FIT2081 Notes

Week 1: Terms and Introduction to Mobile Applications

Software Development Kit (**SDK**) – Bundle of all the software components to develop and deploy on a platform. We use Java SDK and Android SDK.

Application Programming Interface (**API**) – The class library by which we can call (execute) to perform common but complicated tasks. This gets constantly updated and thus, improves with increasing capabilities and updates by levels.

Integrated Development Environment (**IDE**) – Software environment that supplements all the tools developers need to develop applications. E.g. debuggers, version control, language sensitive code editors. We use Android Studio.

There are 3 different types of Mobile Apps which have different capabilities, advantages and disadvantages.

Native Apps

- App's compiled code are developed specifically for 1 type of OS e.g. iOS, Android
- Built using the SDK tools and languages provided by the platform vendor
    1. Android Studio using Java
    2. Swift with XCode IDE for iOS
- Apps are managed and downloaded via the app store (Play, Apple Store)

Mobile Web Apps

- Internet-enabled apps accessible via mobile device's browser.
- Users don't need to download or install the app to access it.
- Can be used for all device OS, written in HTML, CSS, and interaction in JS or Jquery etc.
- New technologies keep developing to create new possibilities for what a Mobile Web App can do

1. E.g. latest is *Service Workers*, which allow push notifications to a Mobile Web App, something previously only possible on a Native App.

Hybrid Apps

- Written using additional language and development environments other than the recommended languages for the platform.
- Still deployed as a Native App.
- Tries to get the pros of both Native and Mobile Apps without their cons

Comparisons between App Types

Native Apps PROS

- ✓ Access to all native API's
- ✓ Faster graphics performance
- ✓ AppStore distribution - security
- ✓ can be used offline, access anytime
- ✓ Can store info locally

Native Apps CONS

- ✓ Must be written for each platform so higher development + maintenance costs
- ✓ Tedious to get app approved on App store + costly

Mobile Web Apps PROS

- ✓ Cross-platform affinity
- ✓ Cheaper to develop + maintain
- ✓ Fast Centralized updates (don't need to wait for approval)

Mobile Web Apps CONS

- ✓ Limited Features as no access to native API
- ✓ Many variations between mobile browsers
- ✓ Cannot use offline
- ✓ No quality control of app = more unsafe

Hybrid Apps

1. Cross-platform affinity
2. Written with web technologies (HTML5, CSS3, JS)
3. Runs locally on the device, supports offline
4. Access to native API's
5. AppStore distribution

Summary

1. All 3 types are thriving in their own way
2. A poorly designed app is bad no matter how its implemented
3. Design requirements for a Mobile App is very different in comparison to a desktop App.

We stick with Native Apps in this unit because:

1. Free, rock solid IDE with good device emulations
2. Web Apps and Hybrid Apps are in a state of uncertainty as the technologies they depend upon rapidly evolve

We use the Android Mobile App Platform because:

1. Highest device count
2. Java is the most coded language in the world, considered an industry standard

App Basics

All our Activities must extend Activity or AppCompatActivity

AppCompatActivity provides backwards compatibility in our app.

List 3 tasks we must do in the **onCreate ()** callback of our activity

1. Call super.onCreate()
2. Inflate our layout in the layout file by calling: setContentView(R.layout.our_layout_file)
3. Retrieve any data inside the bundle (if any)

Note: We must first inflate our layout before referencing any widgets in our layout.

Exercise 1

Mobile Web App is a Website. What does that mean?

A Mobile Web App functions very similarly to a webpage as it is simply an application ran on a mobile device's browser. It contains the same technologies used to develop normal websites such as HTML5/CSS3 for design and logic interactions using Jquery and Javascript etc. Hence a Mobile Web App is essentially a website but made specifically to suit mobile device's browsers.

Week 2: Android Components, API Levels, Fragmentation, and Backward/Forward Compatibility

Android is an open source Linux-based OS designed for touchscreen mobile devices.

It is maintained by Google who has the largest Android App store called Google Play.

OS is a collection of software that manages computer hardware resources and provides common services for computer programs.

Android Versions

1. SDK Platform
    - Includes the version of the framework API
    - Incremented using n.n.n format
2. API Level
    - Integer value which uniquely identifies the API revision offered by a version of the Android platform.
3. API Codename
    - Given whenever there is a major platform release
    - Named after desserts and are incremented alphabetically

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	1.0%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.0%
4.1.x	Jelly Bean	16	4.0%
4.2.x		17	5.7%
4.3		18	1.6%
4.4	KitKat	19	21.9%
5.0	Lollipop	21	9.8%
5.1		22	23.1%
6.0	Marshmallow	23	30.7%
7.0	Nougat	24	0.9%
7.1		25	0.3%

Selecting the correct SDK version

You ideally want to target as many people as possible, however targeting low SDK levels has disadvantages such as lack of features.

Your choice of minimum SDK level should be a trade-off between the distribution of users to target, and the features your application will need.

Android Fragmentation – concerns with the alarming number of different Android OS versions in the market

Big problem as Android updates are slow to reach actual devices due to:

1. Customisation of Android for a device
2. Customisation of Android interface and apps by device manufacturers and Telcos
3. Reluctance by manufacturers to spend time and money customising Android for older devices which would allow users to delay buying new devices

Forward Compatibility – Old apps running on newer platform versions

Android applications are generally forward compatible with new versions of the Android platform, because almost all changes to the framework API are additive.

Rare case something goes wrong is when the application uses a part of the API that is later removed for some reason.

Backward Compatibility – New apps running on older platform versions

Android applications are not necessarily backward compatible with older versions of the Android platform as the App can utilize new framework APIs, which provide new platform capabilities, or replace existing API parts.

Android Support Library

Official Library providing a backward compatibility layer for earlier versions of Android devices. Main function is to provide some important newer APIs to run on older devices.

The Library also provides additional convenience classes and features not available in the standard Framework API for easier development and support across more devices.

Library has been extensively developed and is now considered an essential resource for app development.

Android App Components

1. **Activities** are the UI and logic of the app and defines a single window in the UI. There can be multiple activities in an app, and they can run together to bring a cohesive user experience.
   - ✓ Activities have Lifecycles which we'll see later in Week 3.
   - ✓ An activity can start another activity in the form of Intents (see later).

2. **Services** – processes that run in the background and do not have a user interface. Can be initialized and managed from Activities, broadcast receivers, or other services. Ideal for situations where an app needs to continue running tasks but does not need a visible UI, e.g. music player.

3. **Content Providers** – Class which implements a mechanism for the sharing of data between applications. An app can provide other app's access to its data through a Content Provider, with access provided via a URI (see Week 7-8). They essentially provide an abstraction from the app and its underlying data source.

4. **Content Resolver** – the single, global instance in your application that provides access to your (and other Apps) content providers. It accepts requests from clients and resolves these requests by directing them to the content provider with a distinct authority. Does this by storing a mapping from authorities to Content Providers.

5. **Broadcast Receivers** – the mechanism by which Apps can respond to Broadcast Intents. The receiver operates in the background and do not have a UI. A Broadcast Receiver must be registered by an application and configured with an Intent Filter to indicate the type of broadcast in which it is interested. When a matching intent is broadcasted, the receiver will be invoked by the Android runtime by which it will then have 5 seconds to complete any tasks required of it (e.g. launching a service, making data update, and issue notification) before returning.

6. **Intents** – Mechanism which implement the flow through activities, binding individual components to each other at runtime. Intents activate 3 of 4 component types – activities, services, and broadcast receivers.

Defining an Intent takes two parameters: the current Activity Context, and the target Activity.

Activity Intents – The mechanism by which one activity can launch another activity or service and implement the flow through the activities that make up an application; Can be defined:

1. **Explicitly** - request the launch of a specific activity by referencing the class name
2. **Implicitly** - stating the type of action to be performed or providing data of a specific type. This lets Android runtime choose an appropriate target Activity to activate.

Broadcast Intents – System wide intent that is sent out to all applications that have registered an "interested" Broadcast Receiver.

Service Intents – Same above but in form of background processes.

Activating Components – A unique aspect of Android system design is that any app can start another app's component through:

- **Inter App Activation** – This method activates **Activities, Services & Broadcast Receivers** through **Intents**. The app must deliver a message to the system that specifies their intent. An app must give its permission for any of its components to be activated by an external intent. When the system starts a component, it starts the process for that app (if it's not already running) and instantiates the classes needed for the component.

- **Content Providers** – **Activated when targeted by a request from a Content Resolver**. The resolver handles all direct transactions with the Content Provider so that the component that's performing transactions with the provider does not need to, and instead calls methods on the Content Resolver object. Creates a layer of abstraction between the content Provider and component requesting information (for security). *More details check week 7 and 8 content.*

Example of Activity Intent Coding (Week2App):

1. Activity Intent to take App into another Activity (ToLower Class) + sending a key/value pair of data into that Activity class:

```
public void toLowerAct(View view) {
    Intent intent = new Intent( packageContext: this,ToLower.class);
    EditText editText = findViewById(R.id.strLn);
    intent.putExtra( name: "key1",editText.getText().toString());
    startActivity(intent);
}
```

To retrieve our data in ToLower class, we call getIntent() then call intent.getStringExtra(key).

```
Intent intent = getIntent();
String myStr = intent.getStringExtra("key1");
myStr = myStr.toLowerCase();
```

Note: Always initiate the intent using StartActivity() in the end.

TextView and EditText

TextView is used to display text that is not editable by the user, by can be updated programmatically at any time.

EditText is used for user input.

Basic functions we will always utilize from TextView and EditText will be **.setText()** and **.getText().**

Layout ID's and string ID's

We can identify our components firstly in the layout, every component has a unique ID which we can reference, so for example, an EditText component can have id "strLn" which we can then access the user input by referencing that ID below.

```
EditText f1 = findViewById(R.id.strLn);
```

Furthermore, to have more consistent variables, we do not define literal values, but instead, use key constants, we define these in res/values/strings.xml. We can then reference these string key values by calling R.id.stringValue.

Example strings.xml file:

```
<resources>
    <string name="app_name">FIT2081 Lifecycles App</string>
    <string name="labelOfViewData">view data</string>
    <string name="labelOfNonViewData">non view data</string>
    <string name="labelOfPersistentData">persistent data</string>
    <string name="tute1pm">myKey</string>
    <string name="persDataKey">A</string>
    <string name="viewDataKey">B</string>
    <string name="nonViewDataKey">C</string>
    <string name="fileName">fit2081.dat</string>
```

Referencing our string values:

```
SharedPreferences sp2 = getSharedPreferences(getString(R.string.fileName), mode: 0);
SharedPreferences.Editor editor2 = sp2.edit();
editor2.putString(getString(R.string.persDataKey),persistentState);
editor2.putString(getString(R.string.viewDataKey),ViewState);
editor2.putString(getString(R.string.nonViewDataKey),nonViewState);
editor2.commit();
```

Note: Details about what **SharedPreferences** are is located in Week 3 Content.

View Hierarchy of an App

Detailed Notes in Week 4 Content.

App Manifest File – an XML file which details all an Apps component + capabilities.

- Includes a declaration of all components in the App to inform system
- Also identifies any user permissions the app requires e.g. read-access to something
- Declares the minimum API level required by the App.
- Declares hardware/software features used or required by the app e.g. camera
- Declares API libraries the app needs to be linked against, e.g. Google Maps library.

Application Resources

An Android application will contain a collection of resource files such as strings, images, fonts, color etc. These are all defined and stored under the **res** directory of the application. All the resource files (drawable, string, layout, styles etc.) in our app will be contain a resource ID which we can reference and access by calling R in Java, and by @resource in XML.

Using app resources makes it easy to update characteristics of our app without modifying code and – by providing sets of alternative resources, enables us to optimize our app for a variety of device configurations (e.g. screen size, language, orientation etc.)

Referencing in Java

R.layout.*, R.string.*, R.id.*, R.style.*, R.color.*, R.drawable.*.

Referencing in XML

@+id/.*/, @color/.*/, @layout/.*/, @style/.*/, @drawable/.*/.

Application Context

Context is an Interface to global information about an application environment. It is an abstract class whose implementation is provided by Android. It allows access to application-specific resources and classes. In our class, we reference this by '**this**'.

When an app is compiled, a class named **R** is created which contains references to the application resources. The app manifest file and these resources combine to create the **Application Context**. This **Context** is used to gain access to application resources at runtime, plus a wide range of methods to gather information.

Bundle, what is it?

In all our App's onCreate() callback, we see that there is an object of type **Bundle**. A Bundle is an object storing data as key/value pairs that is passed around Activities in Android to transmit information.

To store information into a Bundle, we store data similarly to how we do it in an **Intent**.

Instead of intent.putExtra(key, value) we call outState.putString(key, value) assuming our Bundle object is called "outState".

Then to retrieve our key/value, we call instate.getString(key).

We will see in Week 3 that **Bundle** is used to save/restore our app's **view state** and is used in 3 important callback methods: onCreate(), onSaveInstanceState(), onRestoreInstanceState().

Exercise 1

Consider an App compiled against API level 22 on a device running Nougat.

a. This usually works. Why?

Android applications are generally forward-compatible with new versions of the Android devices as almost all changes to the framework are additive.

b. How could this go wrong?

The App uses a part of an API that is later removed in the future for some reason.

c. Can it be fixed without changing the API level of the App and the device? Explain.

It cannot be fixed without changing the API level as the API is removed. The app must be updated to the newer API.

Consider an app compiled against API level 25 on a device running Lollipop.

d. How could this go wrong?

Android applications are generally not backwards compatible as newer versions of Android can include new framework APIs, which give apps new platform capabilities, or replace existing APIs.

e. If it goes wrong can it be fixed always or sometimes?

It sometimes can be fixed by extending the **AppCompat** Activity from the App's Main Activity in the Android Support Library.

f. Under what circumstances can it be fixed?

Through the Android Support Library, we can leverage the library to provide the backward compatibility layer across various devices supported.

Exercise 2

a. *Which App components can be activated by an Intent?*

Activities, Services, and Broadcast Receivers.

b. *Explain how each of these app components are targeted by an Intent?*

Activity – An Intent can explicitly target an Activity to activate, or implicitly let Android choose an appropriate Activity to activate.

Broadcast Receivers – Must be registered by an App and configured with an Intent filter to indicate the type of broadcast it wants. Broadcasts can then be targeted by a matching Intent, which gets invoked with the receiver having 5 seconds the complete the task required.

Service – Services receives Intent Data from the Android component and performs its work. They start and handles each Intent and stops itself when it runs out of work.

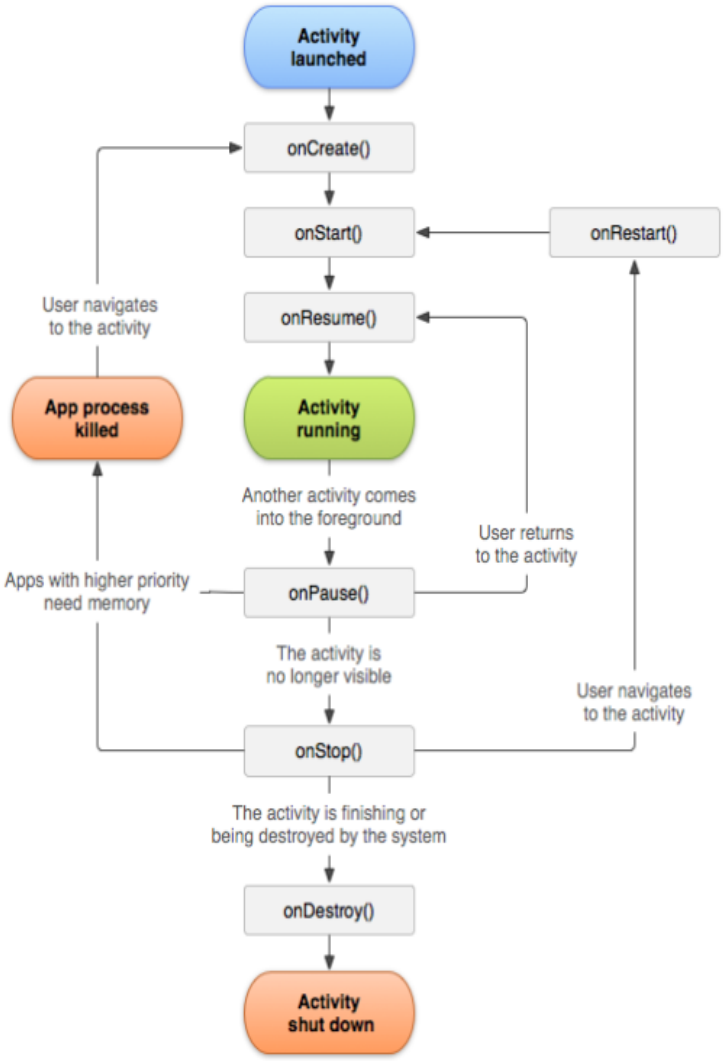
Exercise 3

a. *Name 5 things that can be defined using Android resources*
    1. String values
    2. Images
    3. Fonts
    4. Color
    5. UI layout
b. *Android can select between alternate resources depending on device configuration. Give 2 examples of resource selection*
    1. Screen size: (small, normal, large, xlarge)
    2. Screen orientation: (port, land)
c. *How are alternate resources distinguished?*
    1. Create a new directory like: **res/<resource_name>-<config_qualifier>.**
        i. <resource_name> is the directory name of the default resource
        ii. <qualifier> is the name that specifies a specific configuration for how the resource is to be used.

Week 3: Android Activity Lifecycles, Instance State, Persistent Data, Intents

Activities have 4 possible Lifecycle states which transition as a result of user interaction and OS process kills.

1. Foreground
2. Partially hidden
3. Fully hidden
4. Destroyed



The Activity class provides several callback's that allow the activity to know that a state has changed: that the system is creating, stopping, resuming, or destroying an activity.

Different events, some user-triggered, and some system-triggered, can cause an Activity to transition from one state to another.

Lifecycle Callback's – Within the lifecycle callback methods, we can declare how our activity behaves when the user leaves and re-enters the activity by:

- Overriding all Lifecycle callbacks to perform customised processing required of our app (make sure to call the super method! – super performs a lot of default standard processing required).
- e.g. onCreate() override begins with super.onCreate()
- onSaveInstanceState() & onRestoreInstanceState() are important Activity callback methods that plays a role in saving the **view state** of an Activity.

Types of Data in an Activity Instance State

Note: Not to be confused with Activity's Lifecycle State

Data can consist of:

1. Activity view data – the state of all Views in the Activity's layout
2. Activity non-view data – Activity instance variables
3. Persistent data – data associated with multiple uses of the Activity separated by any amount of time i.e. it spans an Activity's instances

Initial Case: Launching App for First Time

Every time an App Launches, it will always go through the initial lifecycle callback's:

1. onCreate()
2. onStart()
3. onResume()

There is nothing to restore, (remember that the restore callback only gets called if the bundle is not null, hence no need to call **onRestoreInstanceState** callback.

Common Case 1: Configuration change occurs - Many occurrences, such as device reorientation, input device, language changes.

For a configuration change, the Activity is destroyed and recreated. Android also automatically preserves the UI state during configuration change through:

1. **onSaveInstanceState(Bundle outState)** – Method saves the **View-state** from an activity before being killed so that the state can be restored in **onRestoreInstanceState(Bundle inState)**.

Note: savedInstanceState is a reference to a Bundle object that is passed into the onCreate() method of every Android Activities.

Activities can restore themselves to a previous state using the data stored in this bundle; if there is no instance data, then savedInstanceState defaults to null.

Note: onRestoreInstanceState(Bundle inState) only gets executed if inState != null.

Note: We define our own implementation of this method if we want to save additional **non-view data**. We still call the superclass default implementation as we want it to save the state of the **view data**.

```
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState); // call superclass to save view data
    outState.putString("myKey",nonViewState); // save non-view data variable
    Log.i(TAG,  msg: "onSaveInstanceState");
}
```

2. **onRestoreInstanceState(Bundle inState)** – called after onStart() when the activity is being re-initialized, data to restore is given in **inState**.

```
//only gets executed if inState != null so no need to check for null Bundle
protected void onRestoreInstanceState(Bundle inState) {
    super.onRestoreInstanceState(inState); // restore view data
    nonViewState = inState.getString("myKey"); // restore non-view variable nonViewState
    Log.i(TAG,  msg: "onRestoreInstanceState");
}
```

Same thing applies above, we create our own implementation of the method to restore **non-view data**, but still call default super method to restore **view data**.

So, the activity callback lifecycle for a configuration change will go like this:

1. onPause()
2. onSaveInstanceState(Bundle outState) – outState is the saved state to be placed into the Bundle object
3. onStop()
4. onDestroy()
5. onCreate()
6. onStart()
7. onRestoreInstanceState(Bundle savedInstanceState) – savedInstanceState is the data most recently supplied in onSaveInstanceState(Bundle).
   Only called when data in Bundle is **not null**.
8. onResume()

Common Case 2: User taps Back button – Tapping the **Back** button indicates to Android that the user is done with the Activity and so it will not save the view-state (not call **onSaveInstanceState**). Hence the Activity is **destroyed** and is also removed from the back stack.

To navigate back, we have to relaunch the activity, with nothing to restore as the state is lost.

So, the activity callback lifecycle will go like:

1. onPause()
2. onStop()
3. onDestroy()

Note: We could still code to restore our view-state by having a **sharedPreferences** file for saving and restoring data. We can call **saveSharedPreferences()** during onPause() and **restoreSharedPreferences()** during onStart().

```
protected void onStart() {
    restoreSharedPreferences();
    super.onStart();
    Log.i(TAG, msg: "onStart");
}

protected void onPause() {
    saveSharedPreferences();
    super.onPause();
    Log.i(TAG, msg: "onPause");
}
```

This allows us to still save our data despite user tapping “back” button.

SharedPreferences?

SharedPreferences Object files are Objects which points to a **non-volatile** (permanent) memory file (we can specify filename if needed) and stores data sets as key/value pairs. They are useful for storing small-scale information where it is unnecessary to use a database to implement.

Example Implementation See Below

Common Case 3: Activity or dialog appears in foreground

Partial Hide

If a new activity or dialog appears in the foreground, and **partially covers** the activity, the covered activity enters the paused state by calling onPause(). When the covered activity returns to the foreground and regains control, it calls onResume().

So, in this case, the Activity is **NOT destroyed**, and the **instance state** remains intact.

So, the activity callback lifecycle on a **partial hide** will go like:

1. onPause()
2. onResume()

Full Hide

Another case is if the Activity gets **fully covered** from another Activity, in this case, the Activity is **restarted**, and Android automatically saves the **view-state,** we can also manually code to also save our **non-view state** in the same **Bundle.**

So, the activity callback lifecycle on a full hide will go like:

1. onPause()
2. onSaveInstanceState()
3. onStop()
4. onRestart()
5. onStart()
6. onRestoreInstanceState()
7. onResume()

Back Stacks - A collection of inter-connected activities that users interact with.

The activities are arranged in a stack in order of activity opened. When user presses Back button, current activity is popped from the stack (destroyed), and previous activity resumes (onPause to onResume). Continually pressing back will eventually pop the stack until it reaches the Home screen.

Note: A back stack can include Activities from several different apps (remember an Activity can Intent an Activity in another App). There can also be multiple background back stacks.

Summary of Types of Lifecycle Events

1. Activity is partially or fully hidden
    a. For partial hide, Activity is still in memory, no state data (view or non-view) is lost.
    b. For full hide, Activity gets restarted but Android automatically saves the Activity's view state away in a Bundle object.
    c. Persistent data remains in its last updated state in non-volatile memory

2. Configuration events such as reorientation
    a. This is interpreted by Android as still requiring the activity view-state, however the **Activity must be destroyed** for reorientation, so it saves it in a Bundle object which it uses on relaunch to restore the Activity's view-state.
    b. Lifecycle callback is a bit similar to a full hide action, in this one, the Activity is **destroyed**, in a full hide, the Activity is **restarted**.

3. Activity is destroyed by back button or user swipes activity from recent/overview
    a. Android interprets as view-state **no longer required** so does not save it
    b. Persistent data remains in its last updated state in non-volatile memory
    c. Android does not even call onSaveInstanceState so there will be no view-state saved.

Note: Developers must code manually to save and restore non-view data in the same Bundle Object. This is done by overriding the **onSaveInstanceState()** and **onRestoreInstanceState()** methods and inserting key/value pairs into the Bundle.

Note: When an app is uninstalled from the device, all its data including persistent data stored in non-volatile memory is lost

Note: Make sure to call super on all overridden lifecycle callback methods. E.g. super.onSaveInstanceState(outState), super.onStop(), super.onStart() etc.

Example Code for Saving/Restoring an Activity Instance State (view + non-view)

1. onSaveInstanceState(Bundle outState)
    a. Not called if user indicates they are done with the activity (Back)
    b. Must call super to let Android save view-state
    c. Must code to manually save non-view state in Bundle (such as instance variables).

Below is the week's example:

1. First set a value in an EditText input box through a button.

```
public void setNonViewStateInstVar (View view) throws IOException{
    nonViewState = nonViewDataEditText.getText().toString();
    Log.i(TAG,  msg: "nonViewState instance var = " + nonViewState);
}
```

2. put the variable data into the Bundle as a key/value pair by calling putString() on **onSaveInstanceState(Bundle outState)**

```
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState); // save view data
    outState.putString(getString(R.string.tute1pm),nonViewState);

    Log.i(TAG,  msg: "onSaveInstanceState");
}
```

3. Retrieve the variable data back by calling the key in the Bundle on **onRestoreInstanceState(Bundle inState)**

```
//only gets executed if inState != null so no need to check for null Bundle
protected void onRestoreInstanceState(Bundle inState) {
    super.onRestoreInstanceState(inState); // restore view data
    nonViewState = inState.getString(getString(R.string.tute1pm));
    Log.i(TAG,  msg: "onRestoreInstanceState");
}
```

Saving/Restoring Persistent Activity Data (data across all an app's uses that is not likely to be modified)

Many options… (All writes to and reads from the device's non-volatile memory):

1. Saving Key-Value sets
2. Saving files
3. Saving Data in SQL Databases (Week 7+)

Saving/Restoring Persistent Activity Data through Key/Value Sets

- Works by using a **shared preferences** file for storing small data. We use an **Editor** object to create/edit key-value sets, then we call commit() or apply() the changes to the file system.
- We restore our persistent values from a SharedPreferences file as variables in **onCreate**, if there is nothing to restore then set default values.
- We update these variables holding current persistent values as required during activity operation
- We save the values of these variables holding current persistent values to their SharedPreferences file in onPause.
- we save our variables by calling editor.putString(key, value) and retrieve by calling sp.getString(key, default_value)

Below is the week's example:

1. First get the data by creating a variable to take the EditText input string

```
public void setPersistentInstVarVal (View view) throws IOException{
    persistentState = persistentDataEditText.getText().toString();
    Log.i(TAG,  msg: "set persistent instance var val = " + persistentState);
}
```

2. Then we create a function which sets the Persistent State/Data by creating a SharedPreference file and Editor, storing as key/value pair, and calling commit()

```
private void saveSharedPreferences(){
    SharedPreferences sp = getPreferences( mode: 0); // 0 means can only be accessed by this
                                                     // specific activity
    SharedPreferences.Editor editor = sp.edit();
    editor.putString(getString(R.string.persDataKey),persistentState);
    editor.commit();
```

Note: If we want to save data in a specific filename then we can specify by:

```
SharedPreferences sp2 = getSharedPreferences(getString(R.string.fileName), mode: 0);
```

3. Finally, we create a function to restore the data back into our EditText input.

```
private void restoreSharedPreferences(){

    SharedPreferences sp = getPreferences( mode: 0);
    persistentState = sp.getString(getString(R.string.persDataKey), s1: "");
    persistentDataEditText.setText(persistentState);
```

4. Most importantly, we call our functions during the lifecycle of our app at onPause and onStart to save the data during Activity lifecycle changes.

```
protected void onStart() {
    restoreSharedPreferences();
    super.onStart();
    Log.i(TAG, msg: "onStart");
}
```

```
protected void onPause() {
    saveSharedPreferences();
    super.onPause();
    Log.i(TAG, msg: "onPause");

}
```

During onPause(), the persistent data is stored as a key-value pair in the shared Preferences file.

During onStart(), the persistent data is restored from the shared Preference file, or if null, then default value is empty string.

Note: What is the difference between SharedPreferences and Bundle?

SP is an Object which points to a **non-volatile** memory file (we can specify filename if needed) and stores data sets as key/value pairs. Useful for retaining data between different app executions. Bundle is used to safely transfer data between activities or fragments and stores **volatile** data.

Similarities: They both store data as key/value pairs.

Similarities: Both have similar functions to store/retrieve key values:

SP: store is editor.putString(key, value). Retrieve is sp.getString(key, default_value).

Bundle: store is outState.putString(key, value). Retrieve is inState.getString(key).

Note: Difference between getSharedPreferences() and getPreferences()?

getSharedPreferences() is used when we need multiple SharedPreference files identified by name, so we specify two parameters, the filename, and the mode.

getPreferences is used when we only need one SharedPreference file for the activity (hence do not need to specify filename), so we specify one parameter only, the mode.

We only use mode 0, which means the SharedPreference data can only be accessed by that specific activity.

Exercise 1

Both reorientation and back events involve a destroy

a. *List the callbacks (lifecycle and other) in sequence that occur for a reorientation event*

1. onPause()
2. onSaveInstanceState(Bundle outState) – outState is the saved state to be placed into the Bundle object
3. onStop()
4. onDestroy()
5. onCreate()
6. onStart()
7. onRestoreInstanceState(Bundle savedInstanceState) – savedInstanceState is the data most recently supplied in onSaveInstanceState(Bundle).
8. onResume()

b. *List the callbacks (lifecycle and other) in sequence that occur for a back event*

1. onPause()
2. onStop()
3. onDestroy()

c. *What is the difference between a) and b) up to and including destroy?*

In a), instance state is saved as Android calls onSaveInstanceState() whereas in b) Android will not save the instance state.

d. *Why is there a difference?*

Android interprets the user tapping the back button as not needing the Activity anymore, hence it does not invoke onSaveInstanceState(), whereas a device reorientation event indicates the user is still on the Activity, hence the instance state must be preserved.

Exercise 2

a. *What are SharedPreferences files for?*

They are used as a form of storage for small scale private data in key-value pairs.

b. *Should they be used directly to save app settings? if not, what should be used?*

No. *Should use* ***getDefaultSharedPreferences()*** *instead to get the default SharedPreference file for our entire app.*

c. *What's the difference between what is referenced by the return values of getPreferences(…) and getSharedPreferences(…)?*

getPreferences() returns a default **SharedPreferences** file belonging to the activity it was called in.

getSharedPreferences() returns a **SharedPreferences** instance in the specified file name.

Exercise 3

Consider the following statement:

highScore = sharedPref.getInt(getString(R.string.saved_high_score), someVal);

a. *What is someVal?*

someVal is the default value to take if the key's value is null.

b. *What does getString do exactly?*

Converts the resource id referenced in its parameter into a string value.

c. *What role does getString's return value play as a parameter of the getInt method?*

it returns a string which is the key containing the value we want to look for in the sharedPreferences file.

Week 4: Simple UIs & Layouts (Constraint layout, XML, Screen Size & Resolution, Views, ViewGroups)

Views, ViewGroups, Layouts, Layout.Params

A **Layout** defines the structure of our UI in our app, with all elements built using a hierarchy of **View** and **ViewGroup** objects.

View class – imported from the package android.view.View

Represents the basic building block for UI components. A **View** occupies the area on screen responsible for drawing and event handling. Is the base class for widgets, which are used to create interactive UI components (buttons, text fields etc).

ViewGroup subclass – imported from package android.view.ViewGroup

The **base class** for **Layouts** and **Views** containers, it is a special **View** that contains other **View's** as its children. A ViewGroup allows **View's** to be nested, which is represented as a hierarchy called a **Layout**.

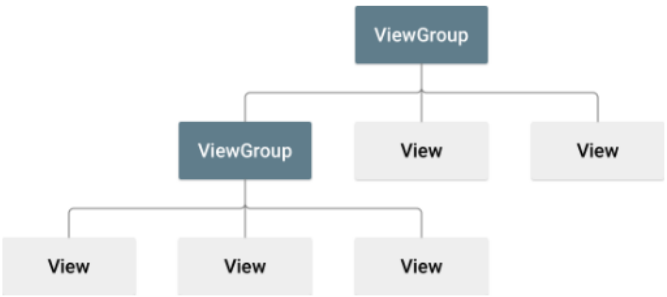


Figure 1. Illustration of a view hierarchy, which defines a UI layout

A Layout is responsible for managing the size, position and behaviour of all the Views it contains.

An example of a ViewGroup is LinearLayout or RelativeLayout, which are both subclasses of ViewGroup.

ViewGroup.LayoutParams – Sets of XML attributes used by the parent **ViewGroup** to define the size and position for each child **View**.

Every Layout Class contains:

1. XML attributes (also inherit attributes from View and ViewGroup)
2. Java methods which applies directly to the View's Layout Params object e.g. addView(View child)

Layout Types

ViewGroup has several Layout subclasses such as:

CoordinatorLayout, FrameLayout, GridLayout, LinearLayout, RelativeLayout

and indirect subclasses such as TableLayout.

Note: since a ViewGroup is a View, a ViewGroup can be contained by another ViewGroup.

So, Layouts can be nested to optimise parts of the UI, however, nesting can create efficiency and clarity issues, which is why Google introduced **ConstraintLayout** (see later).

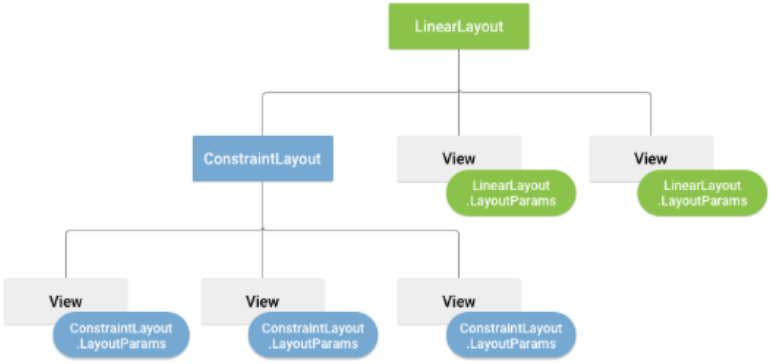
In Addition to ViewGroups

There is something called a View container, which contains many direct and indirect container subclasses that can be part of a UI's View hierarchy (e.g. Toolbar).

Layout Parameters

XML layout attributes defining layout parameters for the View that are appropriate for the ViewGroup in which it resides.

Every **ViewGroup** class implements a nested class that extends **ViewGroup.LayoutParams** – this subclass contains properties that define the size and position for each child view, as appropriate for the view group.



The parent **ViewGroup** defines the layout parameters for each child **View**.

There are many Layout Attributes that the class supports such as width, height, margin, etc. Base LayoutParams class just describes how big the view wants to be for both width and height. It can specify one of:

match_parent = view should be as big as its parent **ViewGroup** will allow

wrap_content = view should be only big enough to enclose its contents

Creating UIs in XML and Java

We can declare a layout in two ways:

1. **Declare UI elements in XML**, seen above.
2. **Instantiate layout elements at runtime**. Android creates View and ViewGroup objects (and are able manipulate their properties) programmatically.

Android Studio IDE allows us to use either or both these methods.

Java VS XML for UI Design

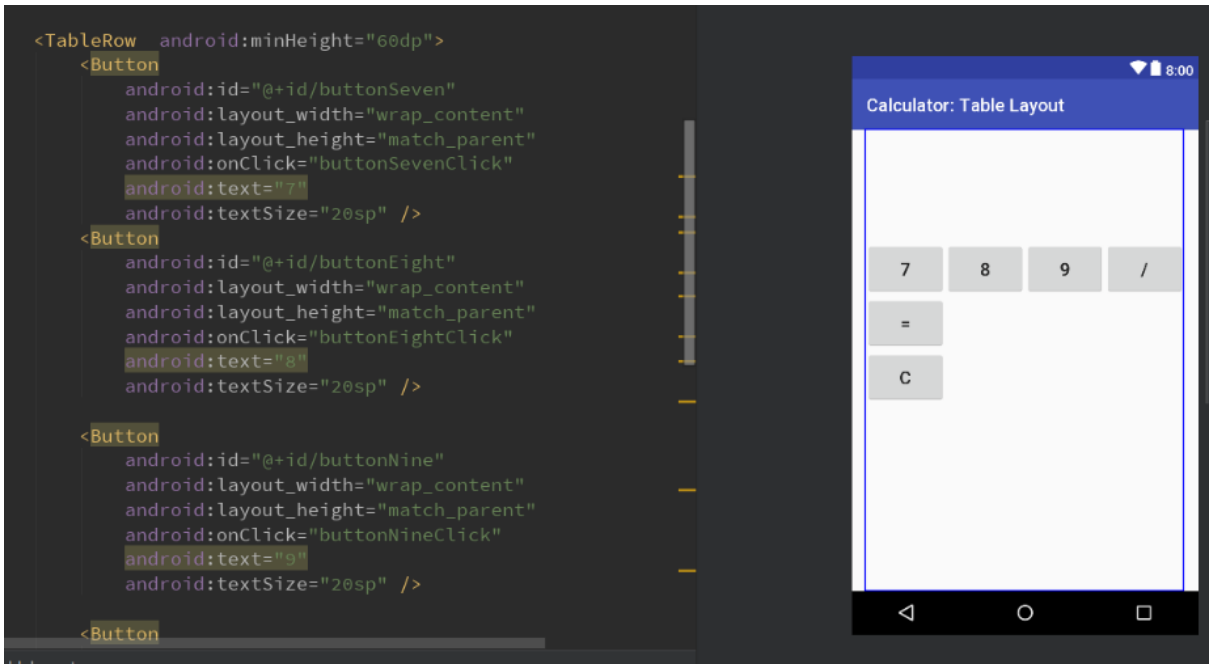
XML is better for separations of concerns – can separate the presentation of App from the behaviour of App.

Using XML files also makes it easy to provide different layouts for different screen sizes and orientations (easy resource selection).

Generally, XML vocabulary for UI elements closely follows the structure and naming of the classes and methods.

1. View Properties are often straightforward
    a. XML: widget attribute: android:text = " ";
    b. Java: widget method: .setText(…);
2. View Layout parameters are more complicated in code e.g.
    a. XML: widget attribute: android:layout_centerHorizontal="true"
    b. Java: .addRule(RelativeLayout.CENTER_HORIZONTAL)

Example code defining UI elements through XML



Defining UI elements in Java

Basic Procedure is…for each widget:

1. Instantiate the widget and customise it by setting its properties usually using "setPropName" methods.
2. Instantiate the **Layout** object to place widget on
3. Instantiate a **LayoutParams** object for the widget
    a. type depends on the layout (e.g. RelativeLayout.LayoutParams)
    b. Use constructor and addRule method to specify the size, position and behaviour of the widget in the layout
    c. Add the widget to the layout using addView() method with 1st parameter being the widget, and 2nd parameter being the LayoutParams object

Example Code defining UI elements (button in this case) through Java

```
//create a Button and set its properties
Button myButton = new Button(this);
myButton.setText("Press Me");
myButton.setBackgroundColor(Color.YELLOW);
myButton.setId(R.id.myButtonId);
```

```
//create a RelativeLayout and set its properties
RelativeLayout myLayout = new RelativeLayout(this);
myLayout.setBackgroundColor(Color.BLUE);
```

```
//now create a RelativeLayout.LayoutParams object for the Button to tell
//its parent Relative Layout how it wants to be sized positioned and behave
RelativeLayout.LayoutParams buttonParams =
        new RelativeLayout.LayoutParams(
                RelativeLayout.LayoutParams.WRAP_CONTENT,
                RelativeLayout.LayoutParams.WRAP_CONTENT);
```

```
//add Layout Parameter rules to an existing LayoutParam
buttonParams.addRule(RelativeLayout.CENTER_HORIZONTAL);
buttonParams.addRule(RelativeLayout.CENTER_VERTICAL);
```

```
//now lets add the Button to the RelativeLayout
myLayout.addView(myButton, buttonParams);
myLayout.addView(myEditText, textParams);

//finally inflate (show the RelativeLayout as this Activity's UI)
setContentView(myLayout);
```

View Properties

Every **View** API has a class reference page in Android Doco's e.g. TextView

Contains a table of XML attributes and a table of Java methods

Example TextView XML attributes

`android:fontFamily`
Font family (named by string or as a font resource reference) for the text.
`android:fontFeatureSettings`
Font feature settings.
`android:freezesText`
If set, the text view will include its current complete text inside of its frozen icicle in addition to meta-data such as the current cursor position.
`android:gravity`
Specifies how to align the text by the view's x- and/or y-axis when the text is smaller than the view.
`android:height`
Makes the TextView be exactly this tall.

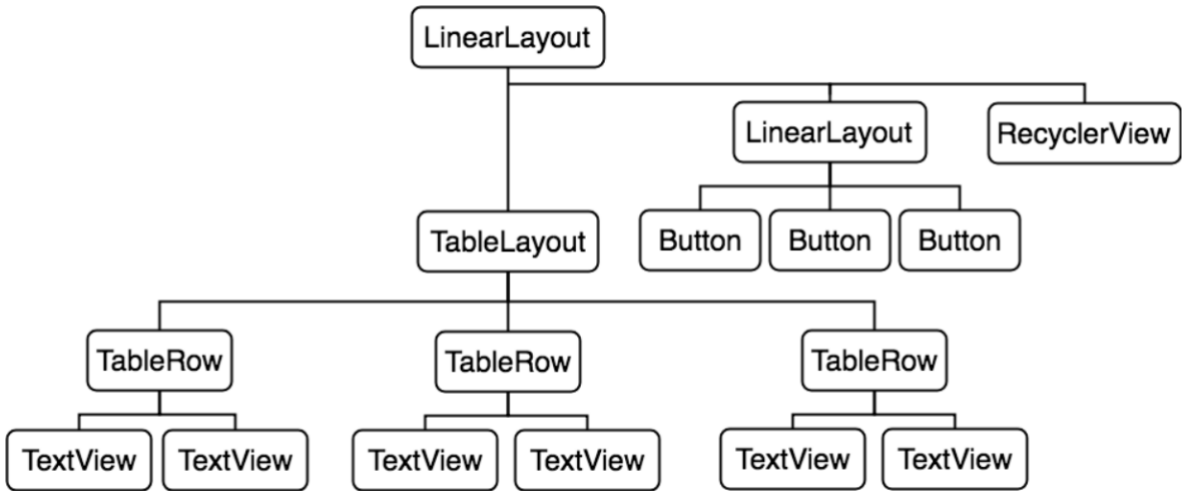
Example Java Methods for TextView

`final void`
`append(CharSequence text)`
Convenience method to append the specified text to the TextView's display buffer, upgrading it to `TextView.BufferType.EDITABLE` if it was not already editable.
`void`
`autofill(AutofillValue value)`
Automatically fills the content of this view with the **value**.

ViewGroup Layout for Week 4

For Week 4 Calculator App, we use TableLayout, which is a subclass of LinearLayout, and consists of a number of TableRow objects.

TableRow objects are by itself, **layouts** which arrange its child **View's** horizontally.



TableLayout is a layout that arranges its children into rows (through **TableRow**) and columns. Each row has zero or more cells, and each cell can hold one View object. the # of columns is dictated by the row with the most cells (Views).

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
             android:layout_width="match_parent"
             android:layout_height="match_parent"
             android:stretchColumns="*"
             android:layout_marginLeft="15dp"
             android:layout_marginRight="15dp">

    <TableRow  android:minHeight="60dp">

        <TextView
            android:id="@+id/resultScreen"
            android:layout_height="match_parent"
            android:layout_weight="1"
            android:textSize="30sp"
            android:textColor="@android:color/black"/>
    </TableRow>

    <TableRow  android:minHeight="60dp">

        <TextView
```

ConstraintLayout – Proposed solution to nested layouts, views are all linearly placed

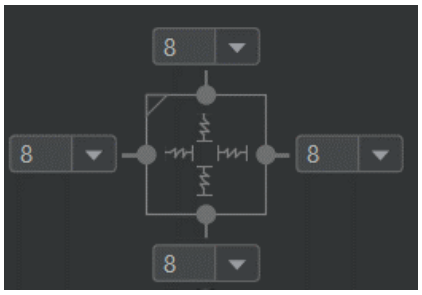
Available to API level 9 with Android 2.3 and higher.

- Views are attached to the layout through software analogues of **springs** or through **horizontal and vertical guidelines**
- These springs can expand and collapse
- Tension on springs holding a View between two endpoints can be biased towards one end by a percentage
- Hard margins can be specified at the end point of each spring

ConstraintLayout.LayoutParams

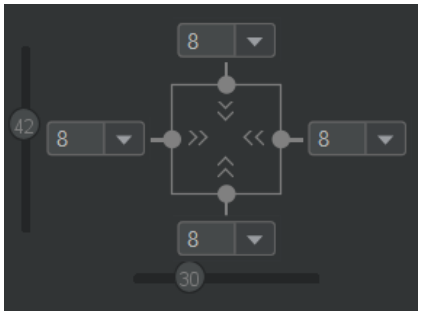
As stated, every ViewGroup implements a nested class that extends **ViewGroup.LayoutParams**. in this case it is **ViewGroup.LayoutParams**, Important attributes are:

match_constraint – layout size is defined by the size of the constraint hence is always 0dp.



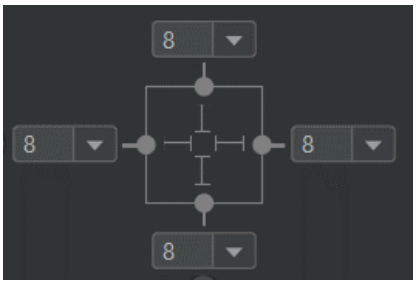
Example: a View's width and height both constrained under "match_constraint". Meaning size of the view matches the constraint, hence is 0dp. Vertical and horizontal bias are redundant.

wrap_content – size is defined by the required caption size of the view



Example: Similar to above except constrained under "wrap_content". Vertical and horizontal bias are considered, and hence positioned slightly upwards in height, and leftwards in width.

specific dimension values – specify the absolute values in size of the view – NOT RECOMMENDED DUE TO DEVICE VARIATION



Example: Similar to above, vertical and horizontal bias are considered.

Guidelines and Barriers

Other than software analogues of **springs**, we can also structure our **View's** by creating transparent horizontal/vertical **guidelines** or **barriers** to constrain (nest) our views onto.

The Difference between a Barrier and a Guideline is that a Guideline defines its own position, whereas a Barrier does NOT define its own position. A Barrier's position is fixed based on the largest absolute position of the views nested within it.

Essentially, A Barrier's position is flexible and always based on the size of the largest views that it references. A Barrier specifies the most extreme size of its views on the specified side whereas a Guideline's position is always fixed.

Chains

Another way to structure our **View's** is through chaining. We can chain multiple **View's** into a linked bi-directional position constraint. There are 3 main styles of chaining by specifying the attribute:

Spread: views are evenly distributed

Packed: views are packed together

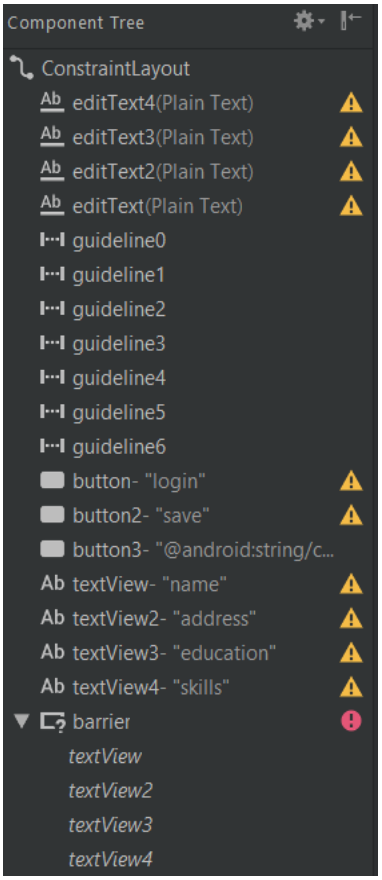
Spread inside: first and last view are affixed to the constraints on each end of the chain but rest are evenly distributed.

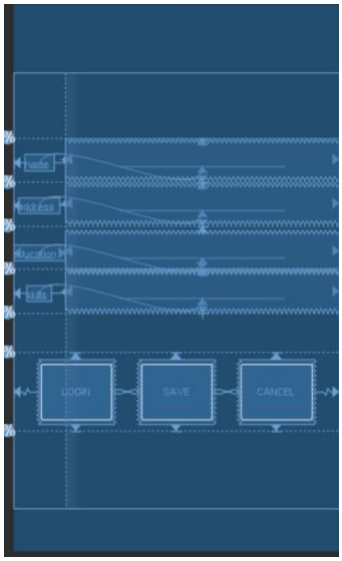
Additional Notes

Example ConstraintLayout Code

We have a ConstraintLayout ViewGroup which nests all our Views, with a Barrier which references 4 TextView's. There are also 4 EditText's and 3 Buttons.

- 7 Horizontal guidelines created to structure layout, with position specified by vertical percentage values.
- 4 EditText Objects are constrained under their respective guideline.
- Barrier defined and 4 TextView objects are nested under it. Barrier size is defined by the maximum width of its TextViews (longest text).
- TextView size is constrained under size of EditText Views on the right.
- Horizontal constraint defined for the lower buttons through chains with "spread" attribute.
- Vertical constraint is placed on buttons through its respective guideline





Styles, Themes and Material Design

Style - 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 layout XML file, located under values/styles.xml.

Style Inheritance – parent attribute in the <style> elements let you specify a style from which your style should inherit attributes. Can inherit from styles we created ourselves (above), or from built-in styles.

Can also specify a color.xml file under values/color.xml which works similarly to styles but solely for colours.

Referencing a value in colors.xml in LayoutParams

```
android:lines="2"
android:maxLines="2"
android:textSize="25sp"
android:textColor="@color/colorPrimaryDark"/>
```

colors.xml

```
<resources>
    <color name="colorPrimary">#3f51b5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
</resources>
```

Why code it like that? – Separate design from content, Separation of Concerns!

Theme – A style applied to an entire Activity or app [in the app's manifest file]. When a style is applied as a theme, every view in the activity or app applies each style attribute that it supports automatically.

Manifest.xml file in Calculator App using a Theme

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="Calculator: Table Layout"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
```

Definition for our theme in styles.xml

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
</style>
```

Material Design – A comprehensive guide for visual, motion, and interaction design across platforms and devices. Important features are ToolBars, Floating Action Buttons, Snackbar, AppBarLayout, etc.

Supports Android 5.0 API (Lollipop) level 21 and above.

Compatibility Complications – cannot use Material theme with standard Activity super class used for maximum backward compatibility, AppCompatActivity.

We use v7 Support Libraries for good backwards compatibility to Android 2.3 API 9.

Getting Most of Material Design with Compatibility 2.3 (API 9)

The AppCompat theme includes a built-in Action Bar for backwards compatibility, Since Lollipop does not include an Action bar anymore, we need to use a theme that suppresses the Action bar and use a widget called a Toolbar.

A **Toolbar** is a **View** that is **richer and more flexible than the traditional Action Bar**. They are also part of the View hierarchy, so they can be animated or even react to scroll events.

Also, as it's a View, we can use it as a standalone element anywhere in the App, and even have multiple ToolBars.

Action Bar and App Bar? - There is no UI element called App Bar or Action Bar. They are synonyms.

- Action Bar used to be default theme before Android Lollipop. Now not anymore.
- Developers are now expected to work with a ToolBar widget
- It is preferred to use ToolBar as your App Bar as Toolbar offers more control over appearance and functionality of Action bar.

To do this, make the app's theme extend Theme.AppCompat.NoActionBar.

Then create the ToolBar, and call setActionBar(toolbar) to designate it as the Action Bar in the Activity. This method adds the standard Action Bar methods for you. To Target older SDK's, we call setSupportActionBar(toolbar) instead.

Example of using the ToolBar widget and replacing the Action Bar with Toolbar

First, make our App's theme extend from Theme.AppCompat.NoActionBar.

```
android:theme="@style/Theme.AppCompat.NoActionBar">
```

```
<style name="Theme.AppCompat.NoActionBar">
    <item name="windowActionBar">false</item>
    <item name="windowNoTitle">true</item>
</style>
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.blah);

    Toolbar toolbar = (Toolbar) findViewById(R.id.my_awesome_toolbar);
    setActionBar(toolbar);
}
```

```
<android.support.v7.widget.Toolbar
    android:id="@+id/my_awesome_toolbar"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:minHeight="?attr/actionBarSize"
    android:background="?attr/colorPrimary" />
```

Exercise 1

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
             android:layout_width="match_parent"
             android:layout_height="match_parent"
             android:stretchColumns="*"
             android:layout_marginLeft="15dp"
             android:layout_marginRight="15dp">

    <TableRow  android:minHeight="60dp">

        <TextView
            android:id="@+id/resultScreen"
            android:layout_height="match_parent"
            android:layout_weight="1"
            android:textSize="30sp"
            android:textColor="@android:color/black"/>
    </TableRow>

    <TableRow  android:minHeight="60dp">
```

This is the start of the TableLayout that holds the calculator's various widgets.

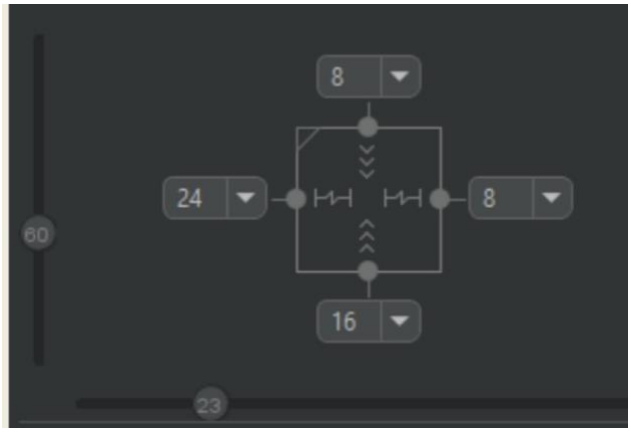
a) What is the purpose of layout_weight?

b) How many columns does it have?

a. Indicates how much of the extra space in the LinearLayout is allocated to the view associated with these LayoutParams.

b. 1 column as row with most cell has at most 1 view.

Exercise 2



Explain all you can about the horizontal and vertical layout of the widget depicted in the Layout Editor diagram above.

Horizontal layout - Left margin 24dp, Right margin 8dp, layout is defined by “match_constraint”, matching size of the view to the constraint size defined in the layout editor. Horizontal bias is 23% but it is redundant.

Vertical layout - Top margin 8dp, Bottom margin 16dp, and layout is defined by “wrap content”, wrapping the size of the view to the size it needs to display its content. Vertical bias is 60% and so it is tilted upwards more to the height constraints defined between the top 8dp and bottom 16dp margins.

Exercise 3

a. *What is the main difference between Guidelines and Barriers in UI design?*

A guideline allows us to specify a boundary and constrain views onto.

Similar to a guideline, a barrier is an invisible line that you can constrain views to, except a barrier does not define its own position; instead, the barrier's position moves based on the position of views contained within it.

The only difference between Barrier and Guideline is that a Barrier's position is flexible and is always based on the size of the largest view referenced within it, whereas a Guideline's position is always fixed.

b. *Assume you have 3 views (Buttons or TextViews), briefly describe two solutions to spread them equally in the available space.*

1. Create vertical/horizontal guidelines depending on what the user requires and spread the 3 guidelines equally to around 33% space. Then create constraints from the 3 views onto the guidelines. Layout should be match_constraint so that the size of the 3 views should all be evenly distributed.

2. Another solution would be to explicitly select all three of the views, then create a horizontal/vertical chain depending on what the user requires. This will automatically link the view with bi-directional position constraints and centre then evenly. Once chains are set, we can also distribute the views horizontally or vertically with different styles:
    i. Spread: views are evenly distributed
    ii. Packed: views are packed together
    iii. spread inside: first and last view are affixed to the constraints on each end of the chain and rest are evenly distributed.

Week 5: Advanced UI Design (FAB, Snackbar, App Bar, Options Menu, NavBar, Toast)

App 1: FAB + Material Design UI Components

This app utilizes the Android support library and implements Material design UI elements from the android.support.design package of the following:

- CoordinatorLayout
- AppBarLayout
- Toolbar
- FloatingActionButton
- Snackbar

The app also contains a ListView for displaying a list of scrollable items. The ListView uses an ArrayAdapter object which takes in a data source (e.g. an array) and converts each item into a view that's placed into the list. Acts as a bridge between data source and our view (ListView).

Configuring ListView with an Adapter

1. Get a reference to the ListView
2. Instantiate an adapter (specifying our context, layout, and data_source as param)
3. Attach the adapter into the ListView using ListView.setAdapter() method.

```
ArrayList<String> listItems = new ArrayList<~>();
ArrayAdapter<String> adapter;
private ListView myListView;
```

```
myListView = findViewById(R.id.listView); // a ListView
adapter = new ArrayAdapter<>( context: this, android.R.layout.simple_list_item_1, listItems);
myListView.setAdapter(adapter);
```

Note: We always call adapter.notifyDataSetChanged() each time we alter or update our data source.

CoordinatorLayout

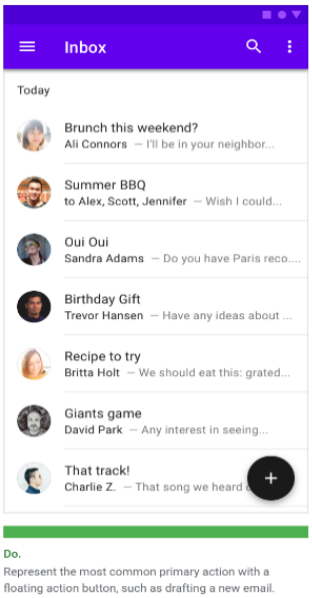
A ViewGroup which provides a top-level container for layouts incorporating MD components and behaviours. Also used as a container for specific interaction with one or more child views. Example interaction – FAB moves up when SnackBar appears.

AppBarLayout

A vertical LinearLayout ViewGroup which implements of many of the features of MD's app bar concept. Depends heavily on being used as a direct child within CoordinatorLayout.

ToolBar

A View which is a generalization of action bars for use within application layouts. Can designate a ToolBar as the action bar for an activity using the setSupportActionBar() method.



FloatingActionButton

A special ImageButton **View** used to perform the primary, or most common action on a screen. They appear to float above a background level.

We can specify what the button does through **setOnClickListener()** and instantiating an on click listener object. We then code an event handler in the listener object called **onClick()**.

Snackbar

```
Snackbar mySnackbar = Snackbar.make(view, stringId, duration);
```

A global widget for displaying lightweight feedback after an operation. **Not a View**, we use it by calling the Snackbar object with 3 parameters, the CoodinatorLayout to attach the Snackbar to, the message it displays, and the length of time to show the message. Afterwards we then call Snackbar.show() to display the message to the user.

Note: We do not instantiate a Snackbar object, we call the class directly by **.make()**

Snackbars can also contain an action which is set **via Snackbar.setAction(text, listener)**. The text works like a button, where if clicked, triggers the listener callback.

E.g. we can code a View.OnClickListener callback called "undoOnClickListener" which removes the most recent item and display it.

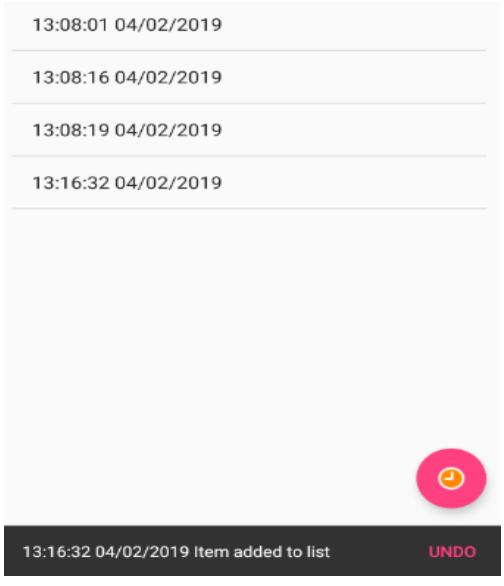
If the listener param is set to "null", then nothing is displayed in the Snackbar.

Example Code combining the logic of FAB and Snackbars

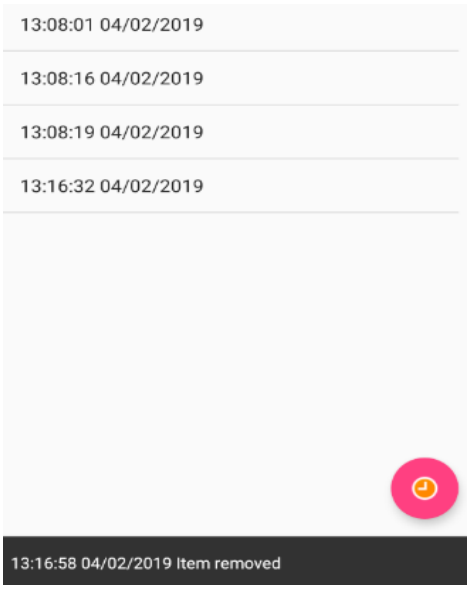
```
FloatingActionButton fab = findViewById(R.id.fab); // a Floating Action Button View
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        SimpleDateFormat dateFormat = new SimpleDateFormat( pattern: "HH:mm:ss dd/MM/yyyy", Locale.ENGLISH);
        String dateValue = dateFormat.format(new Date());
        addListItem(dateValue);
        Snackbar.make(view,  text: dateValue + " Item added to list", Snackbar.LENGTH_LONG)
                .setAction( text: "Undo", undoOnClickListener)
                .show();
    }
});
```

```
View.OnClickListener undoOnClickListener = new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        int lastIndex = listItems.size() - 1;
        String removedDate = listItems.get(lastIndex);
        listItems.remove(lastIndex);
        adapter.notifyDataSetChanged();
        Snackbar.make(view,  text: removedDate + " Item removed", Snackbar.LENGTH_LONG)
                .show();

    }
};
```

Clicking the FAB adds a Date object and emits a Snackbar displaying info which also contains an undo button.



Clicking the undo button emits the undoOnClickListener function, and emits a SnackBar displaying info



Toast

A View containing a quick little message for the user which disappears by themselves. It wants to be as unobtrusive as possible and cannot include an action button.

App 2: Navigation Drawer + Material Design UI Components

Navigation Drawers

A UI panel of options on the left edge of the screen (could also be right) that shows our app's main navigation menu. It is initially hidden from view until the user swipes from the left or clicks the drawer icon in the app bar. It could also be permanently visible if there is enough screen space e.g. on a tablet or on landscape mode.

A navigation drawer allows the app to have many features without being visually overwhelming.

Creating a Navigation Drawer

First, Create a **DrawerLayout ViewGroup** with **2 Child Views**

1. **1st Child** must be the View from the main content, when nav is hidden
2. **2nd Child** is a **NavigationView**, which defines the navigation drawer itself

Example DrawerLayout XML Definition

```
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">

    <!--this is our content layout when nav drawer is hidden-->
    <include
        layout="@layout/app_bar_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

    <!--this is the view for the nav drawer itself-->
    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_main"
        app:menu="@menu/activity_main_drawer"/>

</android.support.v4.widget.DrawerLayout>
```

Defining how the **NavigationView** (Drawer) looks

Important attributes must be specified:

- android:layout_width : should usually be set to “wrap_content”
- android:layout_height : should usually be set to “match_parent”
- app:headerLayout : A layout to define how the header of the drawer looks
- app:menu : a collection of menu items defined under section @menu as an XML, specifying the icon, title, and id.

Week 5 Drawer visuals Example

Our headerLayout is a **LinearLayout** containing 1 ImageView and 2 TextView’s

Our menu layout is a list of items enclosed under ‘menu’ tag. We can specify titles also.

As you can see, the XML definition for navigation menu items are very similar to options menu item definition.

```
<item android:title="Up Down">
    <menu>
        <item
            android:id="@+id/up_button"
            android:icon="@drawable/arrow_up"
            android:title="Up"/>
        <item
            android:id="@+id/down_button"
            android:icon="@drawable/arrow_down"
            android:title="Down"/>
    </menu>
</item>

<item android:title="Left Right"...>

<item android:title="Resize TextView"...>
```

Handling Click Events in Navigation Drawers

Unlike the options menu, our Activity does not automatically listen o the menu clicks, so we need our Activity to implement the interface **NavigationView.OnNavigationItemSelectedListener**

Then, we need to attach a listener to our defined NavigationView by

NavigationView.setNavigationItemSelectedListener(this)

Now we override the callback method **onNavigationItemSelected(MenuItem item)**

This callback provides us with the drawer item logic handling. We identify the item's by calling item.getItemId(), with their id's being respective of their ID's in XML.

Example Handler for Navigation Drawer Items based on W5 App (Incomplete)

```
@SuppressWarnings("StatementWithEmptyBody")
@Override
public boolean onNavigationItemSelected(MenuItem item) {
    // Handle navigation view item clicks here.
    int id = item.getItemId();

    if (id == R.id.nav_camera) {
        // Handle the camera action
    } else if (id == R.id.nav_gallery) {

    } else if (id == R.id.nav_slideshow) {

    } else if (id == R.id.nav_manage) {

    } else if (id == R.id.nav_share) {

    } else if (id == R.id.nav_send) {

    }

    DrawerLayout drawer = findViewById(R.id.drawer_layout);
    drawer.closeDrawer(GravityCompat.START);
    return true;
}
```

ActionBarDrawerToggle

A handy class which ties together the functionality of DrawerLayout and the ActionBar to implement the recommended design for navigation drawers. Stuff it does is:

1. Inserting a hamburger icon in the App bar that rotates on drawer open/close
2. Opening and closing the drawer when clicking the hamburger icon

To use it, we create an instance of it and supply our context, DrawerLayout and toolbar into it. We call syncState() on our instance to sync the drawer's state and its hamburger icon in the app bar.

Example DrawerLayout and NavigationView Java Definition

```
//DrawerLayout
DrawerLayout drawer = findViewById(R.id.drawer_layout); //DrawerLayout
ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(
        activity: this, drawer, toolbar,
        R.string.navigation_drawer_open,
        R.string.navigation_drawer_close);
drawer.setDrawerListener(toggle);
toggle.syncState();

//NavigationView
NavigationView navigationView = findViewById(R.id.nav_view); //NavigationView
navigationView.setNavigationItemSelectedListener(this);
```

Opening and Closing the Drawer

To open and close the drawer, we use the DrawerLayout's closeDrawer(…) and openDrawer(…) methods. There are two main cases to handle:

1. Drawer should close after menu item selection when the appropriate response has executed. So we call closeDrawer() at the end of **onNavigationItemSelected** (see above example).
2. If drawer is open when back button is pressed, then Android should respond by just closing the drawer. We code this logic in the Activity's onBackPressed() callback.

```
@Override
public void onBackPressed() {
    DrawerLayout drawer = findViewById(R.id.drawer_layout);
    if (drawer.isDrawerOpen(GravityCompat.START)) {
        drawer.closeDrawer(GravityCompat.START);
    } else {
        super.onBackPressed();
    }
}
```

Note: GravityCompat states which way the drawer should inflate from, START = from left side, END = from right side.

Options Menu

The primary collection of menu items for an activity. It is where you should place actions that have a global impact on the app, such as “Search”, or “Settings”. The app bar and options menu are related. Menu items are located in the App bar.

Difference between Navigation Drawer is that the options menu is for actions and navigation drawers are for navigation.

Option menu items can appear as **action buttons** (through showAsAction attribute) in the App bar whereas navigation drawer menu items cannot do this.

Only similarity between the two is that Navigation Drawer menu and Options Menu have similar XML definitions.

Note: the item attribute **showAsAction** specifies when and how our menu items should appear in the App Bar. “never” means the menu item cannot flow into the action bar. “always” means the menu item cannot overflow into the Options menu.

Creating the Layout for the Options Menu

We Override a function onCreateOptionsMenu(Menu menu) and inflate a layout as the menu. We get the menu inflater by calling **getMenuInflater,** then call it’s inflate function, supplying the menu layout, and the menu object received. The menu layout should be our XML file containing menu and child items.

Handling Click Events in the Options Menu

We Override a function onOptionsItemSelected(MenuItem item). We distinguish specific events by calling item.getItemId(), which references the id’s of our XML menu items

```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}
```

Example Code

Note: We can also reference our menu items by index as well. We do this by calling myMenu.getItem(indexValue). The item's index is defined by its order in the XML menu resource file.

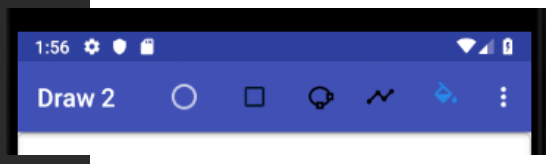
Also remember to return true afterwards to consume the click event.

Example XML Definition for Options Menu

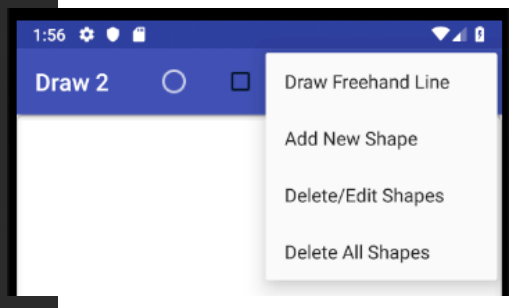
See how the menu items are presented differently based on their **showAsAction** attribute. If nothing is specified for **showAsAction** attribute, default is "never".

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/draw_circle"
        android:icon="@drawable/ic_circle_white"
        android:title=""
        app:showAsAction="always"
        />
    <item
        android:id="@+id/draw_rectangle"
        android:icon="@drawable/ic_square_black"
        android:title=""
        app:showAsAction="always"/>

    <item
        android:id="@+id/draw_ellipse"
        android:icon="@drawable/ic_ellipse_black"
        android:title=""
        app:showAsAction="always"/>

    <item
        android:id="@+id/draw_straight_line"
        android:icon="@drawable/ic_straight_line_black"
        android:title=""
        app:showAsAction="always"/>


    <item
        android:id="@+id/show_Color_selector"
        android:icon="@drawable/md_blue_500"
        android:title=""
        app:showAsAction="always" />
```

.




```
    <item
        android:id="@+id/draw_line"
        android:icon="@drawable/ic_line_black"
        android:title="Draw Freehand Line"
        app:showAsAction="never"/>

    <item
        android:id="@+id/add_shape"
        android:icon="@drawable/ic_add_white_48px"
        android:title="Add New Shape"
        app:showAsAction="never"/>
    <item
        android:id="@+id/edit_delete_shape"
        android:icon="@drawable/ic_mode_edit_white_48px"
        android:title="Delete/Edit Shapes"
        app:showAsAction="never"/>
    <item
        android:id="@+id/delete_all"
        android:icon="@drawable/ic_delete"
        android:title="Delete All Shapes"/>
</menu>
```



```
View.OnClickListener undoOnClickListener = new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        listItems.remove( index: listItems.size() -1);
        adapter.notifyDataSetChanged();
    }
};
```

Exercise 1

Examine the above code:

a. *What is the purpose of "notifyDataSetChanged"?*

Notifies the attached adapter that the underlying data source has been changed and any View reflecting the data set should be updated and refreshed.

b. *When should we call it?*

We call it whenever there are changes made to the data source that the adapter is referencing, such as when client has added, or removed something in the array.

c. *What is the task of "undoOnClickListener?"*

An interface definition for a callback to be invoked when a view is clicked. The method we override is **onClick**, The task of this listener is to remove the most recently added item in **listItems**, which is the item at the end of the list index, hence essentially acting as an "undo" listener action.

Exercise 2

a. *Why return true after handling an options menu's menu item click?*

In order to consume the function and notify the system that the function has completed successfully.

b. *Where in code is an options menu inflated? (class and method)*

The MainActivity Class, function is **onCreateOptionsMenu()** which we override.

c. *Where in code are an options menu's menu item clicks handled? (class and method)*

The MainActivity Class, function is **onOptionsItemSelected()** which we override. However we still call its super method at the end to keep default standard processes.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Your Logic
    return super.onOptionsItemSelected(item);
}
```

Exercise 3

Examine the above code:

a. *When does "onOptionsItemSelected" get invoked?*

   The function gets invoked whenever an item in your options menu is selected.

b. *If you have 2 items (item1 and item2) in your menu and each one has its own method (method 1 and method2) respectively, replace "Your Logic" with a code that invokes these methods based on the selected item.*

```
int id = item.getItemId();

if (id == R.id.item1) {
method1();
    return true;
} else if (id == R.id.item2) {
    method2();
    return true;
}
```

Week 6: Advanced UI Design II (RecyclerView, CardView, Fragments, Resource Selection & Managing Alternate Resources

App 1: CardDemo AppBar

This app utilizes the Android support library and implements from the android.support package for the following:

- RecyclerView
- CollapsingToolbarLayout
- LinearLayoutManager

Similar packages we utilized last week were Toolbars and Options Menu.

RecyclerView

A special ViewGroup for displaying a list of scrollable items. RecyclerView is a more advanced and flexible version of ListView which usually offers smoother scrolling.

Should use the RecyclerView widget when using data collections whose elements change at runtime based on user action or network events.

The widget works for displaying large data sets that can be scrolled very efficiently by maintaining a limited number of [recycled] views.

To work with a RecyclerView, we also need to instantiate two of its subclasses, RecyclerView.LayoutManager & RecyclerView.Adapter.

RecyclerView.LayoutManager

Must be instantiated in order to measure and position item views within a RecyclerView as well as determining when to recycle **Views** that are no longer visible to the user. It is like a container for RecyclerView's list items.
The RecyclerView will instantiate and set the LayoutManager when being inflated.

RecyclerView.Adapter

Responsible for providing views that represent items in a data set.

Setting up our RecyclerView

```
RecyclerView recyclerView;
RecyclerView.LayoutManager layoutManager;
RecyclerView.Adapter adapter;
```

```
recyclerView = findViewById(R.id.recycler_view);

layoutManager = new LinearLayoutManager( context: this);
recyclerView.setLayoutManager(layoutManager);

adapter = new RecyclerAdapter();
recyclerView.setAdapter(adapter);
```

Configuring our RecyclerView's Adapter

An adapter of subclass RecyclerView.Adapter is instantiated and plugged into the RecyclerView.

In this week's app, we code the list's data in a separate java file RecyclerAdapter.java. In other cases, the adapter's data could come from a database, via a content provider.

The adapter requires 3 method implementations to work:

1. **onCreateViewHolder(ViewGroup parent, int viewType)**: Inflates a new View from our supplied card_layout. We create a ViewHolder instance which wraps our View in order to store all required references to widgets inside the View for data binding.
   The ViewHolder gets recycled as the user scrolls down on the screen, with **onBindViewHolder()** reusing the view and binding new data to it.

2. **onBindViewHolder(ViewHolder holder, int position)**: bind a item's data to its ViewHolder instance and also bind listeners (usually onClick of list item).
   Position is the position of the item within the adapter's data set.

3. **getItemCount()**: internal API use, tells Android how many items to consider in our ListView.

Example AdapterClass Implementation on Week 6 CardDemoApp

```
@Override
public ViewHolder onCreateViewHolder(ViewGroup viewGroup, int i) {
    View v = LayoutInflater.from(viewGroup.getContext())
            .inflate(R.layout.card_layout, viewGroup,  attachToRoot: false);
            //CardView inflated as RecyclerView list item
    ViewHolder viewHolder = new ViewHolder(v);
    return viewHolder;
}
```

```
@Override
public void onBindViewHolder(ViewHolder viewHolder, final int position) {
    viewHolder.itemImage.setImageResource(images[position]);
    viewHolder.itemTitle.setText(clubNames[position]);
    viewHolder.groundText.setText(grounds[position]);

    final int fPosition = position;

    viewHolder.groundText.setOnClickListener(new View.OnClickListener() {
        @Override public void onClick(View v) {
            Snackbar.make(v,  text: grounds[fPosition] + ": "
                    + groundCapacities[fPosition],Snackbar.LENGTH_LONG)
                    .setAction( text: "Action",  listener: null).show();
        }
    });

}
```

```
@Override
public int getItemCount() {
    return clubNames.length;
}

public class ViewHolder extends RecyclerView.ViewHolder {
    public View itemView;
    public ImageView itemImage;
    public TextView itemTitle;
    public TextView groundText;
    public TextView colorPatch;

    public ViewHolder(View itemView) {
        super(itemView);
        this.itemView = itemView;
        itemImage = itemView.findViewById(R.id.item_image);
        itemTitle = itemView.findViewById(R.id.item_title);
        groundText = itemView.findViewById(R.id.ground_text);
        colorPatch = itemView.findViewById(R.id.color_patch);

    }
}
```

Note: a new ViewHolder instance is created through the defined constructor. This is where we can store all references to widgets in the CardView instance that will need data binding later on in **onBindViewHolder**.

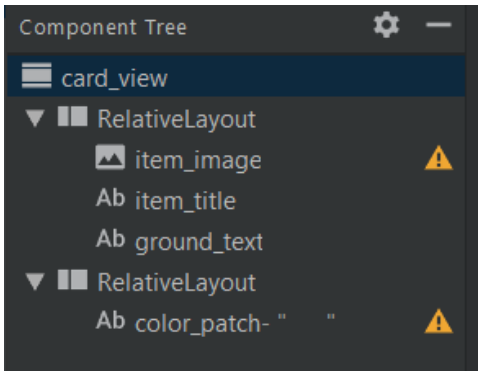
CardView

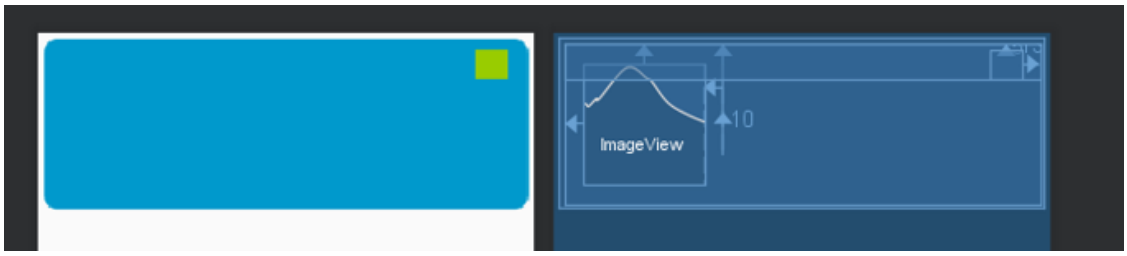
An element of Material Design, CardView extends the FrameLayout ViewGroup to provide rounded corner background and shadows.

CardView can be used as the template for each item in a ListItem or RecyclerView. And stores the data as a specific View which we can structure.

Defining a CardView Layout

In Week 6 Code, our CardView layout is stored at card_layout.xml and this layout is inflated in the **onCreateViewHolder()** method of the RecyclerView's adapter Class.





For each Item, the CardView is attached a new ViewHolder instance containing references to its widgets, which then allows the RecyclerView to bind data to it and attach listeners as required via the method **onBindViewHolder()**.

App Bar – Discussed in Week 4 & 5

CollapsingToolBarLayout

A child view of the AppBarLayout, it implements various App Bar behaviours. A behaviour we see is that the toolbar can collapse when scroll screen up.

App 2: LiveStock App

This app is very similar to App 1's CardDemo App, the main difference is the data source as the CardDemo uses synthetic data while this App retrieves its data from a real web server using HTTP requests. Hence, the RecyclerView's Adapter takes an array of type <StockItem>.

```
adapter = new RecyclerAdapter(dataItems);
recyclerView.setAdapter(adapter);
```

Three Classes

1. MainActivity
    a. Inflate the layout
    b. Send HTTP GET request
    c. Parse the response and generate the list of items
    d. Create and setup the ArrayAdapter
2. RecyclerAdapter
    a. Accept the list of items from the MainActivity class
    b. Build card views
    c. Bind data to the card views
3. StockItem
    a. Represents one item
    b. Has four attributes which are: title, open, close, and volume of the stock.

Data Source

The app sends a request to https://www.alphavantage.co/ to get the live updates of stocks.

The request is :
https://www.alphavantage.co/query?function=TIME_SERIES_INTRADAY&symbol=GOOGL&interval =15min&apikey=PU8GPVJCDYMBBAQF

where:

https://www.alphavantage.co	**The remote server**
TIME_SERIES_INTRADAY	The required function
GOOGL	Stock Symbol (Google)
15min	Interval. Other values:5min, 10min,30min
PU8GPVJCDYMBBAQF	Apikey (Free and lifetime)

App 3: MasterDetail App

Four Java Classes

1. ItemDetailActivity – Activity solely for narrow-width devices. It represents a single Item detail screen. On tablet-size devices, item details should be presented side-by-side (on detail pane) with a list of items in **ItemListActivity**.
2. ItemDetailFragment – a Fragment Class
3. ItemListActivity – Activity representing a list of items. Has different presentations for handset and tablet-size devices:
    - On handset, activity presents list of items, when pressed, leads to **ItemDetailActivity** representing item details
    - On tablets, activity presents list of items and item details side-by-side using two vertical panes.
4. DummyContent – a dedicated class that creates and makes available dummy data

Resource Selection of ItemListActivity.java

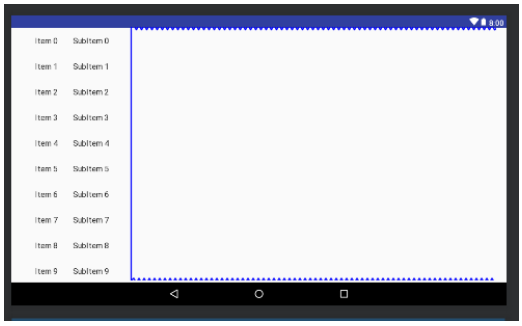
- activity_item_list.xml is inflated by the layout defined elsewhere through: <include layout="@layout/item_list"/>
- There are 2 item_list.xml files, one as default, and one as layout-w900dp. ***Android will decide which file is to be used depending on the device the app is running on. Specifically, whether the current width of its viewport is less than 900dp, or greater than or equal to 900dp***
- To inform the App of Android's decision we have a class Boolean variable mTwoPane set to true or false by calling findViewById on a UI component which is only present if the device's viewport is more than or equal to 900dp.

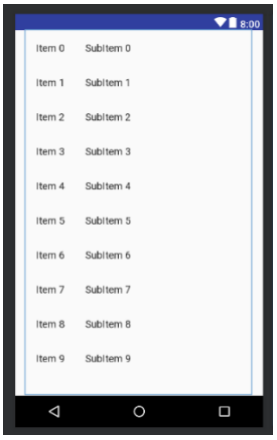
```
if (findViewById(R.id.item_detail_container) != null) {
    // The detail container view will be present only in the
    // large-screen layouts (res/values-w900dp).
    // If this view is present, then the
    // activity should be in two-pane mode.
    mTwoPane = true;
}
```

IN SUMMARY :

Our App's launch layout is **activity_item_list**.xml, which has a FrameLayout which takes an **item_list**.xml as the list layout.

Android will decide to choose the **item_list**.xml containing only a RecyclerView (View for scrollable list items) if the device viewport width is < 900dp, or the **item_list**.xml containing both a RecyclerView + detail pane as an FrameLayout if width is >= 900dp.

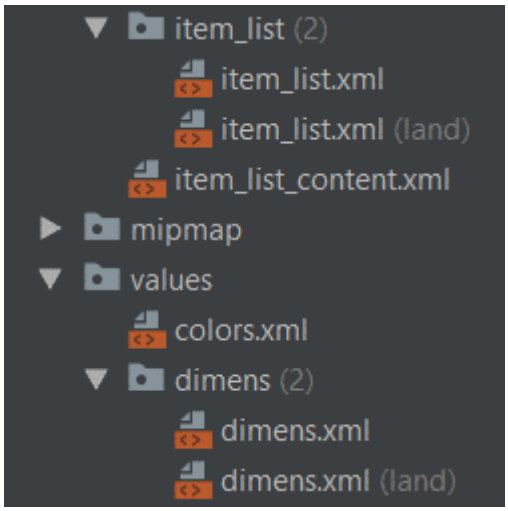




Another Way of Resource Selection

We can code our app so it only distinguishes between portrait and landscape orientation regardless of device.

Example Directory

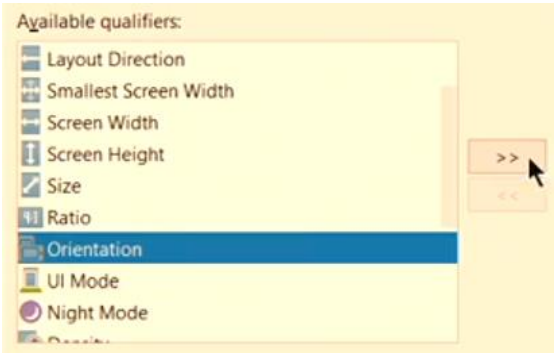


To fix size and height issues, we then have separates dimens.xml files, one for portrait and one for landscape.

Other Forms of resource selection

To create alternate resource files in Android Studio, simply right click the XML file, and select new layout resource file.

Android only selects one layout file, and decides the most appropriate file to load for the user device based on the specified qualifiers.



NestedScrollView

Like a ScrollView except it understands how to enclose another scrolling view or be the child of another enclosing ScrollView. It has many properties to fine tune Android reaction to scrolling gestures.

Fragment

Mini-activities with their own lifecycles (although slaved to their parent Activity). Can even be placed on the back stack. Our class must extend Fragment override its **onCreateView()** method, which returns a View in its lifecycle callback (inflating a layout view depends if we want a visible UI or not).

Fragment Example

```
public void handleBtn1(View view){
    getSupportFragmentManager().beginTransaction()
            .replace(R.id.frag1,new Fragment1())
            .addToBackStack("f1").commit();
```

To make sure fragments align to the back stack, we call addToBackStack().

To replace a UI section in an activity into a fragment, we call: **getSupportFragmentManager().beginTransaction().replace(layoutToReplace, FragClass).commit().**

commit() invokes the callback signifying the method is complete and returns the callback to the system.

What the code statement does is, we replace the current view ID, “frag1” into a new instance of our target fragment class “Fragment1”, and with its onCreateView(), returns the layout to inflate.

We can also call **.add()** instead of **.replace()**, the difference between the two is:

.replace() removes the old and previous fragment and creates a new fragment into the container.

.add() retains the previous existing fragments() and adds a new fragment to the activity.

Exercise 1: What is the relation between onCreateViewHolder and onBindBiewHolder callbacks?

onCreateViewHolder creates the ViewHolder instance for a View and returns the ViewHolder. onBindViewHolder uses the ViewHolder and binds an item's data to the ViewHolder instance continuously whenever it gets recycled.

Exercise 2:

a. *CollapsingToolbarLayout is a wrapper for Toolbar which implements a collapsing app bar, briefly explain how to add a Collapsing Toolbar to your app.*

Enclose the CollapsingToolBar under an App Bar View and inside it, wrap a ToolBar View alongside other views we want such as an ImageView. We can also set the characteristics of our CollapsingToolBar by calling collapsingToolBarLayout.setTitle() or collapsingToolBarLayout.setContentScremColor().

b. *What is the task of ViewHolder in RecyclerView? Explain briefly.*

A ViewHolder describes an item View and metadata about its place within the RecyclerView. They are a cache of View references to minimise the use of the slow findViewById method.

Exercise 3:

a. *What is qualifier in resource selection? support your answer with 4 examples*

Qualifiers in resource selection are practices in how we should always externalize app resources so that we can maintain them independently. We provide alternative resources for many reasons such as to account for screen width, orientation, versions, or screen density. Android supports the automatic selection of resources fitting to the device configuration.

For example for our MasterDetail App, we have separate configuration files for screens with viewport width over 900, and 900.

b. Briefly explain two differences between Fragments and Activities.

Fragment represents a behaviour or a portion **of** user interface **in an Activity**. A single **Activity** can have one or more **fragments** visible. **Fragments** were designed to be modular, self-contained UI components that can be reused many times within the same app or **between** apps. A **fragment** can't exist independently but **Activities** can.

```
Bundle arguments = new Bundle();
arguments.putString(ItemDetailFragment.ARG_ITEM_ID,
        getIntent().getStringExtra(ItemDetailFragment.ARG_ITEM_ID));
ItemDetailFragment fragment = new ItemDetailFragment();
fragment.setArguments(arguments);
getSupportFragmentManager().beginTransaction()
        .add(R.id.item_detail_container, fragment)
        .commit();
```

Exercise 4:

a. What are the two parameters of .add() method?

first parameter is the reference to the layout via ID to inflate the contain data in. Second parameter is the fragment class activity layout to inflate.

b. What is the reason of creating instance from the Bundle class (line 61)? Is it a must to have a bundle?

We create a bundle here to pass data across Android activities and fragments. It is necessary to use it because we want to pass our activity data and restore the activity's state via calling arguments efficiently. We do this by putting our data into the bundle, then calling setAgruments(Bundle) onto our fragment class. We later retrieve our data via the fragment with getArguments(). It is easier for the system to restore its values when the fragment is re-instantiated.

c. what does commit do?

commit() invokes the callback signifying the method is complete and returns the callback to the system.

Week 7: Databases PART 1 (SQLite)

1. *Examine and understand how to correctly implement content providers within an application*
2. *Examine and understand how to access shared data from within another application*
3. *Understand the advantages and disadvantages of Content Providers/Resolvers vs. traditional databases*

Most Android apps need to save data, even if only to save information about the app state during onPause() so the user's progress is not lost. This can be done through a database. There are 3 classes necessary to define a database:

1. Contract Class – used to explicitly define the schema of our database

This is done by setting global constants for all table names and table column names

2. DbHelper Class – used to create and upgrade the tables

This does all the detailed work expected of a database, e.g. open/close db, add, edit, delete table rows, update schema etc.

3. ContentProvider Class – helps an application manage access to data stored by itself. It encapsulates the data and provide mechanisms for defining data security to other apps. **Note: not shown in example app.**

Example Contract Class

```
package edu.monash.fit2081.db.provider;

public class SchemeShapes {

    public static final class Shape {
        //Table name
        public static final String TABLE_NAME = "shapes";

        //Table Column names
        public static final String ID = "_id"; //CursorAdapters will not work if this column
                                               // with this name is not present
        public static final String SHAPE_NAME = "shape_name";
        public static final String SHAPE_TYPE = "shape_type";
        public static final String SHAPE_X = "shape_x";
        public static final String SHAPE_Y = "shape_y";
        public static final String SHAPE_WIDTH = "shape_width";
        public static final String SHAPE_HEIGHT = "shape_height";
        public static final String SHAPE_RADIUS = "shape_radius";
        public static final String SHAPE_BORDER_THICKNESS = "shape_border_thickness";
        public static final String SHAPE_COLOR = "shape_color";

        // To prevent someone from accidentally instantiating the contract class,
        // make the constructor private.
        private Shape(){}
    }
```

DbHelper Class

Must extend SQLiteOpenHelper which provides important callbacks when the database is opened, such as **onCreate** and **onUpgrade**.

We also import package android.database.sqlite.SQLiteDatabase. This allows us to create objects responsible for CRUD methods of a database such as update(), delete(), query().

The constructor, in this case ShapesDbHelper is important as it does 2 tasks:

1. Call the super to check if the database exists, if it does not exist it will invoke **onCreate**.
2. Checks the version of the current database, if lower than stored, then **onUpgrade** will be invoked. If same does nothing. If higher than stored version, then **onDowngrade** will be invoked.

Example DbHelperClass

```
public class ShapesDbHelper extends SQLiteOpenHelper {
    // Database name and version
    private final static String DB_NAME = "ShapesDB.db";
    private final static int DB_VERSION = 1;

    private final static String SHAPES_TABLE_NAME = SchemeShapes.Shape.TABLE_NAME;

    // SQL statement to create the database's only table
    private final static String SHAPES_TABLE_CREATE =
            "CREATE TABLE " +
                    SchemeShapes.Shape.TABLE_NAME + " (" +
                    SchemeShapes.Shape.ID + " INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, " +
                    SchemeShapes.Shape.SHAPE_NAME + " TEXT, " +
                    SchemeShapes.Shape.SHAPE_TYPE + " TEXT," +
                    SchemeShapes.Shape.SHAPE_X + " INTEGER," +
                    SchemeShapes.Shape.SHAPE_Y + " INTEGER," +
                    SchemeShapes.Shape.SHAPE_WIDTH + " INTEGER," +
                    SchemeShapes.Shape.SHAPE_HEIGHT + " INTEGER," +
                    SchemeShapes.Shape.SHAPE_RADIUS + " INTEGER," +
                    SchemeShapes.Shape.SHAPE_BORDER_THICKNESS + " INTEGER," +
                    SchemeShapes.Shape.SHAPE_COLOR + " TEXT);";

  public ShapesDbHelper(Context context) {...}

  @Override
  public void onCreate(SQLiteDatabase db) { db.execSQL(SHAPES_TABLE_CREATE); }

  //couldn't afford to be this drastic in the real world
  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {...}

  //could return a boolean because insert returns the row ID of the newly inserted row, or
  //-1 if an error occurred, we don't bother though
  public void addShape(ShapeValues shape) {...}

  public boolean deleteShape(int shapeID) {...}

  public Cursor getAllShapes() {...}

  //update method requires record data to be packed into a ContentValues data structure
  public boolean updateShape(ShapeValues shape) {...}
```

To perform actions on the database, we need to read and write from it.

For Inserting, deleting and updating records, we call getWritableDatabase().

For querying and filtering the database, we call getReadableDatabase().

ContentValues - A class we use when we want to insert/update records in our database. We need to pack our data into this class, done through contentValues.put(column_attribute_name, value) method.

Example Code for adding a Record

Assume our contentValues Object is already packed with (column_name, value) data pairs.

```
public void addTask(ContentValues contentValues) {
    SQLiteDatabase db = getWritableDatabase();
    db.insert(TaskScheme.TABLE_NAME,  nullColumnHack: null, contentValues);
    db.close();
}
```

Example Code for updating a record - Same assumption.

```
//update method requires record data to be packed into a ContentValues data structure
public boolean updateShape(ShapeValues shape) {

    boolean result; //did the edit succeed or not
    SQLiteDatabase db = getWritableDatabase();
    ContentValues contentValues = new ContentValues();
    //Note: Supply the values we want to update our record through
    //      contentValues.put(column_name, value to update)

    String [] args = {Integer.toString(shape.getId())};
    result = db.update(SchemeShapes.Shape.TABLE_NAME,contentValues, whereClause: SchemeShapes.Shape.ID + "=?",args) > 0;
    return result;
}
```

Example Code for deleting a Record

```
public boolean deleteTask(int taskId) {
    boolean result;
    SQLiteDatabase db = getWritableDatabase();
    String[] args = {Integer.toString(taskId)};
    result=db.delete(TaskScheme.TABLE_NAME,  whereClause: "_id=?", args) > 0;
    return result;
```

Note: The 3rd parameter of delete() `delete(String table, String whereClause, String[] whereArgs)` stores a list of strings, which replaces the '?' values in the 2nd parameter for record matching. If match then record is deleted. The Boolean returns true when at least 1 record has been deleted.

To perform SQL statements, we need to use execSQL() or rawQuery(). The difference between the two is that **execSQL** has no **SELECT** operation, so we can use it for CREATE, INSERT, UPDATE or DELETE operations (versatile); Whereas **rawQuery** has **SELECT**, and is used to return a **Cursor** result-set, so **rawQuery** is used for filtering a database.

The output type of a query operation is a **Cursor**, which is an SQLite data source for a result set.

Example Code for retrieving all Records

```
public Cursor getAllTasks() {
    SQLiteDatabase db = getReadableDatabase();
    Cursor tasks = db.rawQuery( sql: "select * from " + TaskScheme.TABLE_NAME, selectionArgs: null);
    return tasks;
}
```

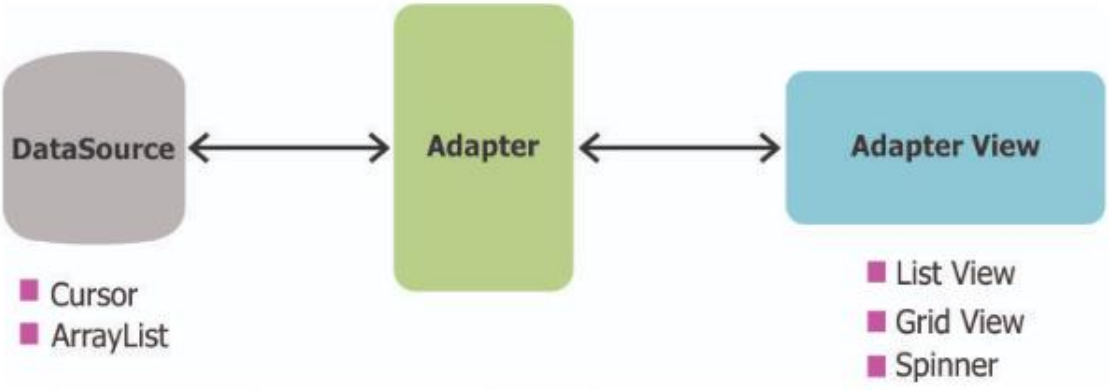
Generic rawQuery Statement :

Cusor returnValue = db.rawQuery(param1, param2);

1. returnValue is a Cursor which is the SQLite data source. The cursor provides read or write access (depending on dB supplied) to the result set returned by a database query.
2. param1 is a String SQL query statement.
3. param2 is a String for selectionArgs, where we may include ?s in the where clause in the query, which will be replaced by the values from selectionArgs. The values will be bound as Strings.

CursorAdapter

We have experienced using a ListView or a RecyclerView to display our data by populating it using an Adapter to a data source. If we want the data to be sourced directly from an SQLite database, we can use a CursorAdapter.



a CursorAdapter exposes data from a Cursor (SQLite data source) to a ListView widget. The cursor must include a column named "_id" or this class will not work. We need to configure two aspects:

1. Which layout template to inflate for an item
2. Which fields of the cursor to bind to which views in the template

Like RecyclerAdapter, we have abstract methods to override called **newView()** and **bindView()**.

newView(): returns a new View to hold the data

bindView(): binds an existing View from **newView()** to hold the data pointed to by **Cursor**.

Context: Interface to applications global information

Cursor: An SQLite data source containing data set returned by a query

ViewGroup: the parent to which the new view is attached to

```
//"Makes a new view to hold the data pointed to by cursor."
@Override
public View newView(Context context, Cursor cursor, ViewGroup viewGroup) {
    return LayoutInflater.from(context).inflate(R.layout.item_delete_edit,
            viewGroup, attachToRoot: false);
}
```

```
@Override
public void bindView(View view, Context context, Cursor cursor) {
    TextView task_name=view.findViewById(R.id.list_task_name_header);
    TextView task_desc=view.findViewById(R.id.list_task_desc_header);
    TextView task_date=view.findViewById(R.id.list_task_date_header);

    task_name.setText(cursor.getString(cursor.getColumnIndexOrThrow(TaskScheme.TASK_NAME)));
    task_date.setText(cursor.getString(cursor.getColumnIndexOrThrow(TaskScheme.TASK_DATE)));
    task_desc.setText(cursor.getString(cursor.getColumnIndexOrThrow(TaskScheme.TASK_DESC)));
    final String id = cursor.getString(cursor.getColumnIndexOrThrow(TaskScheme.ID));
```

Drawing on a View's Background

We can 'draw' on a View's Background by first importing Android's canvas, paint, and colour package.

A View's background is a bitmap which we draw on through a Canvas object. To draw on it, we need a paint object and call onDraw() on our Canvas using our Paint object.

```
public class CustomView extends View {
//    public static Paint paint;
//    public static Canvas canvas;
//    public static ShapeValues[] shapes = null;
//    public static int numberShapes = 0;

    public Paint paint;
    public Canvas canvas;
    public ShapeValues[] shapes = null;
    public int numberShapes = 0;

    public CustomView(Context context) {
        super(context);

        //set some painting and colouring defaults
        paint = new Paint();
        paint.setStyle(Paint.Style.STROKE);
        paint.setAlpha(255);
        paint.setAntiAlias(true);
    }

    //drawing instructions to draw all shapes currently in the db
    @Override
    protected void onDraw(Canvas canvas) {...}
}
```

CustomView

Important class in future App's. It is a View which draws all the shape objects the user creates in the database onto the canvas (screen). It contains a method called **onDraw()** which must **NOT** be called directly but instead, we call **invalidate()**. This instructs Android to redraw the View on the earliest convenience possible whenever the user creates new shape objects.

Exercise 1

In SQLiteOpenHelper:

1. *When does onCreate callback get invoked?*

onCreate callback gets invoked when the database is created for the first time. This is where the creation of tables and the initial population of the tables should happen.

2. *When does onUpgrade callback get invoked?*

Called when the database needs to be upgraded. Should be used where the database file exists but the stored version number is lower than requested in the constructor. The method updates the table schema in the requested version.

```
customView.numberShapes = cursor.getCount();
customView.shapes = shapes;
customView.invalidate();
```

Exercise 2

In the provided code:

a. *What is the datatype of customView?*

View.

b. *What role does customView play in the app?*

customView acts as a specific View Class that the app utilizes in the ViewShape Class to provide a reusable and specific UI implementation. It does this through encapsulating a specific set of functionalities with an easy to use interface, and also by defining a custom set of attributes through the class.

c. *What does invalidate method do?*

Ensuring that the View's onDraw() method is never called directly and should instead use the invalidate() method on the View. This instructs android to redraw the View on the earliest convenience.

Exercise 3

d. In which situation do we have to use ContentValues class?

We use the ContentValues class when we want to insert and update records in our database.

For adding, we need to create a new ContentValues object and pack all the data we want in our table through. put() method, then we call database.insert(ourTable,null,contentValues).

For updating, we do the same thing except we need to identify the record we are to update through an id. We have 3rd param as the where clause containing '?', and 4th param as the arguments supplied to '?'. See above code for example.

Database.update(ourTable, contentValues, whereClause, whereArgs).

Week 8: Databases PART 2, Content Providers and Firebase Database

ContentProvider

Content providers help an application to manage access to data stored by itself, other apps, and provide a way to share data with other apps. They encapsulate the data, and provide mechanisms for defining data security.

We can configure a content provider to allow other applications to securely access and modify the app data.

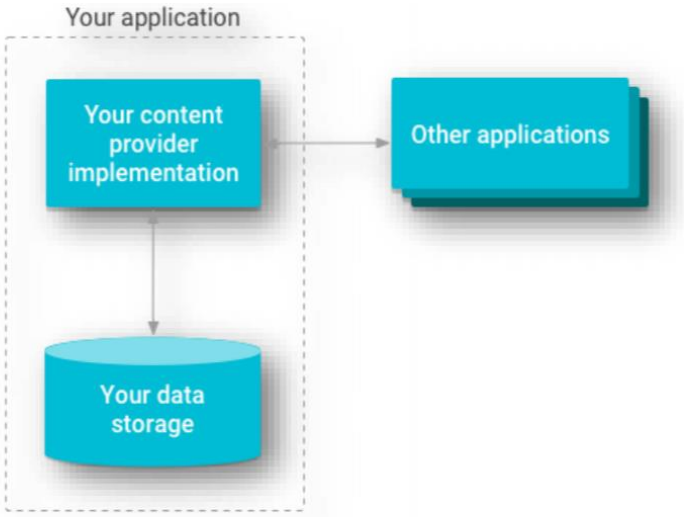


Diagram of how content providers manage access to storage [2].

We use content providers if we plan to share data.

If we don't plan to share data, we may still use them because they provide a nice **abstraction**.

This abstraction allows us to make modifications to our app storage implementation without affecting other existing apps that rely on access to our data. E.g. we might want to swap out an SQLite database for alternative storage.

We look at classes that rely on the CP class:

1. CursorAdapter – seen previous week – an Adapter that exposes data from a **Cursor** to a **ListView**.
2. CursorLoader – a Loader that queries the **ContentResolver** and returns a **Cursor**.

Content providers are used in two scenarios:

1. Implement code to access an existing content provider in another app
2. Create a new content provider in your app to share data with other apps

Accessing a provider

To access data in a content provider, we use the **ContentResolver** object in our app's **Context** to communicate with the provider as a *client*. The **ContentResolver** object communicates with the provider object.

The provider object receives data requests from clients, performs the requested action, and returns the results.

The **ContentResolver** object is used to access the content provider and contains methods for basic "CRUD" functionality. An app can obtain a reference to its **ContentResolver** by calling **getContentResolver()**.

Note: There can be multiple **ContentProviders** for an app, but an app can only have one **ContentResolver**.

Common Implementation

A common pattern for accessing a **ContentProvider** from our UI is to use a **CursorLoader** to run an asynchronous query in the background. The **Activity** or Fragment in the UI calls the **CursorLoader**, which in turn gets the **ContentProvider** using the **ContentResolver**. This allows the UI to continue to be available to the user while the query is still running.

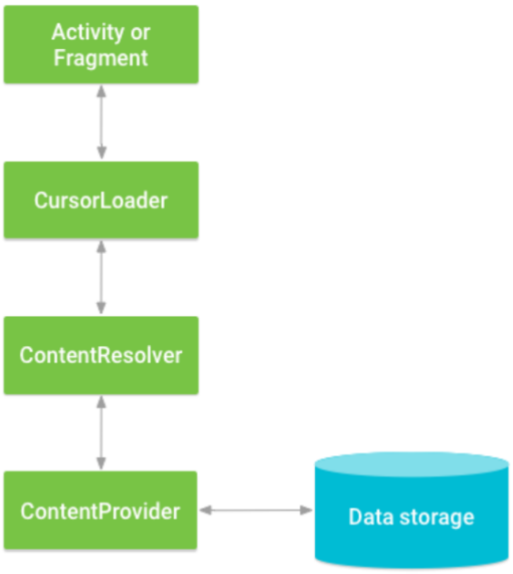


Figure 2. Interaction between ContentProvider, other classes, and storage.

Example Database and Querying

Table 1: Sample user dictionary table.

word	app id	frequency	locale	_ID
mapreduce	user1	100	en_US	1
precompiler	user14	200	fr_FR	2
applet	user2	225	fr_CA	3
const	user1	255	pt_BR	4
int	user5	100	en_UK	5

Each row represents a record. Each column represents some data in the record

The _ID column serves as the primary key column that the provider automatically maintains.

To retrieve a list of words and their locales from the user dictionary provider, we call **ContentResolver.query().** The query() method calls the **ContentProvider.query()** method defined by the provider.

The following shows how to use the **ContentResolver.query()** call

```
// Queries the user dictionary and returns results
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,   // The content URI of the words table
    mProjection,                        // The columns to return for each row
    mSelectionClause,                   // Selection criteria
    mSelectionArgs,                     // Selection criteria
    mSortOrder);                        // The sort order for the returned rows
```

The following shows how to match the arguments to query (Uri, projection, selection, selectionArgs, sortOrder)

Table 2: Query() compared to SQL query.

query() argument	SELECT keyword/parameter	Notes
`Uri`	`FROM table_name`	`Uri` maps to the table in the provider named *table_name*.
`projection`	`col,col,col,...`	`projection` is an array of columns that should be included for each row retrieved.
`selection`	`WHERE col = value`	`selection` specifies the criteria for selecting rows.
`selectionArgs`	(No exact equivalent. Selection arguments replace ? placeholders in the selection clause.)	
`sortOrder`	`ORDER BY col, col,...`	`sortOrder` specifies the order in which rows appear in the returned `Cursor`.

Content URI

A Content URI is a utility class that identifies data in a provider. They include the symbolic name of the entire provider (its **authority**), and a name that points to a table (a **path**).

In the previous dictionary database, the constant **CONTENT_URI** contains the URI of the user dictionary's "words" table.

The full URI is **"content://user_dictionary/words"** where:

1. user_dictionary string is the provider's authority
2. words is the table's path
3. content:// is always present, and identifies that this is a content URI.

Creating a Content Provider

To create, we need to define the following

1. Design the raw storage for our data which includes both
    a. File data – data going into files
    b. "Structured data" – data going into a data structure that's compatible with tables of rows and columns, common one is an SQLite database.
2. Define a concrete implementation of the **ContentProvider** class and its required methods.

- Override these functions:

onCreate()	which is called to initialize the provider
query(Uri, String[], Bundle, CancellationSignal)	which returns data to the caller
insert(Uri, ContentValues)	which inserts new data into the content provider
update(Uri, ContentValues, String, String[])	which updates existing data in the content provider
delete(Uri, String, String[])	which deletes data from the content provider
getType(Uri)	which returns the MIME type of data in the content provider

Note: Insert, update, and delete methods should warn listening **CursorLoaders** of a change to the data.

3. Define the provider's **authority** string, its **content URI**s, and column names.

Declaring a ContentProvider in our App

We add the <provider> element to the app manifest

From database_2: AndroidManifest.xml

```
<provider
    android:name=".provider.ShapesProvider"
    android:authorities="edu.monash.fit2081.db.provider"
    android:exported="true"/>
```

android:name specifies the class used to instantiate the provider.

android:authorities specifies the unique ID of the provider

android:exported specifies the access **ContentResolvers** will have to the provider

Creating the UriMatcher definition

Urimatcher is a utility class which aids in matching URIs in content providers

```
private static final UriMatcher sUriMatcher = createUriMatcher();

private static UriMatcher createUriMatcher() {

    final UriMatcher uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    final String authority = SchemeShapes.CONTENT_AUTHORITY;
    uriMatcher.addURI(authority, SchemeShapes.Shape.PATH_VERSION, SHAPES);
    uriMatcher.addURI(authority, SchemeShapes.Shape.PATH_VERSION + "/#", SHAPES_ID);

    return uriMatcher;
}
```

Accessing the ContentResolver and Performing actions (Methods)

1. **Get the resolver**
   a. From a fragment

```
ContentResolver resolver = getActivity().getApplicationContext().getContentResolver();
```

   b. From an activity

```
ContentResolver resolver =getContentResolver();
```

2. **Call the query method to retrieve some data in the resolver**

Example code in Week8 Database2 App

```
public void getCircleData(View view) {
    Cursor cursor;
    ContentResolver resolver = getApplicationContext().getContentResolver();
    String [] projection = new String[] {SchemeShapes.Shape.ID, SchemeShapes.Shape.SHAPE_TYPE,
                            SchemeShapes.Shape.SHAPE_RADIUS};
    String selection = SchemeShapes.Shape.SHAPE_TYPE + "=?";
    String [] args = {"Circle"};
    String sortOrder = null;

    cursor = resolver.query(SchemeShapes.Shape.CONTENT_URI, projection,
            selection, args, sortOrder);

    int max = 0,total = 0, cur_radius;
    cursor.moveToFirst();

    while (cursor.isAfterLast() == false) {
        cur_radius = cursor.getInt( columnIndex: 2);

        if (cur_radius > max)
            max = cur_radius;

        total += 1;
        cursor.moveToNext();
    }
    shapeNum.setText("There are " + Integer.toString(total) + " Circles.");
    radiusNum.setText("Max Radius is " + Integer.toString(max) + ".");
```

3. Call whatever else methods we would want to perform (see list above) but make sure we supply the **CONTENT_URI** so Android will know where to find the Content Provider Dataset.

CursorLoader

A loader that queries the **ContentResolver** and returns a **Cursor**.

It only queries (select) the data, there is no insert, delete or update.

We use it because it performs queries on a separate thread, monitors the data source for changes and updates the UI, and integrates easily with the life cycle of an Activity and Fragment.

Using CursorLoader

We use CursorLoader by letting our Activity or Fragment implement the **LoaderManager.LoaderCallbacks<Cursor>** interface

- We initialize our data load operation by calling **LoaderManager.initLoader()**, this creates the loader, calling **onCreateLoader()**, loading the data, and calling **onLoadFinished()** asynchronously when done.
- Both activities and Fragments can get their LoaderManager object by calling **getLoaderManager()**. Hence we call **getLoaderManager().initLoader().**
- **initLoader()** takes 3 parameters, the ID of the loader, the data needed to create the loader, can be null, and the activity/Fragment handling the loader, so just call **this**. This data is passed into **onCreateLoader()**.

Example initializing a loader for a Fragment

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    customView = new CustomView(getContext());

    getLoaderManager().initLoader(0, null, this); //create loader if doesn't exist, starts the loader

    return (customView);
}
```

ID of the loader

For activities, **initLoader()** should be called in **onCreate()**, for Fragments, **initLoader()** should be called in **onCreateView()**.

We need to define 3 methods for the interface:

- `Loader<T> onCreateLoader(int id, Bundle args)`
- `void onLoadFinished(Loader<T>, T data)`
- `void onLoaderReset(Loader<T> loader)`

onCreateLoader() : instantiates and return a new Loader for the given ID.

onLoadFinished(): called when a loader has finished its load, this method is used to update the UI after the data has been loaded.

onLoadReset(): removes any references to old data that may no longer be available. It disassociates a loader with a given ID and allows it to be recreated, resulting in **onCreateLoader()** being called again, allowing a new CursorLoader object to be made which can contain a different query for a given ID.

```
public void onLoaderReset(Loader<Cursor> loader) {
    // This is called when the last Cursor provided to onLoadFinished()
    // above is about to be closed.  We need to make sure we are no
    // longer using it.
    mAdapter.swapCursor(null);
}
```

Example Code for CursorLoader in StudentRegistryApp

The ListStudents Fragment class of the app implements the **LoaderManager.LoaderCallacks<Cursor>** to fetch all the students from the DB and updates the ListView.

```
@NonNull
@Override
public Loader<Cursor> onCreateLoader(int id, @Nullable Bundle args) {
    CursorLoader cursorLoader = new CursorLoader(getActivity(),
                                                 SchemeClass.Student.CONTENT_URI,
                                                 SchemeClass.Student.PROJECTION,
                                                  selection: null,
                                                  selectionArgs: null,
                                                  sortOrder: null
    );
    return cursorLoader;
}

@Override
public void onLoadFinished(@NonNull Loader<Cursor> loader, Cursor data) {
    studentCursorAdaptor.swapCursor(data);
}

@Override
public void onLoaderReset(@NonNull Loader<Cursor> loader) { super.onResume(); }
```

Firebase Database

Google's cloud-based real-time database service. Allows data to be synced in real-time. It is a NO-SQL database, there are no tables, columns, rows etc. Data is stored in root-child relationships.

Implementing a Firebase Database

1. In our build.gradle list of dependencies, we implement:

```
implementation 'com.google.firebase:firebase-core:16.0.6'
implementation 'com.google.firebase:firebase-database:16.0.5'
```

2. Create a reference to our FirebaseDatabase:

DatabaseReference mRef = FirebaseDatabase.getInstance().getReference()

3. Insert the config file google-sevices.json into our app root directory

Inserting data into a child from our reference

1. Reference our child called "etc" or create a child called "etc" in the database if it doesn't exist through:
    a. DatabaseReference mCondition = mRef.child("etc")

    Note: This is a child of mRef database. "etc" becomes a root-child of mRef.

2. push some data into the child through:
    a. mCondition.push().setValue(someData)

DatabaseReference

A starting point for all database operations, this allows us to read, write, and create new DatabaseReferences. Used to refer to the location of a specific node in our database structure.

DataSnapshot

DataSnapshot is a special object from Firebase which is an efficiently-generated immutable copy of the data. It is created when we add a ChildEventListener to a child, and supply listener callbacks which generate a DataSnapshot. It is used to retrieve data from the node whenever a listener event is triggered from the database.

addChildEventListener

a method which allows us to instantiate a ChildEventListener() interface to our child, allowing us to override several listener callbacks when changes to our child data occur. A DataSnapshot object is invoked, which allows us to perform actions such as retrieving the value through **dataSnapshot.getValue().**

Some examples of listener callbacks:

1. **onChildAdded(@NonNull DataSnapshot dataSnapshot, @Nullable String s)** – This is invoked when data has just been pushed to a root child.
2. There are other listener methods such as onChildChanged, onChildRemoved, etc.

Example Week 8 Firebase App Analysis

1. Contains a custom implementation of ArrayAdapter containing “ForecastStatus” objects, with the adapter called ForecastAdapter.
2. Has a ListView View layout which is binded to the ForecastAdapter.

```
ListView listView = findViewById(R.id.list_status);
ArrayList<ForecastStatus> data = new ArrayList<ForecastStatus>();
itemsAdapter = new ForecastAdapter ( context: this,  data);
listView.setAdapter(itemsAdapter);
```

3. Contains DatabaseReferences variables to insert data into the root child of our database called “status”.

```
public void btn1Handler(View view) {
    ForecastStatus forecastStatus=new ForecastStatus(getTimeStamp(), status: "Sunny");

    DatabaseReference mRef= FirebaseDatabase.getInstance().getReference();
    DatabaseReference mCondition = mRef.child("status");
    mCondition.push().setValue(forecastStatus);
}
```

4. Contains Listener callbacks for child’s we want to listen which provides a DataSnapshot which allows us to perform actions to manipulate data.

```
@Override
protected void onStart() {
    super.onStart();
    mCondition = mRef.child("status");
    mCondition.addChildEventListener(new ChildEventListener() {
        @Override
        public void onChildAdded(@NonNull DataSnapshot dataSnapshot, @Nullable String s) {
            data.add(dataSnapshot.getValue(ForecastStatus.class));
            itemsAdapter.notifyDataSetChanged();
        }

        @Override
        public void onChildChanged(@NonNull DataSnapshot dataSnapshot, @Nullable String s) {}
```

Exercise 1

outputParam = new CursorLoader(param1, param2, param3, param4, param5, param6)

Briefly explain the datatype and role of the input and output parameters of the given statement. The statement is how a query() method works in a ***ContentResolver***

param1: parent activity of type **Context**

param2: table to query of type **Uri**

param3: projection to return of type **String[]**

param4: selection clause of type **String**

param5: selection arguments of type **String[]**

param6: sort order of type **String**

outputParam : Type is **CursorLoader**, which loads queries towards the **ContentResolver** and returns a **Cursor**.

Exercise 2

a. Explain what a UriMatcher does.

it is a Utility class to help with matching URIs in content providers - for classes that needs to respond to lots of URIs.

b. With respect to content provider, how does query method serve multiple URIs

query() method can specify the **Uri** to target from the **table_name**, the **Uri** maps to the table in the provider named **table_name**.

Exercise 3

A CursorLoader's load is asynchronous so we don't know when it finishes; But we can't use its data until its finished loading it. How is this problem resolved?

There is a callback function the moment the data is finished loading called onLoadFinished(), this allows us to define what we want to do once the data is finished loading, such as refreshing the UI with the loaded data.

Exercise 4

resolver.insert(param1, param2);

1. *What is the role and datatype of param1 and param2?*

param1: Uri - maps to the table in the provider

param2: ContentValues - Specifies the data we want to insert into our record, to insert data, we call contentValues.put(column_name, value)

2. *How did you declare and initialize object resolver?*

We create a variable of type **ContentResolver** and depending if we want it inside a fragment or activity, we call **getContentResolver()** for an Activity, and **getActivity().getApplicationContext().getContentResolver()**

Exercise 5

Assume you have an application that stores its data in Google Firebase.

a. Write a piece of code that pushes the string “Summer-2019” into Firebase root child “week8”.

```
mRef= FirebaseDatabase.getInstance().getReference();
child = mref.Child("week8");
child.push().setValue("Summer-2019");
```

b. The same app has a TextView with ID=”tv”. Write a piece of code that updates the content of “tv” with the value of each new child added to the child “week8”.

```
mCondition = mRef.child("week8")
TextView tv = findViewById(R.id.tv);
mCondition.addChildEventListener(new ChildEventListener() {
    @Override
   public void onChildAdded(@NonNull DataSnapshot dataSnapshot, @Nullable
Strings) {
        tv.setText(dataSnapshot.getValue().toString());
}
}
```

Week 9: Google Maps API, Web Services, WebView UI component, AsyncTasks Class, JSON Format

Google Maps

A set of API classes and interfaces grouped together in the com.google.android.gms.maps package. The main class of the API is **GoogleMap**, which is the entry point for all methods related to the displayed map.

We use a class in Google API called **SupportMapFragment**, this class inflates the map image into the XML Fragment and then supports all the functionality we expect from a Google map. We call **onMapReady()** to retrieve a reference to **GoogleMap**.

Web Services

Function calls made using HTTP in order to perform CRUD operations on remote data. We use standard HTTP vocab like GET, POST, PUT, and DELETE to signal CRUD data operations. In our App, we make the following HTTP GET request: "https://restcountries.eu/rest/v2/name/" + selectedCountry", which returns data for our selected country in JSON format. We can then use this data in our app for manipulation.

The main 4 classes we use in order of callbacks are:

1. HttpsURLConnection – Retrieves a HttpsURLConnection **object** representing a connection to the URL's sever
2. InputStream – opens a data "pipe" to the server if the connection object is OK
3. InputStreamReader – translates character data into the "pipe"
4. JsonReader – retrieves the "pipe" and utilize JsonReader methods to retrieve data in the pipe as required

WebView – a View widget that displays Web pages

It does not include any features of a fully developed web browser, however by making some custom settings and coding, such as enabling JavaScript, adding an address bar etc, the WebView can offer similar features of a browser.

In most cases, using a standard web browser, like Chrome is better to deliver content, this is done by invoking a browser with an **Intent**. However, a **WebView** can offer a more specifically-designed environment for out App.

Implementing a WebView in an App

1. Add a WebView to our app layout XML file
2. call .loadUrl() on our WebView reference in Java.

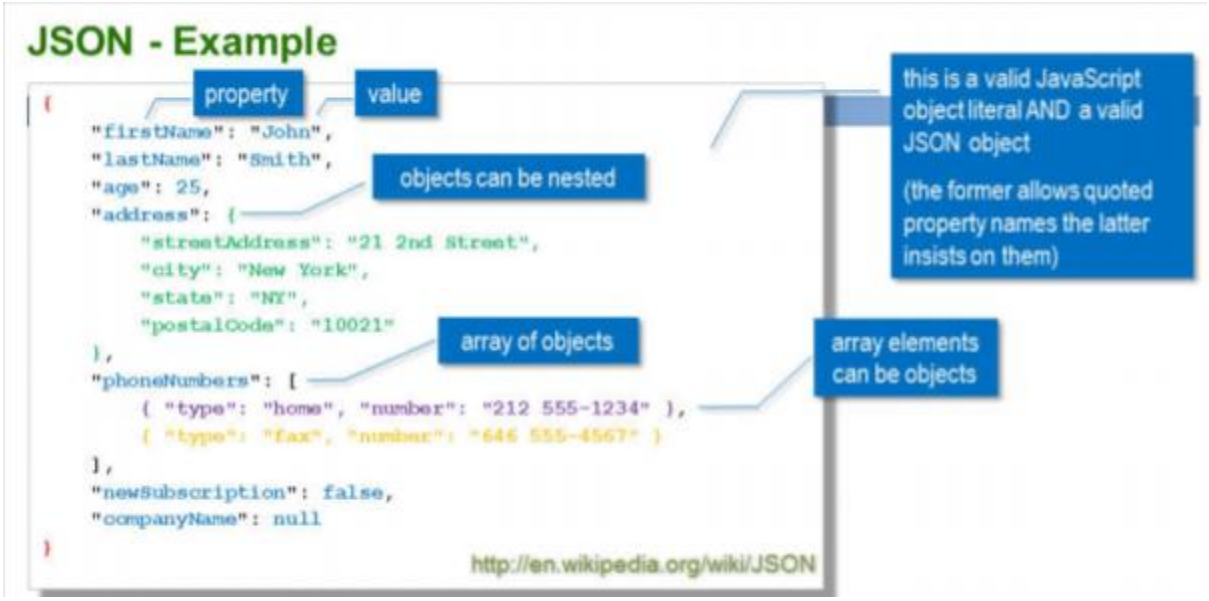
Example in Week 9 CountryInfo App

```
@Override
protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_webview);
    getSupportActionBar().setTitle(R.string.title_wikipedia_details);

    country = getIntent().getStringExtra( name: "country");

    myWebView = findViewById(R.id.webview);
    myWebView.setWebViewClient(new WebViewClient());
    myWebView.loadUrl("https://en.wikipedia.org/wiki/" + country);
}
```

JSON – Lightweight data format popular for data transfer across the Web.

Like JavaScript object literal notation, consisting of:

1. Objects – Comma separated list of key/value pairs enclosed in {…}
2. Arrays – comma separated list enclosed in […]
3. Literal value – Strings, Numbers, Booleans, null



AsyncTask – Enables proper and easy use of the UI thread. This class allows us to safely update UI components by running background threads whose results are published on the UI thread.

Defining an Async Class in Java

We do this by creating a class and extending the AsyncTask class which takes 3 generic types called **Params, Progress and Result**.

There are 3 interface methods to implement which utilize the above generic types:

1. First is the **Params** of the type of task to pass into **execute()** to perform **doInBackground().**
2. Second is the **Progress** data to pass during **publishProgress()** for **onProgressUpdate()**.
3. Third is the return **Result** after **doInBackground()** is finished executing, which will be the input to **onPostExecute()**.

Hence, there are 2 interface methods we need to implement + 1 optional method

1. **doInBackground()** : performs the task in the background thread and returns some data.
2. **onPostExecute()** : Automatically invoked after **doInBackground()**. It retrieves the output from **doInBackground()**, and runs in the UI thread, allowing us to update our UI components safely.
3. **Optional**: **onProgressUpdate()**: Invoked in the UI thread when the system calls **publishProgress()** during **doInBackground()**, it notifies the any form of progress in the UI while the background computation is still executing.

App Example

```
private class GetCountryDetails extends AsyncTask<String, String, CountryInfo>

    @Override
    protected CountryInfo doInBackground(String... params) {...}

    @Override
    protected void onPostExecute(CountryInfo countryInfo) {...}
}
```

Exercise 1

a. *To manipulate a GoogleMap a reference to it is required. From where is this reference obtained?*

The reference is obtained from SupportMapFragment from the Google Maps API package "com.google.android.gms.maps"

b. *What does a Geocoder do exactly?*

the class converts area addresses into geographic coordinates which we can use to place markers on a map, or position the map

c. *AsyncTask is a Generic class. What does this mean?*

It means that the AsyncTask class can be parameterized over types. This enables stronger type checking at compile time and provide programmers a set of related methods to use safely.

Exercise 2

Describe in detail what is the effect of the presented code

```
addresses = geocoder.getFromLocation(point.latitude, point.longitude, 1);
```

```
Snackbar.make(mapFragment.getView(), msg, Snackbar.LENGTH_LONG).
    setAction("Details", (addresses.size() == 0) ?
    (new ActionOnClickListener(selectedCountry)) : null).show();
```

address is a list of type <Address> which retrieves locations from the getFromLocation method provided by Geocoder, it accepts a latitude and longitude and returns a list of addresses.

Then there is a Snackbar which has a message “Details” which can clicked when there are at least 1 address present for a selected country.

Exercise 3

Point your browser at https://restcountries.eu/rest/v2/name/India

a. *At the top level and first level of nesting describe the structure of the returned JSON*

Top level is an array of 2 objects, One for British Indian Ocean Territory, another is for India.

The next level describes the properties of the object, with properties such as "name", "alpha2Code" etc.

b. *Describe the detailed structure of the value of the "languages" property for India*

The property "languages" has an array, which contains two objects consisting of key/value pairs. the objects have 4 properties, "iso639_1", "iso639_2", "name", "nativeName". All property and values have type String.

Exercise 4

```
private class GetCountryDetails extends AsyncTask<String, String, CountryInfo> {

    @Override
    protected CountryInfo doInBackground(String... params) {...}

    @Override
    protected void onPostExecute(CountryInfo countryInfo) {...}
}
```

a. *What is the role of the first generic type in the GetCountryDetails class?*

The data type of the input to be sent to the task, the input parameter to **doInBackground()**.

b. *What is the role of the third generic type in the GetCountryDetails class?*

The return data type of **doInBackground()**.

c. *What is the name of the AsyncTask instance?*

GetCountryDetails.

Exercise 5

Continuing from last exercise…

a. What is the meaning of String…params?

It specifies the input parameters passed to doInBackground, in this case, they can be represented as an array of strings, or as a sequence of string arguments.

b. How can input parameter be accessed in doInBackground?

The input parameter can be accessed by calling the index of the parameter name. e.g. to get the first value, we can call params[0].

c. When and on what thread does doInBackground execute?

Background thread and executes when the caller calls the .execute method, which starts the task.

d. When and on what thread does onPostExecute execute?

UI thread and executes when doInBackground has finished processing and returns the output to it. It defines what happens when the task is complete.

Additional Notes

Week 10: App Bar & Options Menu Recap, String Array Resources, Touch Processing, Canvas Recap, App Bar Icons

App Bar and Options Menu

Discussed in Week 4 and 5 Content.

String Array Resources

Allows us to define an array of strings to be referenced from our App. We store this in our res/values/filename.xml, and reference it in Java by **R.array.string_array_name.** The syntax is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="planets_array">
        <item>Mercury</item>
        <item>Venus</item>
        <item>Earth</item>
        <item>Mars</item>
    </string-array>
</resources>
```

Elements:

1. <resources> - This is **required**, it acts as the root node.
2. <string-array> - Defines an array of strings. Contains one or more <item> elements
    a. Has a name – which acts as the name of array, this name acts as the resource ID to reference.
    b. Has <item>'s – which is a string which can include styling tags. The value can be a reference to another string resource as well.

To reference the above example string array, we call:

```
Resources res = getResources();
String[] planets = res.getStringArray(R.array.planets_array);
```

Touch Processing & Gesture Detection

Two steps, First is detecting the touch event and what its gesture is. Second is interpreting the data to see if it meets the criteria for any of the gesture that our app supports.

Capturing and Handling Touch Events

When a user touches the screen, this triggers the **onTouchEvent(MotionEvent event)** callback on the View that received the touch events and supplies a MotionEvent object which provides information about the touch event.

The gesture starts when the user first touches the screen, continuing as the system tracks the position of the user's finger(s), and ends by capturing the final event of the user's fingers leaving the screen.

MotionEvent Class

MotionEvent objects are created during touch event callbacks and provide users with methods to query the position and properties of the current pointers, such as **getX(int), getY(int)**.

Each pointer has a unique id that is assigned when it first goes down (indicated by **ACTION_DOWN** or **ACTION_POINTER_DOWN**). a pointer id remains valid until the pointer eventually goes up (**ACTION_UP** or **ACTION_POINTER_UP**) or when the gesture is cancelled (**ACTION_CANCEL**).

Most of these methods accept the pointer index as a parameter rather than the id.

MotionEvent Batching

For Efficiency, motion events with **ACTION_MOVE** may batch together multiple movement samples within a single object.

The **MotionEvent** delivered to **onTouchEvent()** provides the details of every interaction. Our app can use the data provided by **MotionEvent** to determine if a gesture it cares about happened.

Multi-Touch Gestures

Multi-touch gesture is when multiple pointers (fingers) touch the screen at the same time. The following touch events generate when multiple pointers touch the screen:

1. ACTION_DOWN – First pointer that touches the screen. pointer index is always 0 in the **MotionEvent**.
2. ACTION_POINTER_DOWN – For extra pointers that enter the screen after the first, data for this pointer is at the index returned by **getActionIndex()**
3. ACTION_MOVE – For when a change has occurred during a press gesture.
4. ACTION_POINTER_UP – For when a non-primary pointer goes up.
5. ACTION_UP – For when the last pointer leaves the screen.

We keep track of individual pointers within a **MotionEvent** via each pointer's index and ID

1. Index – a **MotionEvent** stores information about each pointer in an array, the index is its position within this array
2. ID – each pointer has an ID that stays persistent across touch events to allow tracking of individual pointer across the entire gesture.

The order of pointers in a motion event is undefined, Thus the index of a pointer can change from one event to the next, but ID is guaranteed to remain constant. We call **getPointerId()** to obtain a pointer's ID for a given index. We can also call **findPointerIndex()** to obtain the pointer index for a given ID.

Example App which allows us to track the first primary pointer

```
private int mActivePointerId;

public boolean onTouchEvent(MotionEvent event) {
    ...
    // Get the pointer ID
    mActivePointerId = event.getPointerId(0);

    // ... Many touch events later...

    // Use the pointer ID to find the index of the active pointer
    // and fetch its position
    int pointerIndex = event.findPointerIndex(mActivePointerId);
    // Get the pointer's current position
    float x = event.getX(pointerIndex);
    float y = event.getY(pointerIndex);
    ...
}
```

Get a MotionEvent's Action for Multiple pointers

We can use the MotionEvent method **getActionMasked()**, or the compatibility version **MotionEventCompat.getActionMasked()** to retrieve the action of a **MotionEvent**.

MotionEvent is an array of pointers. We can call **getActionIndex()** to return the index of the pointer associated with the action.

Example App to detect the **MotionEvent's** action for multiple pointers

```
public boolean onTouch(View view, MotionEvent motionEvent) {
    int count = motionEvent.getPointerCount();
    int typeEvent = motionEvent.getActionMasked();
    switch (typeEvent) {
        case MotionEvent.ACTION_DOWN:
            // the down event of the primary pointer (first)
            break;
        case MotionEvent.ACTION_UP:
            // the up event of the primary pointer
            break;
        case MotionEvent.ACTION_POINTER_DOWN:
            int idx=motionEvent.getActionIndex();
            pntrID=motionEvent.getPointerId(idx);
            float x=motionEvent.getX(idx);
            float y=motionEvent.getY(idx);

            break;
        case MotionEvent.ACTION_POINTER_UP:
            int index=motionEvent.findPointerIndex(pntrID);
            break;
    }

    gestureDetector.onTouchEvent(motionEvent);
    scaleGestureDetector.onTouchEvent(motionEvent);
    return true;
}
```

Capturing touch events for an Activity or View

To intercept touch events in an Activity or View, we override the **onTouchEvent()** callback. **getActionMasked()** extracts the action the user performed from the **MotionEvent**, letting us to know what action has occurred.

```
public class MainActivity extends Activity {
...
// This example shows an Activity, but you would use the same approac
// you were subclassing a View.
@Override
public boolean onTouchEvent(MotionEvent event){

    int action = MotionEventCompat.getActionMasked(event);

    switch(action) {
        case (MotionEvent.ACTION_DOWN) :
            Log.d(DEBUG_TAG,"Action was DOWN");
            return true;
        case (MotionEvent.ACTION_MOVE) :
            Log.d(DEBUG_TAG,"Action was MOVE");
            return true;
        case (MotionEvent.ACTION_UP) :
            Log.d(DEBUG_TAG,"Action was UP");
            return true;
        case (MotionEvent.ACTION_CANCEL) :
            Log.d(DEBUG_TAG,"Action was CANCEL");
            return true;
        case (MotionEvent.ACTION_OUTSIDE) :
            Log.d(DEBUG_TAG,"Movement occurred outside bounds " +
                    "of current screen element");
            return true;
        default :
            return super.onTouchEvent(event);
    }
}
```

Capturing touch events for a single View

An alternate to **onTouchEvent()**, we can attach a **View.OnTouchListener** object to any View object using the **setOnTouchListener()** method.

```
View myView = findViewById(R.id.my_view);
myView.setOnTouchListener(new OnTouchListener() {
    public boolean onTouch(View v, MotionEvent event) {
        // ... Respond to touch events
        return true;
    }
});
```

Note: Do not mix **OnTouchListener/onTouch()** and **onTouchEvent()**

1. **View.onTouchListener** is an interface for a callback to be invoked when a touch event is dispatched to the View. The callback will be invoked before the touch event is given to the view.
   i.e. **onTouch** in **OnTouchListener** fires before **onTouchEvent** is given to the View.

So, both ways (overriding **onTouchEvent()** in our **Activity** or **View**, or Attaching an **onTouchListener()** to our **View**, and overriding **onTouch()** callback) above allows us to process simple gestures.

However, **The Main problem with the above is**, it is extremely tedious to interpret a sequence of events, there are too many gestures to handle. We will need to process the touch events ourselves and define complex logics.

GestureDetector Class

To solve this, Android provides the **GestureDetector** class for detecting common gestures. Examples of supported gestures include **onDown(), onLongPress(), onFling**. We can combine **GestureDetector** with **onTouchEvent()** or **onTouch()**.

To use the class, we instantiate a **GestureDetector** object:

```
gestureDetector=new GestureDetector( context: this,new MyGestureClass());
```

First param is the Activity, and second param must be an interface or class which implements gestures. There are 3 interfaces and 1 class we can supply:

`interface`
`GestureDetector.OnContextClickListener`
The listener that is used to notify when a context click occurs.
`interface`
`GestureDetector.OnDoubleTapListener`
The listener that is used to notify when a double-tap or a confirmed single-tap occur.
`interface`
`GestureDetector.OnGestureListener`
The listener that is used to notify when gestures occur.
`class`
`GestureDetector.SimpleOnGestureListener`
A convenience class to extend when you only want to listen for a subset of all the gestures.

Then we supply our GestureDetector object inside our **Listener** in order to make it possible for our **GestureDetector** object to listen to events.

we override the View or Activity's Listener method (**onTouchEvent** or **onTouch** depending on Activity or View) and pass along all observed events to the detector instance.

We return true for methods we want our app to include (to consume it), otherwise we return false.

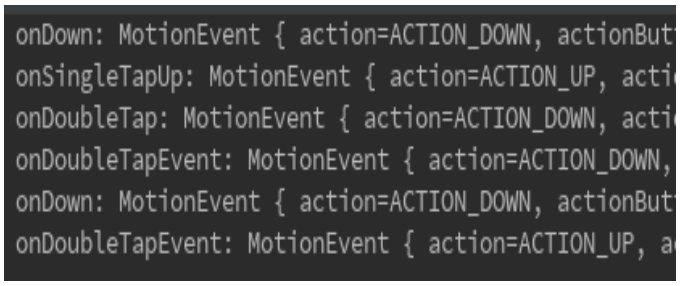
We will use something called a **convenience class** later which essentially is a Java Class that implements the interface **GestureDetector.SimpleOnGestureListener**. We then extend the **convenience class**, allowing us to solve the issue of not needing method definitions we do not need (but have access to all methods).

GestureDetector Class Coding Example

```
public class MainActivity extends Activity implements
        GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener{

    private static final String DEBUG_TAG = "Gestures";
    private GestureDetectorCompat mDetector;

    // Called when the activity is first created.
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Instantiate the gesture detector with the
        // application context and an implementation of
        // GestureDetector.OnGestureListener
        mDetector = new GestureDetectorCompat(this,this);
        // Set the gesture detector as the double tap
        // listener.
        mDetector.setOnDoubleTapListener(this);
    }
```

```
@Override
public boolean onTouchEvent(MotionEvent event){
    if (this.mDetector.onTouchEvent(event)) {
        return true;
    }
    return super.onTouchEvent(event);
}

@Override
public boolean onDown(MotionEvent event) {
    Log.d(DEBUG_TAG,"onDown: " + event.toString());
    return true;
}

@Override
public boolean onFling(MotionEvent event1, MotionEvent event2,
        float velocityX, float velocityY) {
    Log.d(DEBUG_TAG, "onFling: " + event1.toString() + event2.toString());
    return true;
}

@Override
public void onLongPress(MotionEvent event) {
    Log.d(DEBUG_TAG, "onLongPress: " + event.toString());
}

@Override
public boolean onScroll(MotionEvent event1, MotionEvent event2, float distanceX,
        float distanceY) {
    Log.d(DEBUG_TAG, "onScroll: " + event1.toString() + event2.toString());
    return true;
}

@Override
public void onShowPress(MotionEvent event) {
    Log.d(DEBUG_TAG, "onShowPress: " + event.toString());
}
```

What the App does is it will log our tap movements and tell us what kind of tap gesture has occurred. For example, if we double tap on our Android emulator, the following callbacks will emit in order:



App Bar Icons

Graphical elements placed in the Action/App Bar representing individual action items.

Mini-Exercise

1. **MotionEvents** of which action type need a coded response to draw a rectangle of appropriate dimensions by drag-drawing on the “canvas”?
   Only ACTION_DOWN and ACTION_UP
2. **MotionEvents** of which action type need a coded response to draw a freehand line of small circles by drag-drawing on the “canvas”?
   Only ACTION_MOVE

Exercise 1

a. *Explain how to set up a View so touch events on it can be detected and responded to.*

Instantiate a View object in the onCreateView() callback and pass the context to its constructor to use as a reference to the current Activity’s View.

We attach a View.OnTouchListener on our View by calling setOnTouchListener on the View object. We then override the interface method onTouch() , supplying the View and the MotionEvent. Utilizing the MotionEvent, we can detect the type of touch event that has occurred and code what happens during a touch event.

b. *Explain another way*

Another option to detect touch events is overriding the **onTouchEvent()** callback method. The previous touch event method will execute before **onTouchEvent()**, hence it is required to return false on **onTouch()**.

Exercise 2

a. *When would the touch event identified by MotionEvent.ACTION_POINTER_DOWN occur?*

the touch event identified by that constant will occur when extra pointers enter then screen after the first primary pointer.

b. *Why does a pointer have an index and an id?*

The MotionEvent class provides many methods to query the position and other properties of a pointer. There are methods such as getX(), getY(), getPointerId() etc, but most of these methods accept the pointer index as a parameter rather than the ID.

However the order of pointers in a MotionEvent is undefined, so the index of a pointer can change from one event to the next, but ID is guaranteed to remain consistent.

Hence, a pointer should have both an index and an ID to be effective, and we use **getPointerId()** and **findPointerIndex()** accordingly to find a pointers respective pointer and id.

Exercise 3

Why is there no MotionEvent.ACTION_POINTER_MOVE action type?

There is no ACTION_POINTER_MOVE because even with multiple pointers used, the ACTION_MOVE action isn't pointer specific. The motion contains the most recent point, as well as any points since the last down or move event and that's why there is no need for a pointer specific action like ACTION_POINTER_MOVE. E.g. there are two or more fingers used, if you move all fingers, you don't get two or more events processing.

Another option instead is to get the position of the pointer that matters by tracking its pointer ID.

Exercise 4

```
int x = (int) event.getX(); int y = (int) event.getY();
int dX, dY;

int action = MotionEventCompat.getActionMasked(event);
switch (action){
    case MotionEvent.ACTION_DOWN:
        mFirstTouchX = x; mFirstTouchY = y;
        break;
    case MotionEvent.ACTION_MOVE:
        if (selectedShapeDrawing.equals("Line")) {
            dX = 5; dY = 5;
            storeShape( shape: "Circle", x, y, dX, dY);
        }
        break;
    case MotionEvent.ACTION_UP:

        if (!selectedShapeDrawing.equals("Line")) {
            dX = x - mFirstTouchX; dY = y - mFirstTouchY;
            storeShape(selectedShapeDrawing, mFirstTouchX, mFirstTouchY, dX, dY);
        }
        break;
}
```

a. *Which variable is referencing a MotionEvent object?*

   event is the variable.

b. *Which case's code would get executed the most under typical drag-draw use? Explain.*

   MotionEvent.ACTION_MOVE gets run the most because ACTION_DOWN occurs on touch down and ACTION_UP occurs on touch up. ACTION_MOVE constantly tracks all the movement that occurred between ACTION_DOWN and ACTION_UP. When the user's finger drags across the screen, onTouch will execute repeatedly (OS runs highest frequency it can manage) to track the movement.

c. *Explain what the second case is doing (user view).*

   When ACTION_MOVE case is run, it checks if the selected shape is a line. If it is a line, it will constantly store a circle with radius of 5 on the x and y coordinates in the screen. This gets repeated every time the user drags on the screen.

Exercise 5

```
int x = (int) event.getX(); int y = (int) event.getY();
int dX, dY;

int action = MotionEventCompat.getActionMasked(event);
switch (action){
    case MotionEvent.ACTION_DOWN:
        mFirstTouchX = x; mFirstTouchY = y;
        break;
    case MotionEvent.ACTION_MOVE:
        if (selectedShapeDrawing.equals("Line")) {
            dX = 5; dY = 5;
            storeShape( shape: "Circle", x, y, dX, dY);
        }
        break;
    case MotionEvent.ACTION_UP:

        if (!selectedShapeDrawing.equals("Line")) {
            dX = x - mFirstTouchX; dY = y - mFirstTouchY;
            storeShape(selectedShapeDrawing, mFirstTouchX, mFirstTouchY, dX, dY);
        }
        break;
}
```

Continuing from last exercise, explain how this code manages to store a shape whose position and size are base on a finger drag.

When the user first taps the screen, onTouch() gets invoked and detects the event as ACTION_DOWN, we then stores the x and y coordinates as mFirstTouchX and mFirstTouchY.

When the user taps up from the screen, we also capture the x and y coordinates during tap up.

For the rectangle we then calculate the width and height by subtracting the original mFirstTouchX with tap up's x, and mFirstTouchY with tap up's y.

We then set the rectangle in its original tap down coordinates with the calculated width and height by calling storeShape.

For the circle, we calculate the radius by finding the difference in width and height like above, then takes the max as the radius.

We then set the circle in its original tap down coordinates with the calculated radius by calling storeShape.

Week 11: Touch Processing + Gestures Detection Continued

Capturing Gestures Events and reacting to them

Refer to Week 9 Content

Capturing Scaling Gesture Events and reacting to them

The **GestureDetector** Class helps us to detect common gestures used by Android such as scrolling, flinging, and long press.

For **scaling**, Android provides **ScaleGestureDetector**.

ScaleGestureDetector uses ScaleGestureDetector.OnScaleGestureListener.

GestureDetector and **ScaleGestureDetector** can be used together when we want a view to recognize additional gestures.

To report detected gesture events, gesture detectors use listener objects passed to their constructors.

Android provides the interface class ScaleGestureDetector.SimpleOnScaleGestureListener as a helper class we can extend. By creating our own class and extending it, we will have access to all methods, but we will only need to define the methods we require in our app.

Basic Scaling Example

```
private ScaleGestureDetector mScaleDetector;
private float mScaleFactor = 1.f;

public MyCustomView(Context mContext){
    ...
    // View code goes here
    ...
    mScaleDetector = new ScaleGestureDetector(context, new ScaleListener());
}

@Override
public boolean onTouchEvent(MotionEvent ev) {
    // Let the ScaleGestureDetector inspect all events.
    mScaleDetector.onTouchEvent(ev);
    return true;
}
```

```
@Override
public void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    canvas.save();
    canvas.scale(mScaleFactor, mScaleFactor);
    ...
    // onDraw() code goes here
    ...
    canvas.restore();
}

private class ScaleListener
        extends ScaleGestureDetector.SimpleOnScaleGestureListener {
    @Override
    public boolean onScale(ScaleGestureDetector detector) {
        mScaleFactor *= detector.getScaleFactor();

        // Don't let the object get too small or too large.
        mScaleFactor = Math.max(0.1f, Math.min(mScaleFactor, 5.0f));

        invalidate();
        return true;
    }
}
```

Week 11 Canvas Shape App Analysis

The same app as previous weeks except now it implements more complex drawing functions.

1. Double tap – Draws the selected shape with a standard-size after a double-tap
2. Tap then Drag – Draws the selected shape with size dynamically changing with drag. The final size is set on tap up.
3. Long tap then Drag – Selects the last drawn shape and drags it dynamically around the canvas for reposition. Final position is set on tap up.
4. Pinch-in/out – selects the last drawn shape and rescales its size based on drag size. Change only occurs at end of pinch gesture.

Most of the touch processing is handled through two gesture detector class object instantiations, **GestureDetector** and **ScaleGestureDetector**.

We supply the object into our listener by invoking **onTouchEvent()** inside the CustomView's **onTouch()** callback.

The MotionEvent from the customView's **onTouch** are passed to the 2 gesture detectors for gesture analysis and reaction:

```
mDetector = new GestureDetector(getContext(), new MyGestureListener());
mScaleDetector = new ScaleGestureDetector(getContext(), new myScaleListener());
```

```
customView.setOnTouchListener(new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent ev) {
        //Circle is the default shape if can't find the key
        selectedShapeDrawing = getActivity().getSharedPrefe
        int x = (int) ev.getX(); int y = (int) ev.getY();
        int dX, dY;
```

```
//pass all MotionEvent to Gesture Detectors for them to decide if some
//combination of MotionEvents is a Gesture or not
mDetector.onTouchEvent(ev);
mScaleDetector.onTouchEvent(ev);

return true; //event handled no need to pass it on for further handling
```

Gesture detectors analyse the touch event (**MotionEvent**) fed to them through their **onTouchEvent** method, and if applicable, trigger the appropriate callbacks on their supplied listener in their defined custom classes, **MyGestureListener()** and **myScaleListener()**.

Convenience Class – a class which simplifies access to another class

1. In our case, we use it as a class which implements either
    a. SimpleOnScaleGestureListener – for Scaling processing
    b. SimpleOnGestureListener – for Gesture processing
2. We then let our class extend the **convenience class** instead
3. This allows us to implement and override only the method headers which we require for our App. Offers **convenience** as it allows us access to all methods without cluttering up our code with a load of implemented but empty methods.
4. When a **convenience** class implements the interfaces:
    a. Boolean methods automatically default to 'false', it is important we change it to true otherwise there will be Android bugs.

Exercise 1

a. What is a Convenience class?

A class that contains all the method implementations we may need. It does this by extending the interface we want to use. For the methods we require we declare the method, for the ones we don't need we don't need to declare them.

b. What problem does it solve?

It solves the problem of code bloating by only overriding the callbacks we need and leaving the rest empty.

c. What's special about a Convenience class for a listener interface?

It implements all the method headers from several gesture listener interfaces and therefore it implements all its callbacks which we can choose to override and code a response to. We can just leave the rest empty as it would return false anyway, which indicates that our app does not utilize the gesture, preventing code bloating.

d. Why is it special?

It is special because it usually returns true and by allowing us to return false, we can prevent unnecessary blocks of code.

Exercise 2

a. *What is the difference between onSingleTapUp and onSingleTapConfirmed callbacks?*

onSingleTapUp is notified when a tap occurs with the up MotionEvent that triggered it, while **onSingleTapConfirmed** will only be called after the detector is confident that the users first tap is not followed by a second tap leading to a double tap gesture.

b. onFling and onScroll have four input parameters each. Briefly describe the data type and the role of each parameter in both callbacks

onFling (MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) : a type of scrolling that occurs when a user drags and lifts their finger quickly. Called once after onScroll.

1. e1: the first down MotionEvent that started the fling.
2. e2: the move motion event that triggered the current onFling
3. velocityX: the velocity of this fling measured in pixels per second along the x axis
4. velocityY: the velocity of this fling measured in pixels per second along the y axis

onScroll (MotionEvent e1, MotionEvent e2, float distanceX, float distanceY) : callback called repeatedly during the general process of moving the viewport.

1. e1: the first down MotionEvent that started the scrolling
2. e2: the move MotionEvent that triggered the current onScroll
3. distanceX: the distance along the X axis that has been scrolled since the last call to onScroll. This is NOT the distance between e1 and e2.
4. distanceY: the distance along the Y axis that has been scrolled since the last call to onScroll. This is NOT the distance between e1 and e2.

c. Briefly compare between onFling and onScroll in terms of
   i. Number of times the callback is called per gesture
   ii. what the end-user has to do to invoke the callback

onFling is called when the user lifts the finger after scrolling at the end of movement quickly. It is called once only after onScroll. To use it, we instantiate a **GestureDetector** Class which implements **GestureDetector**.**SimpleonGestureListener** and pass all MotionEvents into our class by supplying it in our Activity or View's onTouch listener or onTouchEvent callback. We then override **onFling** in our class and define the callback.

onScroll is the general process of moving (dragging) the viewport, and so is repeatedly called when the user drags in the screen. We do the same stuff to invoke the callback above except we override **onScroll** in our class.