Mobile Development – An overview of Android and IOS

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

Brent Simmons

Mobile Developer

Summer 2015

Contents

[Android - Google 4](#_Toc428430915)

[1. Setting up Android Environment 4](#_Toc428430916)

[2. General Overview 4](#_Toc428430917)

[3. Lifecycle – Activities and Fragments 4](#_Toc428430918)

[a. Activity lifecycle 4](#_Toc428430919)

[4. Navigation – Intents and Fragment Manager 5](#_Toc428430920)

[a. Interfaces – properly handling communication between Fragments 5](#_Toc428430921)

[b. Activity 5](#_Toc428430922)

[c. Fragment 5](#_Toc428430923)

[d. Back stack 5](#_Toc428430924)

[5. Menu – Navigation Drawer 5](#_Toc428430925)

[6. Layouts (.xml) 6](#_Toc428430926)

[a. RelativeLayout 6](#_Toc428430927)

[b. LinearLayout 6](#_Toc428430928)

[c. ScrollView 6](#_Toc428430929)

[7. References: R.java 6](#_Toc428430930)

[a. R.id.insertnamehere 6](#_Toc428430931)

[b. R.layout.insertnamehere 6](#_Toc428430932)

[c. R.drawable.insertnamehere 6](#_Toc428430933)

[8. Drawable – custom backgrounds and onClick events 6](#_Toc428430934)

[9. Dynamic listviews – custom adapters 6](#_Toc428430935)

[10. Values – constants 7](#_Toc428430936)

[a. color.xml 7](#_Toc428430937)

[b. strings.xml 7](#_Toc428430938)

[c. styles.xml 7](#_Toc428430939)

[11. AsyncTask – background tasks 8](#_Toc428430940)

[12. SQLite database 8](#_Toc428430941)

[13. Getting JSON from REST API - Volley 8](#_Toc428430942)

[14. Searching phone for program 10](#_Toc428430943)

[Diagrams of the Manion Wilkins mobile app 11](#_Toc428430944)

[iOS - Apple 16](#_Toc428430945)

[1. Overview 16](#_Toc428430946)

[a. Swift and XCode 16](#_Toc428430947)

[b. Playground 16](#_Toc428430948)

[2. Storyboard 16](#_Toc428430949)

[a. Split screen 16](#_Toc428430950)

[b. Connections 16](#_Toc428430951)

[3. Navigation 16](#_Toc428430952)

[a. Navigation drawer 16](#_Toc428430953)

[b. Protocols and extensions 17](#_Toc428430954)

[c. ContainerView 17](#_Toc428430955)

[4. Views 17](#_Toc428430956)

[a. UIViewController 17](#_Toc428430957)

[b. UIView 17](#_Toc428430958)

[c. UITableViewController 17](#_Toc428430959)

[5. Layouts 17](#_Toc428430960)

[a. Auto-layout 18](#_Toc428430961)

[6. TableViewCell 18](#_Toc428430962)

[a. Example 18](#_Toc428430963)

[7. SQLite database 19](#_Toc428430964)

[a. Example 19](#_Toc428430965)

|  |
| --- |
| Android - Google |

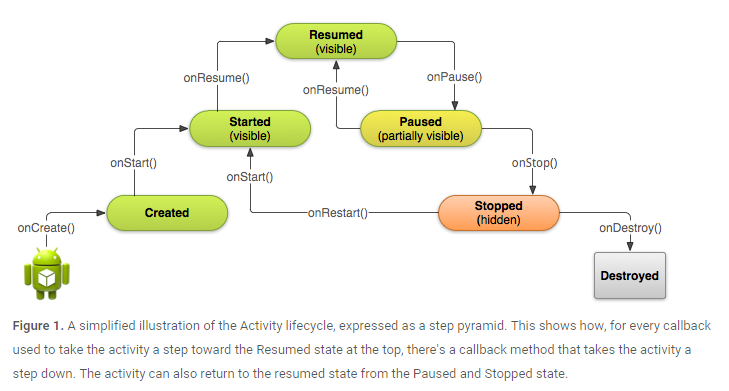
## Setting up Android Environment

* May need to change build.gradle (Module: app) file; change build tools version to 22.0.1 as version M is currently unsupported. [Update: M is now supported]
* Download all needed files from SDK manager. Tools: all the generic ones. Extras: Android Support library, Android Support Repository, Google Repository.

## General Overview

An Android app consists of Activities and Fragments. Generally, Activities take the whole screen and transition in to other Activities (example login screen to app content). Fragments are held within an Activity and are managed by the Activity.

## Lifecycle – Activities and Fragments

* 1. Activity lifecycle (taken from Android dev page):
  2. Fragment lifecycle:

Receives all the same method calls as its parent activity. It has some more methods as well:

* + onAttach()
  + onCreateView()
  + onActivityCreated()
  + onDestroyView
  + onDetach()

## Navigation – Intents and Fragment Manager

### Interfaces – properly handling communication between Fragments

The Fragment implements an interface which the Activity calls. When the Fragment needs the Activity, the Fragment gets that Activity and then can access the interface implemented in the Activity. This is useful for handling transitions between child Fragments of the parent Activity. Also, passing data can be handled here as well.

### Activity

To transition between Activities, Intents are used.

*Intent i = new Intent (CurrentActivity.this, NewActivity.class)*

*startActivity(i)*

If you want to pass simple info between Activities (Act1 -> Act2), you can use Intent.putExtra(tag, yourinfo) in Act1. This can then be retrieved from Intent.getExtra(tag) in Act2.

### **Fragment**

To transition between Fragments, interfaces should be used in the parent Activity. Get the FragmentManager and then call

*FragmentManager.beginBeginTransaction().replace(R.id.container, new NewFragment(), tag).commit()*

This replaces the current fragment with a new fragment. You have the option to add to back stack here as well.

To pass information between Fragments without an interface, you can use Bundle. Bundle only takes simple variables like String and Int. They are basically the same as an Activity Intent.putExtra and require tagging of the item you are sending.

### Back stack

This stack is what is queued when you hit the back button. By default every transition is added. You can choose to add/remove Fragments and Activities. Also you can set in the Manifest file not keep a back stack at all.

## Menu – Navigation Drawer

Update: Google has announced Material Design Support Library. This includes Navigation View which makes it easy to create the newer material design look. Implementation found here: http://www.android4devs.com/2015/06/navigation-view-material-design-support.html

Most apps require some basic form of navigation. The Android standard is to have menu button on the left side of the navigation bar. This button opens a slide out drawer menu. All the actions of this menu should be handled in an Activity. I have used the default auto-generated navigation drawer, but it has depreciated methods. I went with creating my own by following (http://www.android4devs.com/2015/01/recycler-view-handling-onitemtouch-for.html). It uses RecyclerView which helps to improve the performance of the app by reusing views already on the screen. In addition, it allows for greater customization of the menu layout and responding to user actions.

## Layouts (.xml)

### RelativeLayout

As it sounds, it lays everything out relative to one another. You can define which items you want to be laid out by using their ids. Set margins and alignments so that the items stay constant with screen rotation and differing screen sizes.

### LinearLayout

Can be implemented in horizontal or vertical. Essentially a row; good for keeping items from overlapping on screen rotation.

### ScrollView

Can be wrapped around any layout. Calculates screen size and allows scrolling if screen size is smaller than the layout it contains.

## References: R.java

### R.id.insertnamehere

You can get items from layout by their ids. In an Activity or Fragment, call findElementById(R.id…).

### R.layout.insertnamehere

Used when inflating view (LayoutInflator). You assign your layout to your Activity class in your onCreate() or onCreateView (Fragment).

### R.drawable.insertnamehere

Used to get drawable items. Can be referenced in the .xml or the .java files.

## Drawable – custom backgrounds and onClick events

Drawable files are can be pictures or .xml files. The pictures can be assigned to backgrounds. The .xml allows for changing backgrounds onClick, rounding corners, adding borders etc. This is where one can really customize the look of the app.

## Dynamic listviews – custom adapters

Used to display arrays of data with a specific view. Need to create a ListView and assign it a custom adapter class. This adapter class is usually given the array of data and iterates through it building each view. The following example code shows how the ListView is created in the activity and what happens in the getView method in the custom adapter. Info\_Class just stores a Name and Date.

*// ActivityOne*

*private ArrayList<Info\_Class> arr = new ArrayList()  
private ListView listView*

*@Override  
public void onStart() {*

*super.onStart()*

*arr.add(new Info\_Class(“Name”,”Date”))  
listView = (ListView) findViewById(R.id.listViewId)*

*//Declare custom adapter class*

*Adapter\_For\_ListView adapter = new Adapter\_For\_ListView (ActivityOne.this, arr)*

*}*

*// Adapter\_For\_ListView extends ArrayAdapter<Info\_Class>*

*@Override  
public View getView(int position, View convertView, ViewGroup parent) {  
 LayoutInflater inflater = (LayoutInflater) context.getSystemService(Context.LAYOUT\_INFLATER\_SERVICE)  
 View rowView = inflater.inflate(R.layout.adapter\_for\_listview\_layout, parent, false)  
  
 TextView name = (TextView) rowView.findViewById(R.id.nameId)  
 TextView date = (TextView) rowView.findViewById(R.id.dateId)   
  
 name.setText(info.get(position).getName());  
 date.setText(info.get(position).getDate());  
  
 return rowView;  
}*

The example above does not handle the recycling of views very well. Views are only created for what is displayed on screen to conserve memory. When the user scrolls, new views are rendered. If the same view has just been used, it is more efficient to refresh the data in the view instead of calling and recreating the whole view again. You can use ViewHolder to check if the view has already been created, but this is not mandatory. An upgrade to ListView has been introduced called RecyclerView. While being more complex, RecyclerView forces the use of ViewHolder and is more customizable with the ability to manage animations, touches, and layouts.

## Values – constants

These are used for maintaining consistency throughout the app.

### color.xml

Define colors here to be referenced for backgrounds, text colors, and drawables.

### strings.xml

Define type String here to be used for static information.

### styles.xml

Define the overall color scheme of the app. Can create multiple styles which could be used to allow the user to personalize the app.

## AsyncTask – background tasks

AsyncTask is a cleaner solution to multithreading. AsyncTask handles tasks to be performed off the main thread of the application. It contains methods such as doInBackground, onPostExecute, onPreExecute, and onProgressUpdate. These are useful for loading data from the server or images in ListView. One thing to note is that it should only be used for processes of a few seconds at most as it is tied (loosely) to the Activity (i.e. killing the Activity kills the task). For longer running background tasks, an Android Service should be used.

## SQLite database

This database is created on the first instance of a call to the database (i.e. getWritableDatabase or getReadableDatabase). It persists until the app is uninstalled. The standard is to create a static class with the table name and columns. These can then be referenced when creating the database. SQLite database can handle basic querying, inserts, updates, and deletes.

The DatabaseTesting project I created is a good example of how the Android docs set up a database. This tutorial lays it out really well: http://hmkcode.com/android-simple-sqlite-database-tutorial/ .

If you want to delete the physical database, go to Android Device Monitor (DDMS) and delete the file under file explorer - data/data/*your\_project\_name*/*database\_name*.db.

## Getting JSON from REST API - Volley

The way the REST API is called in the oldboys\_android project works, but the methods used are now depreciated as of API 22. The Volley Library (Volley) was created by Google to take care of many of the issues surrounding HttpClient and HttpUrlConnection. Volley caches everything so that if an Activity is destroyed, you can still use the data you have. In addition if an activity is cancelled, it takes care of that nicely instead of the programmer catching and throwing an exception.

This example shows a GET method obtains JSON data from a REST API:

*String url = "http://bsimms2.byethost5.com/index.php/schedule"*

*JsonObjectRequest jsonRequest = new JsonObjectRequest*

*(Request.Method.GET, url, null, new Response.Listener<JSONObject>() {*

*@Override*

*public void onResponse(JSONObject response) {*

*// the response is already constructed as a JSONObject!*

*try {*

*response = response.getJSONObject("args")*

*String site = response.getString("site"),*

*network = response.getString("network")*

*System.out.println("Site: "+site+"\nNetwork: "+network)*

*} catch (JSONException e) {*

*e.printStackTrace()*

*}*

*}*

*}, new Response.ErrorListener() {*

*@Override*

*public void onErrorResponse(VolleyError error) {*

*error.printStackTrace()*

*}*

*});*

*Volley.newRequestQueue(this).add(jsonRequest)*

From this great tutorial: http://code.tutsplus.com/tutorials/an-introduction-to-volley--cms-23800

Another example from the tutorial above that shows a POST to a REST API:

*String url = "http://httpbin.org/post"*

*StringRequest postRequest = new StringRequest(Request.Method.POST, url,*

*new Response.Listener<String>() {*

*@Override*

*public void onResponse(String response) {*

*try {*

*JSONObject jsonResponse = new JSONObject(response).getJSONObject("form")*

*String site = jsonResponse.getString("site"),*

*network = jsonResponse.getString("network")*

*System.out.println("Site: "+site+"\nNetwork: "+network)*

*} catch (JSONException e) {*

*e.printStackTrace();*

*}*

*}*

*},*

*new Response.ErrorListener() {*

*@Override*

*public void onErrorResponse(VolleyError error) {*

*error.printStackTrace()*

*}*

*}*

*) {*

*@Override*

*protected Map<String, String> getParams()*

*{*

*Map<String, String> params = new HashMap<>()*

*// the POST parameters:*

*params.put("site", "code")*

*params.put("network", "tutsplus")*

*return params*

*}*

*};*

*Volley.newRequestQueue(this).add(postRequest)*

This POST sends in parameter (getParams()) and returns the parameters.

One thing to note is that Volley is currently not officially supported by AOSP which means that you have to download the library separately. From a brief implementation test and google search, I have concluded that it is still buggy/unclear in some areas especially the JSONObjectRequest class. While Google has integrated Volley into most of their apps, it may unwise to do so until Volley has been declared officially part of Android.

## Searching phone for program

You can use other applications on the phone to perform actions such as opening pdfs. This requires the use of Intents (this is a very powerful class).

Here is an example from the Manion Wilkins prototype that searches the external storage of the phone and opens a pdf through the default viewer:

*public void openPDF(String fileLocation){*

*File pdfFile = new File(Environment.getExternalStorageDirectory().getAbsolutePath()+ fileLocation)*

*if(pdfFile.exists())*

*{*

*Uri path = Uri.fromFile(pdfFile)*

*Intent pdfIntent = new Intent(Intent.ACTION\_VIEW)*

*pdfIntent.setDataAndType(path, "application/pdf")*

*try*

*{*

*startActivity(pdfIntent)*

*}*

*catch(ActivityNotFoundException e)*

*{*

*//enter error message here; ‘no default pdf view found’*

*}*

*}else{*

*//enter error message here; ‘pdf file does not exist on phone’*

*}*

*}*

## Diagrams of the Manion Wilkins mobile app

*A brief overview of the application’s functionalities and how to achieve them*

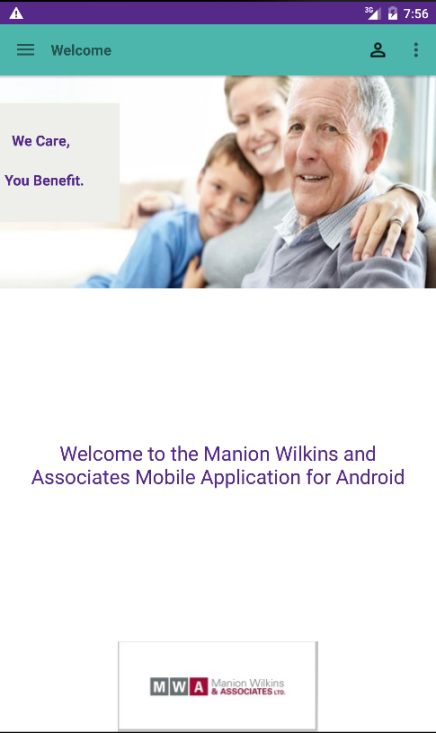
**MainActivity**

Has control of the entire screen

Reference: MainActivity.java

**Status Bar**

Can be customized, but varies across different phone versions. It overlay the currently displayed Activity.



**Navigation Bar**

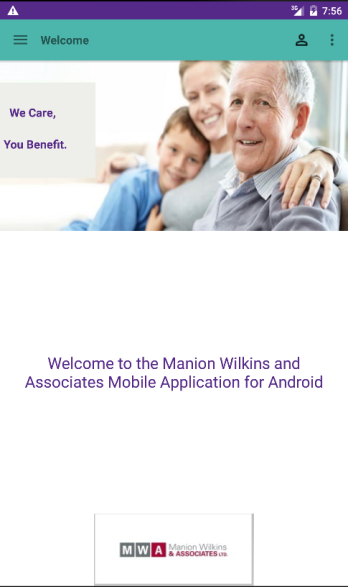
Part of the Activity. Maintains consistency throughout the application.

Reference: MainActivity.java

**Fragment**

Controlled in FragmentManager in the Activity.

Reference: MainActivity.java



**Fragment\_Welcome.java**

Method: onCreateView inflates layout fragment\_welcome.

Entirely static view. Nothing dynamically generated.

**Drawer/Menu button**

Opens the slide-in drawer.

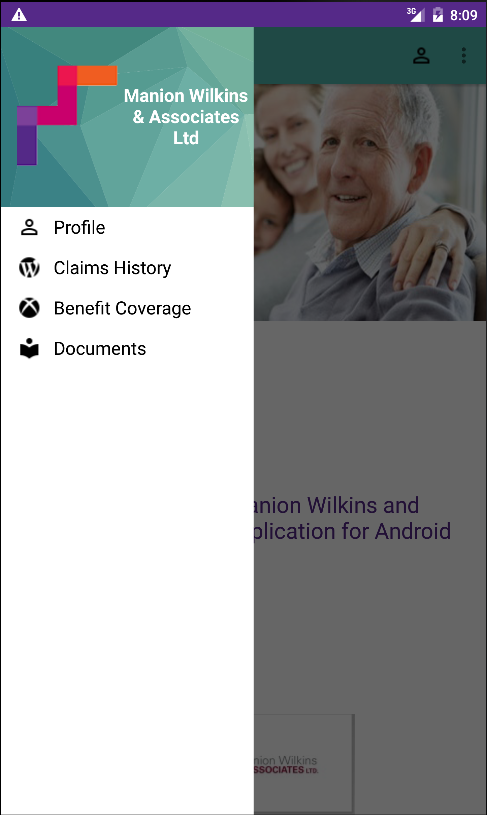
MainActivity.java

DrawerLayout is the drawer

ActionBarDrawerToggle is the event listener for the drawer

The whole bar is clickable which is set in title.setOnClickListener





**Menu**

In layout activity\_main:

**android.support.v7.widget.RecyclerView**

The view is built from Adapter\_MainMenu

It includes layouts mainmenu\_header and mainmenu\_row

It is used to switch between the Fragments in the MainActivity.java

**onClick**

MainActivity.java

mRecyclerView.addOnItemTouchListener

I intercepted the touch to have full control of the events (no default OnTouch events performed).

I get the item clicked and through the FragmentManager, I transition to the new Fragment.



**ActionBar Buttons**

Reference: MainActivity.java

Method: onOptionsItemSelected



**Submenu button/bar**

Reference: MainActivity.java

Is hidden/shown when needed:

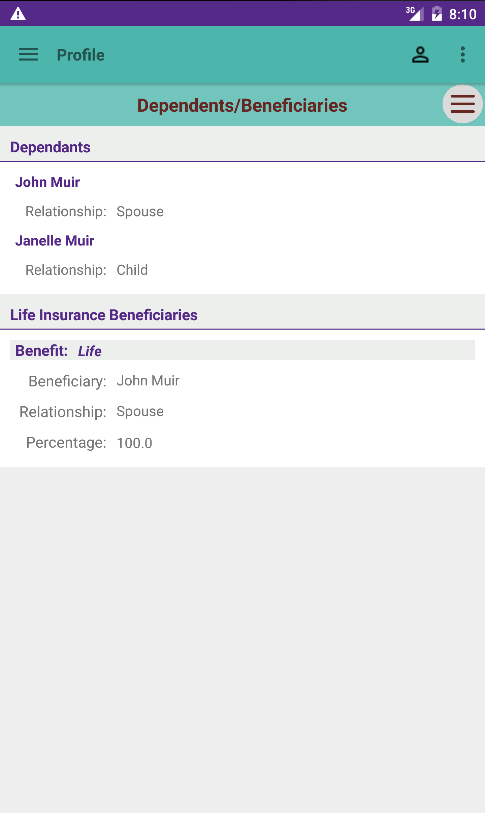
**submenu\_bar**.setVisibility(View.***VISIBLE***);

Is set through setSubmenuButton()

->has onMenuClickListener

Is updated through updateSubmenuAndTitle()

The whole bar is clickable which is set in submenu\_bar.setOnClickListener



**fragment\_profile\_depedantsandbeneficiaries.xml**

<ScrollView

<RelativeLayout

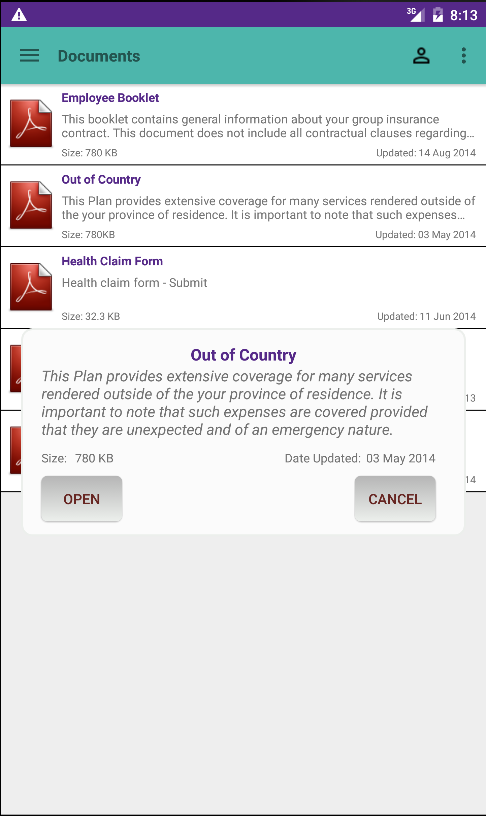
//Dependants wrapped in RelativeLayout

//bunch of texts view

//Life Insurance Beneficiaries

</RelativeLayout>

</ScrollView>



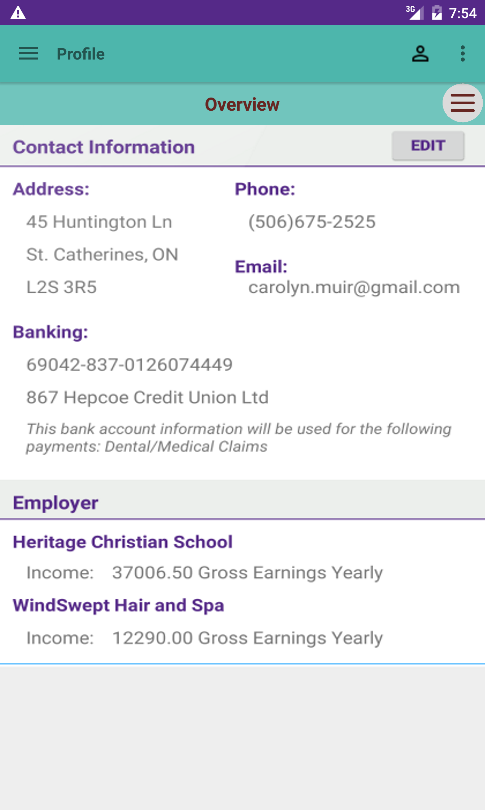
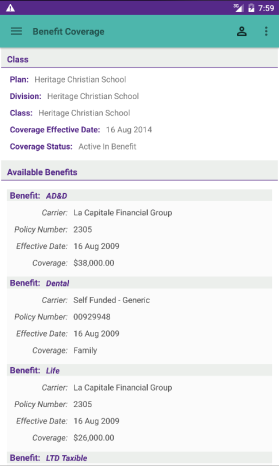
**Fragment\_Documents.java**

Static view with onClickListeners on each pdf view

->Find RelativeLayout by id and set click listener

This listener triggers a popUpWindow to open. It would be wise to build your own as this one is very tough to customize.

The openPdf method searches for a .pdf viewer on the phone, if found, opens it.



**SQLite Database**

Fragment\_Profile\_ContactANDEmployer.java

This loads the contact information in from the database.

The EDIT button loads a new Fragment to allow for editing the information.

I have created another project (**DatabaseTesting**) as a better example of how to implement a database. It takes more of a Model View Controller approach.

**Fragment\_BenefitCoverage.java**

Uses ListView to dynamically create the view.

fragment\_benefitcoverage\_listview.xml only contains a ListView

In this fragment, I set up the ListView with a header and footer and dynamically generated the different benefits in the middle. It starts as an ArrayList of type Info\_Benefit (a POJO). This is then passed to Adapter\_Benefit which inflates each view and loads the data in accordingly.

Header: adapter\_benefit\_header.xml

Footer: adapter\_benefit\_footer.xml

Individual view: adapter\_benefit\_coverage.xml

**Fragment\_ClaimHistory.java**

Uses ExpandableListView to dynamically create the view.

fragment\_claim\_expandablelistview.xml only contains a table row across the top with the four column names and an ExpandableListView.

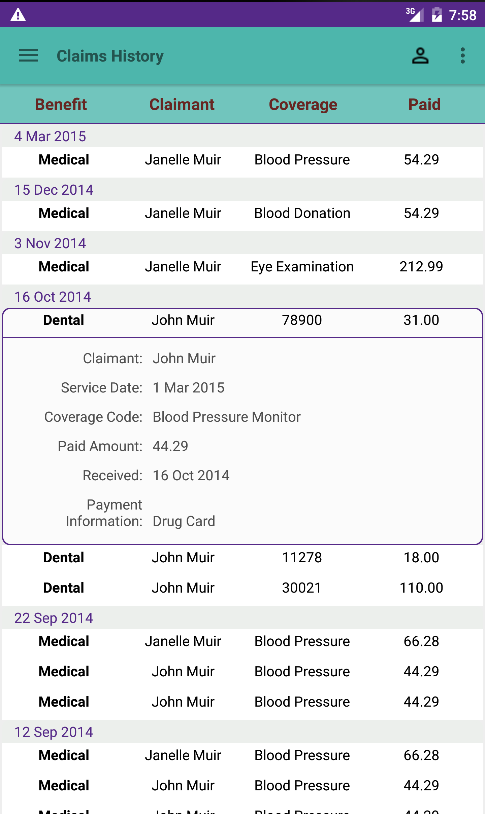
ExpandableListView has two components, the group and its children. These are set in Adapter\_ClaimHistory.java

getChildView sets the child view and getGroupView sets group view.

For efficiency, I check if the view is already inflated

*if ( convertView == null )*

and if it is, I just assign new values in the current view. If it is not, I inflate the new view.



**onClick**

Reference: Fragment\_ClaimHistory.java

Opening and closing the view is set in the onViewCreated method.

*lv.setOnGroupClickListener* and *lv.setOnChildClickListener*

Actions onClick:

->The date bar ignores the click.

->The column group bar opens the child view.

->Clicking on the group bar or child view closes the child view.

**Borders**

Reference: Adapter\_ClaimHistory.java

In order to dynamically set each background as an .xml, setBackgroundResource is called. This sets the currently inflated view to that background.

|  |
| --- |
| iOS - Apple |

## Overview

### Swift and XCode

Swift is a new language produced by Apple that feels like a mix between Java and Javascript. XCode is the dev environment with a central view surrounded by multiple action and overview tab columns.

### **Playground**

Playground files allow for real-time debugging of complex logic and compiling of views. The programmer codes on the left and the output of each line is display on the right. For loops and recursion, the ‘stacktrace’ can be examined.

## Storyboard

Storyboard acts as a control room for your app. It can handle all the views, navigation, class assigning, referencing, and screen sizes. The Storyboard allows for a visual design approach to the app; this can all be done purely programmically as well.

### S**plit screen**

Besides the three main views that can be toggled in the top right, there are a couple of others to make note of. In Storyboard, the hierarchical view can be opened to see what specific view is currently in focus. In addition, the screen can be split into two to allow connections between the views in Storyboard and their respective classes. The second pane can also be adjusted so it previews the current view selected. This preview is very useful for identifying problems with multiple screens sizes and portrait/landscape views.

### Connections

In Storyboard, ViewControllers can be assigned to a class. In addition, it allows direct reference to items within the ViewController. Most of this is handled by a simple drag-and-drop (press ctrl and drag to where you want the reference in your class) although items that will be reused (i.e. TableViewCells) cannot be tied to a specific reference. Navigation between ViewControllers can be handled this way as well with Segues.

A common occurrence with Storyboard is that you will delete some connection, but it will still be retained in the source code of Storyboard. You then have to switch to source code view and delete the problem code manually. Usually easy to find as it is the error states an unknown reference.

## Navigation

### **Navigation drawer**

This feature was primarily present on Android. Some have argued that it is not needed for primary navigation and apps should be redesigned to have simple tabs (Facebook started with a navigation drawer, but have recently switched to tabs on all mobile platforms). In terms of user familiarity, the navigation drawer feels natural for displaying menus.

One con to the implementation of the drawer is the need for an external library: SWRevealViewController (Developed by John Lluch). This library comes highly recommended by the general iOS community.

### Protocols and extensions

Protocols and extensions can be used to delegate information to a central controller. A protocol in Swift is essentially the same as an interface in Java. One can set optional and mandatory methods to include. Extensions are global methods. They can be run from anywhere within your project scope, but cannot store variables.

To use for navigation, generally a protocol is set up to maintain code integrity. An extension is made from the class in which you want to have central control. Then all the ‘child’ classes or views will just call the extension and delegate control to that class. This makes it easier to switch between views in a ViewController or ContainerView.

### ContainerView

This acts as any other View and must be contained in some type of Controller. This view houses child views that it can transition in and out as needed. Make sure to clear the previous view before loading the next one to decrease memory usage or you can check if you have the view loaded and reload.

## Views

Views are contained within a ViewController. A ViewController can rotate through child views or give the screen to another ViewController.

### UIViewController

Found on Apple Mobile Dev docs:

* ViewDidLoad - Called when you create the class and load from xib. Great for initial setup and one-time-only work.
* ViewWillAppear - Called right before view appears, good for hiding/showing fields or any operations that you want to happen every time before the view is visible. Because you might be going back and forth between views, this will be called every time your view is about to appear on the screen.
* ViewDidAppear – Called as soon as the view appears. Animations can started here. Clear loading feature.
* ViewWill/DidDisappear - Same idea as WillAppear.
* ViewDidUnload/ViewDidDispose – Mostly handles clean-up of views; performed automatically by Swift.

### UIView

This is contained inside of the ViewController. It is what will be seen on screen. There will usually be a main one that contains all the other views.

### UITableViewController

TableViewController contains a TableView which displays the TableViewCells. The cells are customizable and can be dynamically loaded with data based on a cell ‘reuse identifier’.

## Layouts

### A**uto-layout**

Auto-layout was implemented by Apple to deal with multiple screen sizes. It takes care of all the resizing based on the width and height of the screen. Best to use the preview screen when setting up views as it is difficult to set up the proper constraints. One can also ‘update frames’ on the whole view or just the view in focus. This will reflect the view that will be generated at run-time (run-time view highly consistent with preview screen).

To get auto-layout working properly, no ambiguous constraints are allowed. If the view in focus is properly defined, the constraints glow blue, if not, they glow orange. Constraints can be added manually (recommended) or by clicking ‘add missing constraints’.

Apple does not recommend defining alternative size classes as auto-layout should be used to take care of all the screen sizes. One can specify size classes though and this is useful when designing for screen rotation.

Trouble arose when utilizing what should be a relatively simple feature, UIScrollView. It is tough to get it right especially when using ContainerView. I do not recommend coupling the two together unless necessary.

## TableViewCell

### Example

This takes place in a ViewController with TableView.

*//Get your reference to TableView*

*class CustomTableViewController : UIViewController {*

*@IBOutlet weak var tableView: UITableView!*

*let list = [ new CustomListClass(“Name”) ]*

*override func viewDidLoad() {*

*super.viewDidLoad()*

*}*

*func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell*

*let cell = tableView.dequeueReusableCellWithIdentifier("cell", forIndexPath: indexPath) as! CustomCell*

*cell.configureForMenu(list[indexPath.row])*

*return cell*

*}*

*class CustomCell: UITableViewCell {*

*@IBOutlet weak var nameLabel: UILabel!*

*@IBOutlet weak var exCell: UIView!*

*func configureForMenu(items: CustomListClass {*

*nameLabel.text = items.name*

*}*

*}*

*class CustomListClass {*

*var name : String?*

*init(name: String){*

*self.name = name*

*}*

*}*

## SQLite database

Install libsqlite3.dylib.

### Example

*//Example of SELECT statement; loading data onto screen*

*func loadData(){*

*let contactDB = FMDatabase(path: databasePath as String)*

*let querySQL = "SELECT street, city, province, postal, phone, email FROM CONTACTS WHERE ID = 1 "*

*let results:FMResultSet? = contactDB.executeQuery(querySQL, withArgumentsInArray: nil)*

*if results?.next() == true {*

*name.text = results?.stringForColumn(“name")*

*println(“Results found”)*

*} else {*

*println("Results NOT found...")*

*}*

*contactDB.close()*

*}*

*//Example of UPDATE statement; updating row in database*

*func saveToDB(){*

*let contactDB = FMDatabase(path: databasePath as String)*

*let insertSQL = "UPDATE contacts SET name = ‘\(name.text)’ WHERE ID = 1 "*

*let result = contactDB.executeUpdate(insertSQL, withArgumentsInArray: nil)*

*}*

#### Refactoring

Refactoring does not work very well with XCode. It usually only changes the file name and does not take into account other files. You have to do a manual find-and-replace.

Before doing any major refactoring, you can save the current state of your XCode project.

Changing app name can be found under Bundle Name. Do not change Product Name as this blows the whole project to absolute pieces and then some…