Drawer Layout



## Floating Action Button (FAB)

* 
* Android Studio’s Basic Activity Template includes one
* FAB is used to perform a **single, obvious, most common action**
* Material Design Principles: [here](https://material.io/components/buttons-floating-action-button#anatomy)

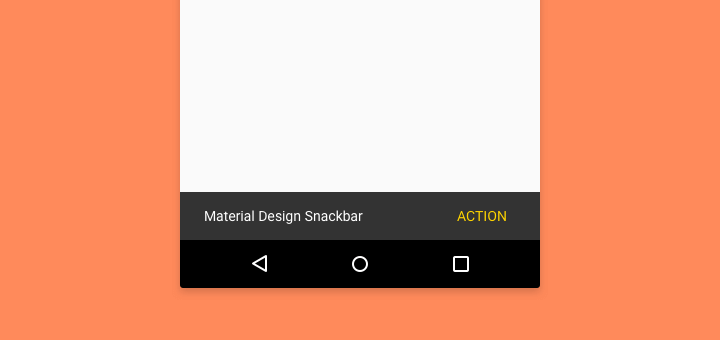
### Customising FAB looks:

* + Use material design guidelines to design appropriately
  + Use Android Studio 🡪 Tools 🡪 Android 🡪 Theme Editor to customise colours. By default, the FAB background colour (backgroundTint) is taken from the theme’s “colorAccent” unless overridden.
  + Use its srcCompat property to customise the button’s image

### Responding to a FAB being clicked:

* + get a reference to the component
  + Assign an on click listener object (type required is View.OnClickListener interface type)
  + Code an event handler in the listener object’s class (onClick required by View.OnClickListener interface)

## Snackbar

* 
* Android Studio’s Basic Activity Template includes one
* A panel that appears at the bottom of the screen containing a message and an optional action button. The panel can be made to disappear after some time or require the user to swipe it away.
* Contrast to Toast: toasts always disappear by themselves and do not include action buttons
* Note the behaviour of the FAB when the SnackBar appears.

### Making a Snackbar:

* + The first parameter of the Snackbar.make(…) method is a starting point for the SnackBar to walk up the UI’s View tree looking for a CoordinatorLayout to be its parent. This will allow the CoordinatorLayout to lift the FAB up out of the way when the SnackBar is shown for instance.
  + The Snackbar.make(…) returns a SnackBar object. The Snackbar’s setAction() method is invoked on this object to specify both the text on its action button and the listener object that contains the event handler for when the action button is clicked.
  + The Snackbar’s setAction() method returns the Snackbar object it was invoked on. This returned Snackbar object is then finally displayed using the Snackbar’s show() method.
  + Example:

Snackbar.*make*(view, "Item added to list", Snackbar.*LENGTH\_LONG*).setAction("Undo", undoOnClickListener).show();

## Layout

* activity\_main:
  + displayed by Launch Activity (MainActivity)
  + includes content\_main
    - Nesting layouts in this way simplifies individual layouts allowing, in this case, separation of app “décor” such as CoordinatorLayout, AppBarLayout and FAB from actual app content, in this case a ListView
  + CoordinatorLayout coordinates interaction between its child views including some default interactions and some customised actions as expressed by certain child view properties
    - e.g. FAB moves up to make way for a SnackBar’s appearance (a default interaction)
* content\_main:
  + Contains a ListView View container
* Exotica (layout element attributes):
  + Attributes beginning with “tools:” address the Layout editor and will have no effect on the actual running app.
  + Attribute values beginning with “?” refers to that attribute in the theme currently in effect.
* XML files of layout can be used: [here](https://developer.android.com/reference/android/R.layout)

## ListView

* The ListView component implements a View that shows items in a vertically scrolling list. The items come from the ListAdapter associated with this View
* The arrangement:
  + data (ArrayList of String in this case)🡨🡪 adapter 🡨🡪 display component (ListView in this case)

### Setting up ListView:

* + get a reference to the ListView
  + instantiate an adapter (specifying its data source, an ArrayList of String called listItems in this case)
  + plug the adapter into the ListView using the ListView’s setAdapter(…) method.

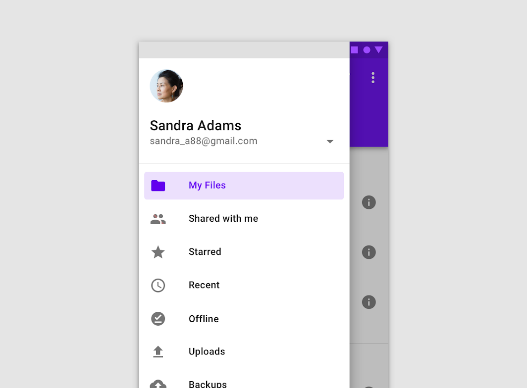
### Adding and removing listview:

* + Use the ListView’s add(itemToAdd) and remove(indexToRemove) methods.
  + In this case the item to remove is the last added i.e. the one at index array size – 1.
  + Use the adapter’s notifyDataSetChange() method to ensure the adapter’s connected display component is refreshed to allow for the add/remove.

## Navigation Drawer vs Options Menu

* “The options menu is the primary collection of menu items for an activity. It's where you should place actions that have a global impact on the app, such as "Search," "Compose email," and "Settings."”
* “The nav drawer spans the height of the screen, with everything behind it visible but darkened by a scrim.”
* “As per the Android Design guide, any drawers positioned to the left/start should always contain content for navigating around the application, whereas any drawers positioned to the right/end should always contain actions to take on the current content. This preserves the same navigation left, actions right structure present in the Action Bar and elsewhere.”
* Remember that Option menu items can appear as action buttons in the App bar. Navigation drawer menu items cannot do this.

## Navigation Drawer

* 
* Material Design: [here](https://material.io/components/navigation-drawer#usage)

### Setting up a drawer

* An instance of the DrawerLayout component in an Activity’s launch layout (drawer\_layout in activity\_main in this case)
  + It’s the launch Activitiy’s top View (even above any layout, it’s a ViewGroup but still surprising!!!)
  + It coordinates the visual interaction between the drawer and the UI without the drawer open
    - i.e. it doesn’t represent the Drawer itself. Possible clue as to why it’s the top View
  + Must be coded, it’s not in the component palette
* An instance of the NavigationView component (nav\_view in this case) as a child of the
  + Represents the drawer itself
  + Must be coded, it’s not in the component palette
  + Attributes point to a layout for the drawer’s header and an XML description of the drawer’s menu
  + In code a reference to the NavigationView is obtained and used to set a listener for menu item clicks
* A layout describing the header section of the drawer (nav\_header\_main in this case)
  + Optional. NavigationView’s headerLayout attribute value points at this
* A menu layout specifying the options to be displayed within the drawer (activity\_main\_drawer in this case)
  + NavigationView’s menu attribute value points at this
  + Same XML vocabulary as for Options menu layout.
* An ActionBarDrawerToggle instance
* A listener object containing event handling code to respond to user selection of each of the drawer’s menu items. This listener is assigned to the NavigationView in code using its setNavigationItemSelectedListener method in the launch Activity’s onCreate lifecycle callback
  + Not the same as Options menu where its Activity automatically listens for the menu to be opened by the user (onCreateOptionsMenu) and for an a menu item to be selected (onOptionsItemSelected)
    - Here we set the Activity instance as the designated Listener:
      * Make the application promise to implement the appropriate interface i.e. MainActivity implements NavigationView.OnNavigationItemSelectedListener
      * Code the interface’s only method onNavigationItemSelected in the Activity
      * Handle the possible menu item selections in a conditional control structure
* An ActionBarDrawerToggle instance
  + “This class provides a handy way to tie together the functionality of DrawerLayout and the framework ActionBar to implement the recommended design for navigation drawers.”
  + This includes
    - Inserting a hamburger icon in the AppBar that rotates on drawer open/close by whatever means (swipe, hamburger icon, code)
    - Actually opening and closing the drawer using the hamburger icon and synchronising the two
  + It’s set up by passing an instantiated instance as the only parameter to the DrawLayout’s addDrawerListener method in the launch Activity’s onCreate lifecycle callback
    - This should be followed by an immediate call to the instance’s syncState method for initial synchronisation of the drawer’s open state and its hamburger icon in the AppBar

### Opening and Closing the drawer

* To open and close the drawer from code (rather than user interaction) use the DrawerLayout’s closeDrawer(…) and openDrawer(…) methods.
* Should be done after every menu item selection when the appropriate response has executed. So, after the conditional control structure in the onNavigationItemSelected method
* In addition, if the drawer is open when the back button is clicked intercept the back button’s usual consumption and action by Android and instead make it just close the drawer. This can be coded in the Activity’s onBackPressed() method.

### Layout Analysis

* activity\_main (launch layout)
  + Contains DrawerLayout (coordinates draw and UI with draw closed)
  + Includes app\_bar\_main
    - Contains CoordinatorLayout coordinating:
      * AppBarLayout containing Toolbar
      * content\_main
        + which is just a TextView in this template version
      * FAB
        + Default template version
    - Contains NavigationView (represents the drawer itself)
      * Attribute value of which specifies nav\_header\_main
        + The drawer’s header layout
      * Attribute value of which specifies menu activity\_main\_drawer
        + A menu layout file (top element menu) that specifies the drawer’s menu

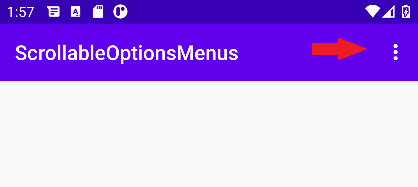
Same XML vocabulary as for Options menu

* main
  + Nothing to do with the drawer just the layout of Options menu (default template version)

## What about the primary content of the UI?

* “To use a DrawerLayout, position your primary content view as the first child with width and height of match\_parent and no layout\_gravity. Add drawers as child views after the main content view and set the layout\_gravity appropriately. Drawers commonly use match\_parent for height with a fixed width.”
* It helps to use an include for the primary content. This keeps the layout complexity down and allows you to focus on editing at a given layout in the View hierarchy.
* The drawer’s layout\_gravity should be “start” for draws opening from the left and “end” for drawers opening from the right. Google are trying to deprecate “left” and “right” for some reason and replace them with “start” and “end”. Same move in Java’s Swing libraries so must be a good reason.
* It helps to use an include for the primary content. This keeps the layout complexity down and allows you to focus on editing at a given layout in the View hierarchy.
* The drawer’s layout\_gravity should be “start” for draws opening from the left and “end” for drawers opening from the right. Google are trying to deprecate “left” and “right” for some reason and replace them with “start” and “end”. Same move in Java’s Swing libraries so must be a good reason.

## Options Menu

* 
* App Bar is Material Design’s name for the Action Bar. It’s a “dedicated piece of real estate at the top of each [Activity’s] screen [below the status bar] that is generally persistent throughout the app.”
* The Options menu is opened by clicking the 3 vertical dots at the extreme right hand end of the App Bar. It can contain action buttons. If there are too many action buttons or there is not enough room in the App Bar these actions buttons are pushed into the Overflow (aka Options) menu as menu items.
* **Making an Options Menu**:
  + XML or Code (XML is preferred for maintainability)
  + Create an XML layout resource in the res/menu project directory
* **Inflating:**
  + Override the Activity method onCreateOptionsMenu(menu)
  + Use the passed in menu reference (an Activity can only have at most 1 Options menu so there is no ambiguity here) to inflate the Options menu according to the specified menu resource (res/menu/menu\_fab\_example.xml in this case)
  + return true to display the menu
    - You will often see: return super.onCreateOptionsMenu(menu) which is probably more correct as the super gets to do any work it does and decide whether or not to display the menu
    - Just returning true seems to work just fine though
* **Responding to menu item selection in the options menu:**
  + Override the Activity method onOptionsItemSelected(MenuItem menu)
  + Extract the id of the menu item passed in (which is the menu item that was clicked)
  + Code a conditional control structure (if/if else/else or switch) to deal with different cases of the id (i.e. different menu item selections)
  + In each case return true to indicate you have consumed the event
    - This prevents the default action which involves calling the item's associated Runnable or sending a message to its Handler (these have to be set up but it’s rare to see this default approach used)
  + Note we defer to the super’s actions and its return value if we do not know what to do

## Scrollable layout

* To make your constraint layout scrollable, all you need is to wrap it by a ScrollView

# Week 6 – Recycler View & Alternate Resources

## RecyclerView

* A flexible viewgroup for providing a limited window into a large data set 🡪 allow information to be presented to the user in the form of a scrollable list (recycling the created views)
* More advanced and smoother than ListView
* It is a container for displaying large data sets that can be scrolled very efficiently by maintain a limited number of views
* Use the RecyclerView widget when data collections whose elements change at runtime based on user action or network events
* Can be dragged and dropped from palette
* RecyclerView needs a RecycleView.Adapter (create a class: RecyclerAdapter extends RecyclerView.Adapter<RecyclerAdapter.ViewHolder>) to supply its with data for its RecyclerView list items
* Adapter is class nested inside the RecyclerView 🡪 highly-specialised class that can only serve as an adapter to a RecyclerView

### RecyclerView’s Adapter

* For the adapter to do its job (bind app-specific data to views displayed within a RecyclerView) 3 adapter methods must be coded:
  + onCreateViewHolder: creates a ViewHolder instance for a list item
  + onBindViewHolder (input para is ViewHolder from onCreateViewHolder): bind a list item’s data to its ViewHolder instance
    - data
    - listener
  + getItemCount: internal API use

### ViewHolder

* Each RecyclerView list item is an instance of a subclass of the RecyclerView.ViewHolder class
* Using this class eliminates repeated use of the slow findViewById for each-and-every list item by the RecycleView (it produces smooth scrolling lists)
  + It saves addresses obtained by findViewById in a hash map then only uses findViewById again when the address is not found
* This class can be nested inside the adapter class (no other class will use it anw)
* When RecyclerView’s list (its LayoutManager to be precise) needs a new list item:
  + onCreateViewHolder is called and pass a reference to the ViewGroup that will contain the list item
  + onBindViewHolder to pass it the new ViewHolder instance and the position in the adapter’s data set of the data the ViewHolder must bind to

## Cardview

* Cardview extends the FrameLayout class and lets you show information inside cards that have a consistent look across the platform. It has shadow and rounded corners
* Can be used as the template for each item in a ListView or RecyclerView
* Can be dragged and dropped from palette

## JSON

* JSON is a lightweight data-interchange format
* Syntax rule:
  + Data is in name/value pairs
  + Data is separated by commas
  + Curly braces hold objects
  + Square brackets whole arrays
* Example (nested data included):

{

"firstname":"Tim",

"lastname":"John",

"websites":[

{

"description":"Company",

"url":"http://company.com",

"live":false

},

{

"description":"School",

"url":"http://school.com",

"live":true

}

]

}

* In order to save array of objects in shared preference (since SP only saves primitive data):
* Convert array of objects into a string using **Gson**

### Gson

* A Java library can be used to convert Java Object into JSON format and the other way around
* How to use
  + Before using, have to add dependency

dependencies {

implementation 'com.google.code.gson:gson:2.8.6'

}

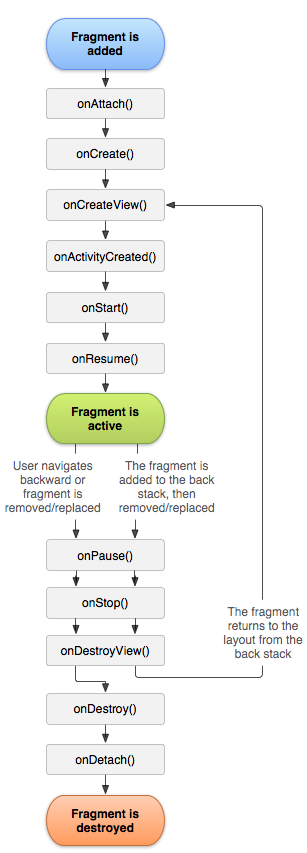
* + Create a new instance
    - Gson gson = new Gson();
  + Convert
    - String dbStr = gson.toJson(db)
  + Get data from Json
    - Db = gson.fromJson(dbStr,type)

## Fragments

* In order to have an activity with 2 tasks such that they have different layouts (XML), different logic and can be reused in other activities 🡪 **Fragments**
* Fragments’ characteristic:
  + Has its own lifecycle
  + Receive its own input events
  + Sorta like “subactivity”

### Fragments vs Activities

* Activity represents the full screen <> fragment is a portion of UI in an activity
* Activity contain 0 or multiple fragments
* Fragments can be reused in multiple activities
* A fragment can’t exist independently 🡪 always part of an activity
* A fragment can be added or removed while the activity is running



* Replace the current fragment with a new one:

getSupportFragmentManager().beginTransaction().replace(R.id.frag1,new Fragment1()).addToBackStack("f1").commit();

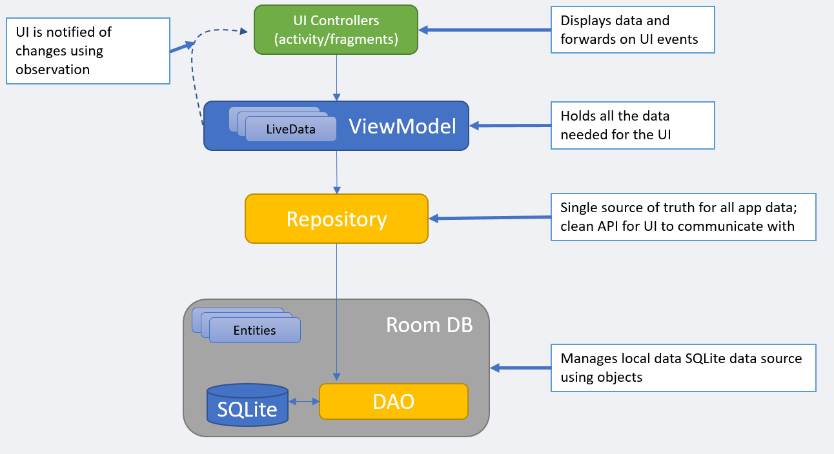
* + getSupportFragmentManager: returns the FragmentManager which is used to create transactions for adding, removing or replacing fragments.
  + beginTransaction: starts a series of edit operations on the Fragments
  + .replace: replaces the current fragment with a new fragment of type Fragment1 on the layout with id R.id.frag1
  + addToBackStack: Adds this transaction to the back stack. This means that the transaction will be remembered after it is committed, and will reverse its operation when later popped off the stack.

# Week 7 - Database

## Database

* Database is a collection of structured data
* Android uses SQLite as a database management system
* **SQL:** a language allows you to access and manipulate databases

## SQLite, Rooms, and LiveData



* **Data** **Access** **Objects** (DAO): main classes where you define your database interactions 🡪 include variety of query methods
* **Room** **DB**: the object provides the interface to the underlying SQLite
* **Repository**: class that contains all of the code necessary for directly handling all data sources used by the application. This avoids the need for the UI controller and ViewModel to contain code that directly access resources such as databases or web services
* **ViewModel**: The object provides the data for a specific UI component (fragment/activity) and contains data – handling business logic to communicate with the model 🡪 best place to implement your logic and pre-process data
* **LiveData**: a data holder that allows a value to become observable 🡪 an observable object has the ability to notify other objects when changes to its data occur thereby solving the problem of making sure that the UI always matches the data within the ViewModel

## Room Repository

* Repository manages local (SQLite) and remote data source

# Week 8 – Content Provider

* **Content provider:** a class that helps application to share their data with other application on the same device 🡪 implement it on top of the Room Database
* **How to access the CP of an application?** Access to the data is provided via a Universal Resource Identifier (URI) defined by the Content Provider
* **What is URL Authority?** A unique name that identifies the content provide. You cannot have two content providers with the same authority value on the same device
* **The CP Application’s methods:**

## onCreate() 🡪 initialise the provider:

@Override

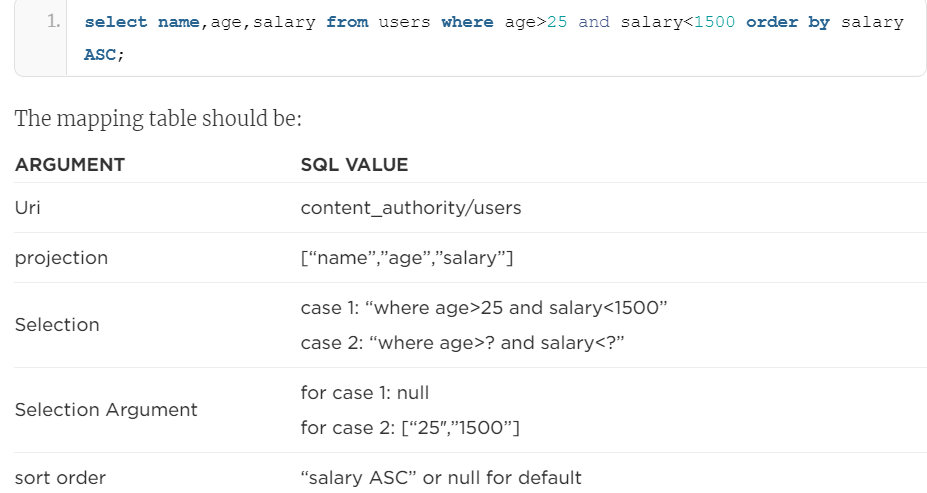
public boolean onCreate() {

db = TasksDatabase.getDB(getContext());

return true;

}

## query(Uri, String[], Bundle, CancellationSignal) 🡪 return data to the caller



public Cursor query(Uri uri, String[] projection, String selection,

String[] selectionArgs, String sortOrder) {

SQLiteQueryBuilder builder = new SQLiteQueryBuilder();

builder.setTables(Task.TABLE\_NAME);

String query = builder.buildQuery(projection, selection, null, null, sortOrder, null);

final Cursor cursor = db

.getOpenHelper()

.getReadableDatabase()

.query(query, selectionArgs);

return cursor;

}

* + cursor: an interface provides random read – write access to the result set returned by a database query 🡪 a data structure that holds **one or more rows** retrieved from a database. It contains methods to move the cursor to the next row, previous, first, or to the last row

## insert(Uri, ContentValue) 🡪 insert new data to the CP

public Uri insert(@NonNull Uri uri, @Nullable ContentValues contentValues) {

long rowId = db

.getOpenHelper()

.getWritableDatabase()

.insert(Task.TABLE\_NAME, 0, contentValues);

return ContentUris.withAppendedId(CONTENT\_URI, rowId);

}

* This method inserts a new row to the database and returns a new Uri with the inserted row ID.
* **Content Values**
  + Special data structure that is used to hold the data of **one row** only, used to send data to the database
  + Format: key-value pair format
* ContentValue vs Bundle: the keys in the content value are the table column’s names
* ContentValue vs Cursor: ContentValue only holds 1 row whereas Cursor might hold many

## update(Uri, ContentValue, String, String[]) 🡪 update the existing CP

* It accepts similar parameters as the insert method. It returns the number of rows that get affected by the update operation.

public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {

int updateCount;

updateCount = db

.getOpenHelper()

.getWritableDatabase()

.update(Task.TABLE\_NAME, 0, values, selection, selectionArgs);

return updateCount;

}

## delete(Uri, String, String[]) 🡪 delete data from the CP

* Delete one or more rows from the database 🡪 return the number of rows that is affected by the delete operation

@Override

public int delete(Uri uri, String selection, String[] selectionArgs) {

int deletionCount;

deletionCount = db

.getOpenHelper()

.getWritableDatabase()

.delete(Task.TABLE\_NAME, selection, selectionArgs);

return deletionCount;

}

## getType(Uri) 🡪 returns the MIME type of data in the CP

* Used to tell other application the type of the return of the given URI

## UriMatcher() 🡪 not method

* Used to create a new instance of the UriMatcher class. In this method, we give each URI a unique code that will be used later by the content provider’s methods to tell which table to use or to retrieve the requested ID.
* Example of using UriMatcher in delete():

public int delete(Uri uri, String selection, String[] selectionArgs) {

int uriType = sUriMatcher.match(uri);

int deletionCount = 0;

switch (uriType) {

case MULTIPLE\_ROWS\_TASKS: //no trailing row id so selection may indicate more than 1 row needs to be deleted if they can be found

deletionCount = db

.getOpenHelper()

.getWritableDatabase()

.delete(Task.TABLE\_NAME, selection, selectionArgs);

break;

case SINGLE\_ROW\_TASKS: //trailing row id, so just one row to be deleted if it can be found

String id = uri.getLastPathSegment();

String selectionId = Task.COLUMN\_ID + " = ?";

String [] selectionArgsId={String.valueOf(id)};

deletionCount = db

.getOpenHelper()

.getWritableDatabase()

.delete(Task.TABLE\_NAME, selectionId, selectionArgsId);

break;

default:

throw new IllegalArgumentException("Unknown URI: " + uri);

}

return deletionCount;

}

## How to access the content provider from other application?

* **Content Resolver** is used 🡪 used to access content provider
* Contains a set of objects that mirror those of the content provider (insert, query, delete, etc)
* Query table and show the number of rows

Uri uri= Uri.parse("content://fit2081.tasks.db.provider/tasks");

Cursor result= getContentResolver().query(uri,null,null,null);

tV.setText(result.getCount()+"");

* How to send a row to be saved in the database of other application (remember: insert method needs the data of the new row to be wrapped (inside) by ContentValue class)

public static final String COLUMN\_NAME = "taskName";

public static final String COLUMN\_DESCRIPTION = "taskDescription";

ContentValues values= new ContentValues();

values.put(COLUMN\_NAME,"New Task Name");

values.put(COLUMN\_DESCRIPTION,"New Task Desc");

Uri uri2= getContentResolver().insert(uri,values);

## Firebase

# Extra Notes

## Different layouts

### CoordinatorLayout

* [Link](https://developer.android.com/reference/android/support/design/widget/CoordinatorLayout.html)
* A layout provides an additional level of control over touch events between child views

### AppBarLayout

* [Link](https://developer.android.com/reference/android/support/design/widget/AppBarLayout.html)
* A vertical LinearLayout that is used to implement App bar concept such as scrolling gestures
* This view depends heavily on being used as a direct child within a Coordinator Layout. If you use AppBarLayout within a different ViewGroup, most of its functionality won’t work

### CollapsingToolbarLayout

* [Link](https://developer.android.com/reference/android/support/design/widget/CollapsingToolbarLayout.html)
* A wrapper for Toolbar which implements a collapsing App Bar. Designed to be used as a direct child of a AppBarLayout