Mobile Application Development

Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

User Interface Design & Development - Part 1

Goals of this Section

- Understand the basics of Android UI Development
- Be able to create and use some more different widgets (views) such as **`AdapterViews`** and **`ArrayAdapters`**
- Share data between Activities using **`Bundles`** (just a brief look, we'll cover it and more in detail, in the Persistence lecture notes)
- Understand how to develop and use **`Fragments`** in a multi-screen app

Mobile Development in General

- When developing software for the web or a desktop computer, you only need to consider the mouse and the keyboard.

- With a mobile device, you must take into account the entire world around you (and your users).

- The “60 second Vs 60 minute” Use Case

Possible User Input Sources

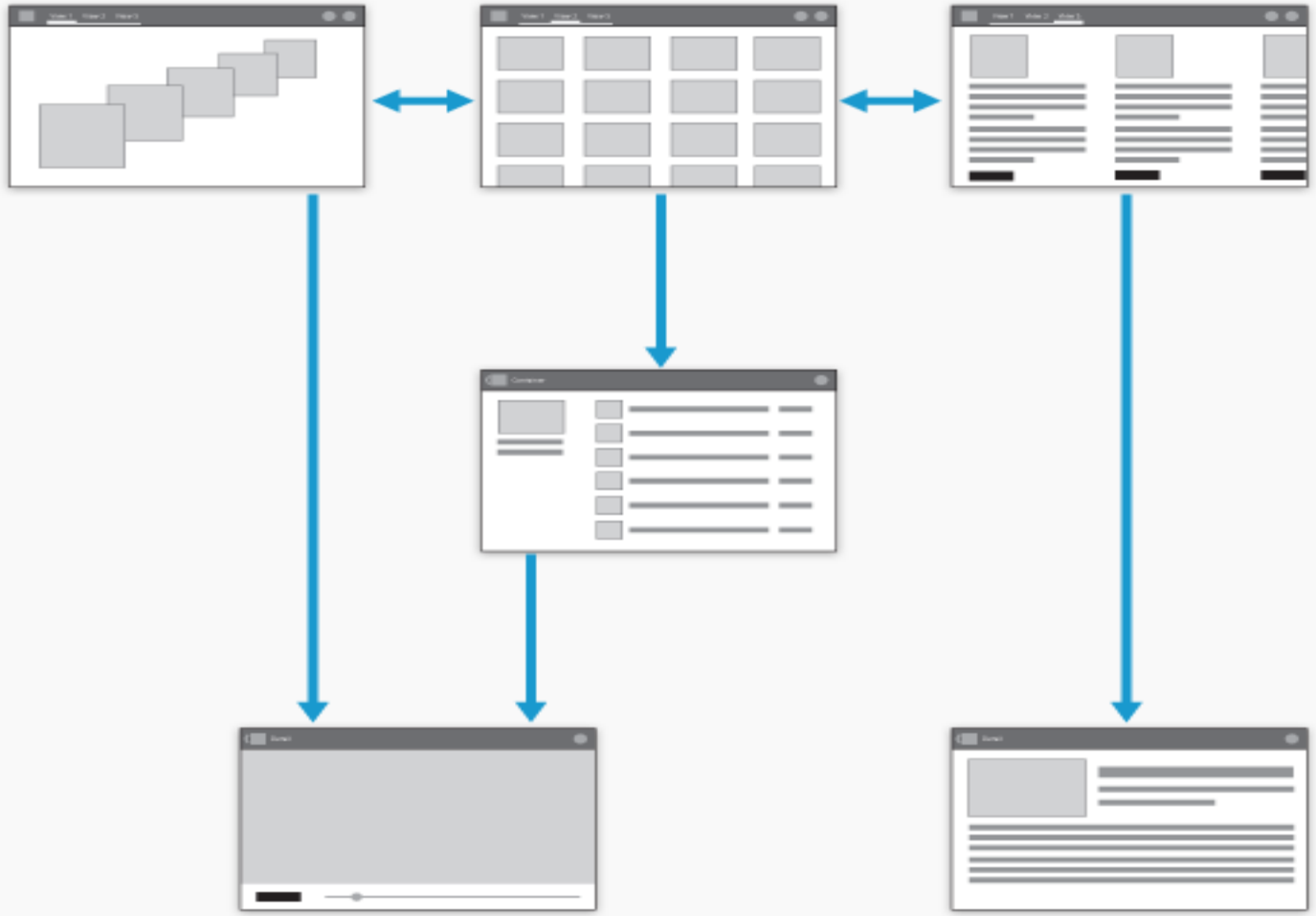
- Keyboard
- "Click" Tap via Touch (or Stylus)
- Wheel or Trackball
- GPS or Network Location
- Accelerometer Motion
- Orientation / Compass / Altitude
- Vibration
- Sound / Music
- WiFi Coverage
- Environment Lighting
- Multitouch & Gestures
- Device Security / Loss

App Structure

- Apps come in many varieties that address very different needs
- For example:
  - Apps such as Calculator or Camera that are built around a single focused activity handled from a single screen
  - Apps such as Phone whose main purpose is to switch between different activities without deeper navigation
  - Apps such as Gmail or the Play Store that combine a broad set of data views with deep navigation
- Your app's structure depends largely on the content and tasks you want to surface for your users

App Structure

A typical Android app consists of top level and detail/edit views. If the navigation hierarchy is deep and complex, category views connect top level and detail views.



Top level views

The top level of the app typically consists of the different views that your app supports. The views either show different representations of the same data or expose an altogether different functional facet of your app.

Category views

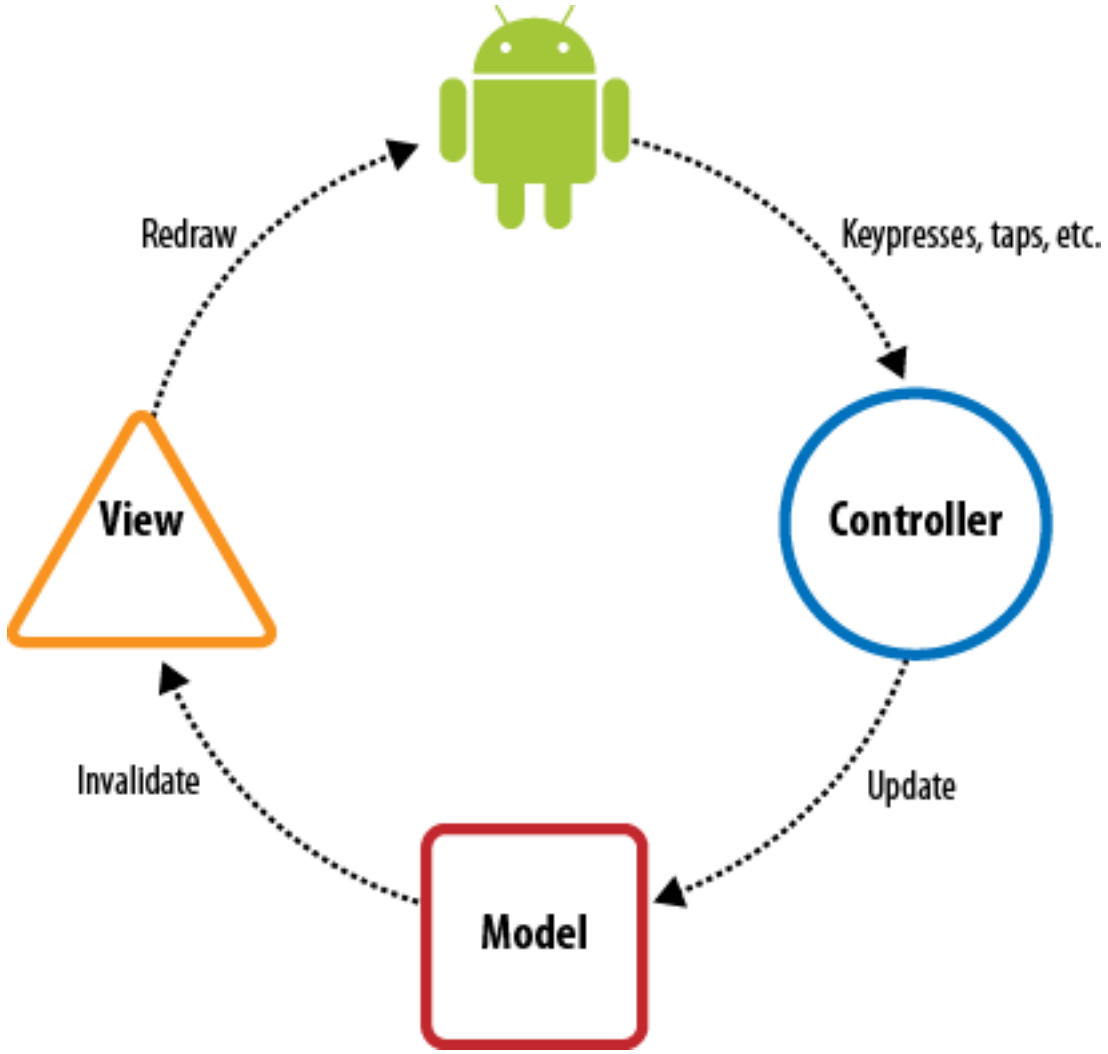
Category views allow you to drill deeper into your data.

Detail/edit view

The detail/edit view is where you consume or create data.

App Structure & The Android Framework

- The Android UI framework is organised around the common MVC pattern.



Some General UI Guidelines – (**UIGs**)

- Activity and Task Design
  - *Activities* are the basic, independent building blocks of applications. As you design your application's UI and feature set, you are free to **re-use activities** from other applications as if they were yours, to enrich and extend your application.
- "Everything is a Resource"
  - Many of the steps in Android programming depend on creating resources and then loading them or referencing them (in XML files) at the right time

UIGs - Screen Orientation

- People can easily change the orientation by which they hold their mobile devices
  - Mobile apps have to deal with changes in orientation frequently
  - Android deals with this issue through the use of resources (more on this later)
- Start with Portrait Orientation
  - It is natural to start by designing the UI of your main activity in portrait orientation
  - That is the default orientation in the Eclipse plug-in

UIGs - Unit Sizes

- Android supports a wide variety of unit sizes for specifying UI layouts;
  - px (device pixel), in, mm, pt (1/72nd of an inch)
- All of these have problems creating UIs that work across multiple types of devices
  - Google recommends using resolution-independent units
    - dp (or dip): density-independent pixels
    - sp: scale-independent pixels
- In particular, use sp for font sizes and dp for everything else

UIGs - Layouts

- **LinearLayout**: Each child view is placed after the previous one in a single row or column
- **RelativeLayout**: Each child view is placed in relation to other views in the layout or relative to its parent's layout
- **FrameLayout**: Each child view is stacked within a frame, relative to the top-left corner. Child views may overlap
- **TableLayout**: Each child view is a cell in a grid of rows and columns
- …

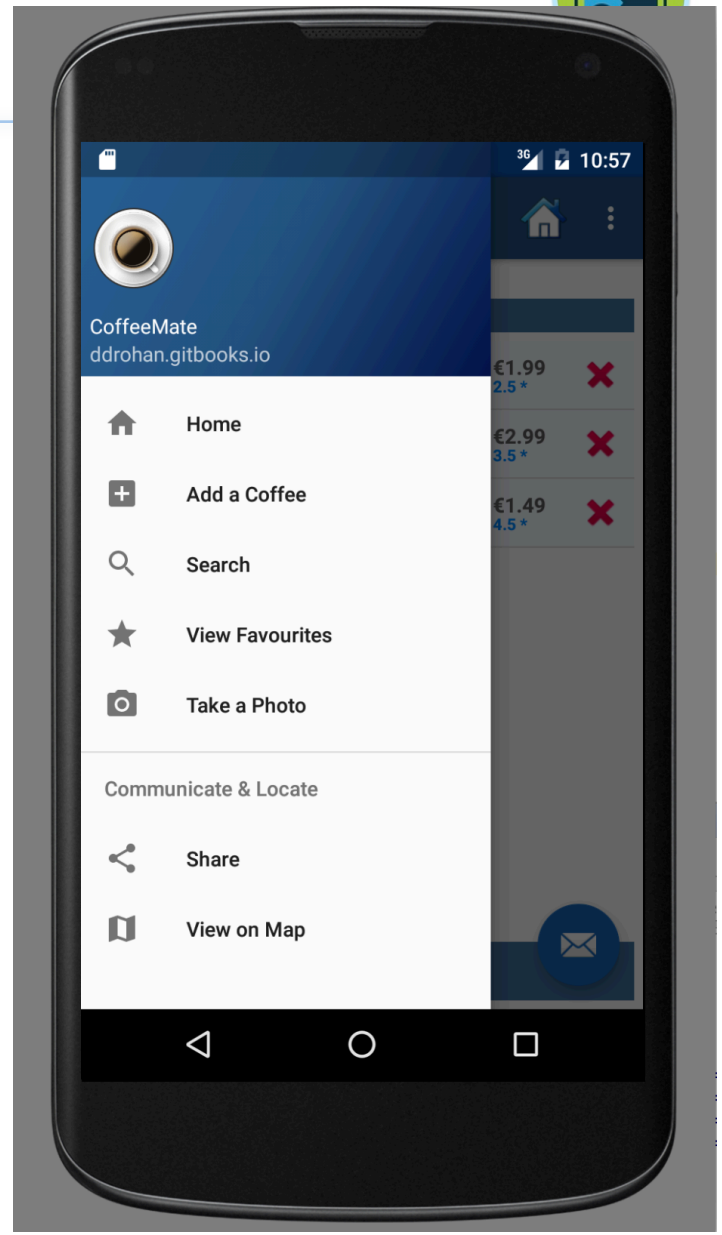
UIGs - Specifying the Size of a View

- We've previously discussed the use of resolution-independent measurements for specifying the size of a view
- These values go in the XML attributes
  - android:layout_width and android:layout_height
- But, you can get more flexibility with
  - fill_parent: the child scales to the size of its parent
  - wrap_content: the parent shrinks to the size of the child

Case Study

- *CoffeeMate* – an Android App to keep track of your Coffees, their details, and which ones you like the best (your favourites)
- App Features
  - List all your Coffees
  - View specific Coffee details
  - Filter Coffees by Name and Type
  - Delete a Coffee
  - List all your Favourite Coffees

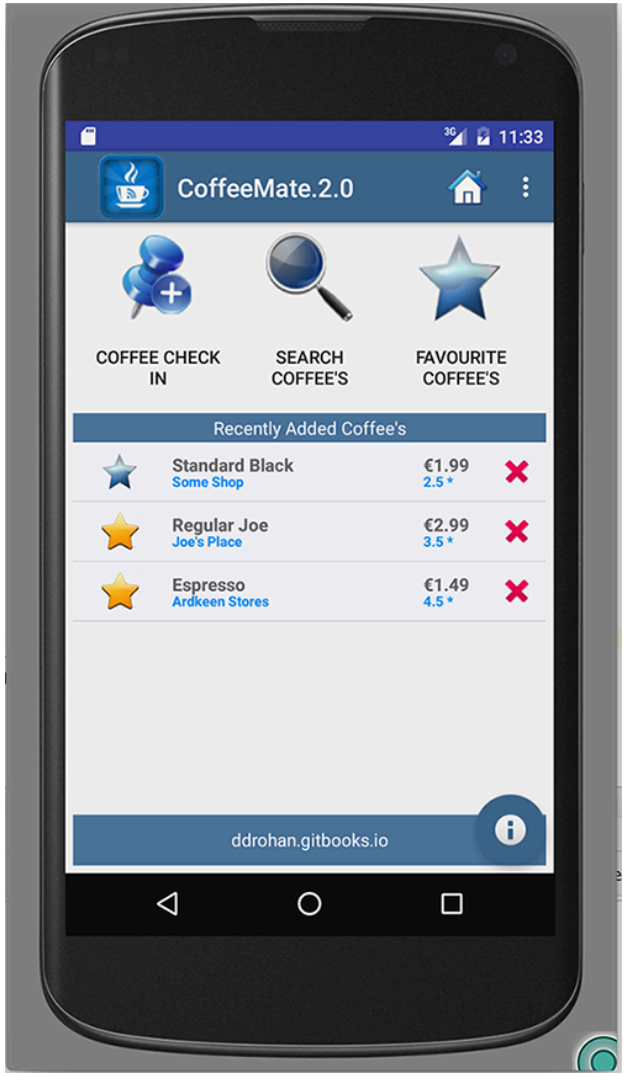
(View Nearby Coffees / on a Map ???)

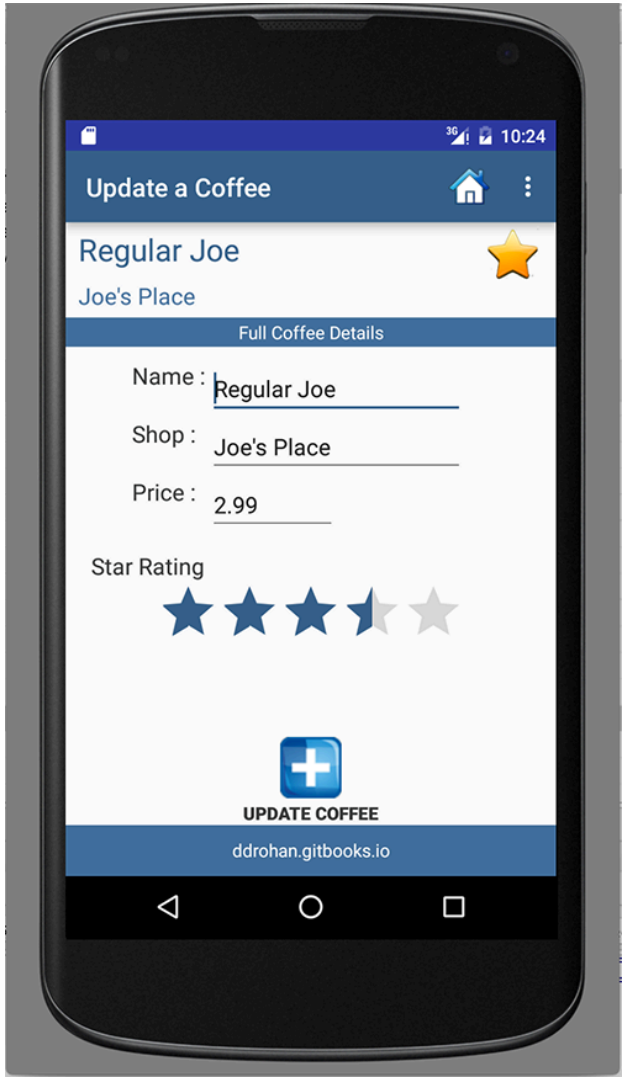


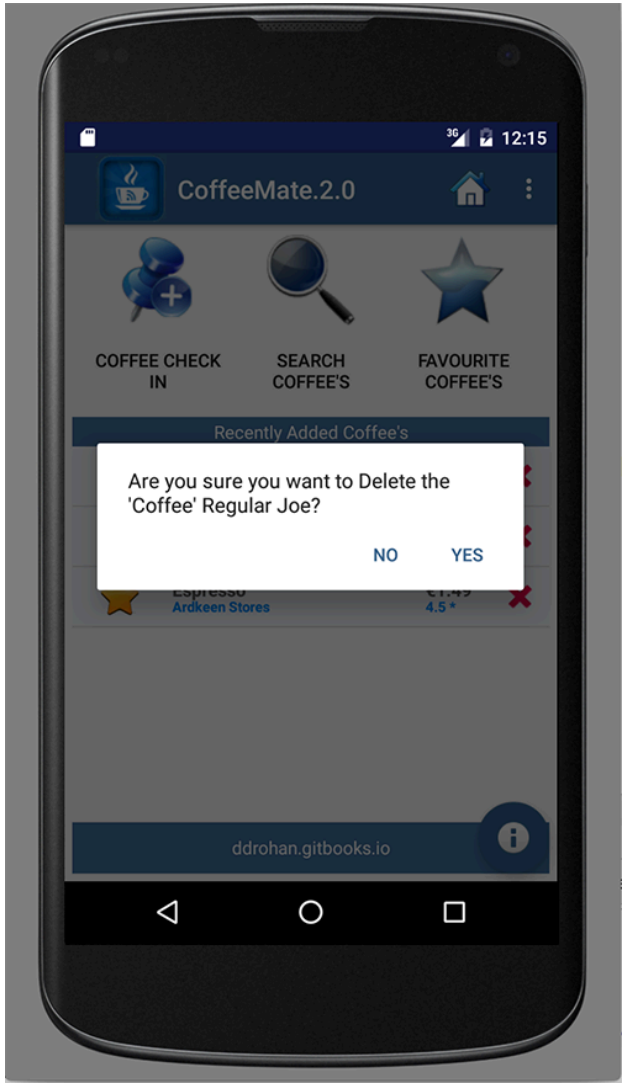
CoffeeMate 2.0

Using Fragments and Custom ArrayAdapters

CoffeeMate 2.0

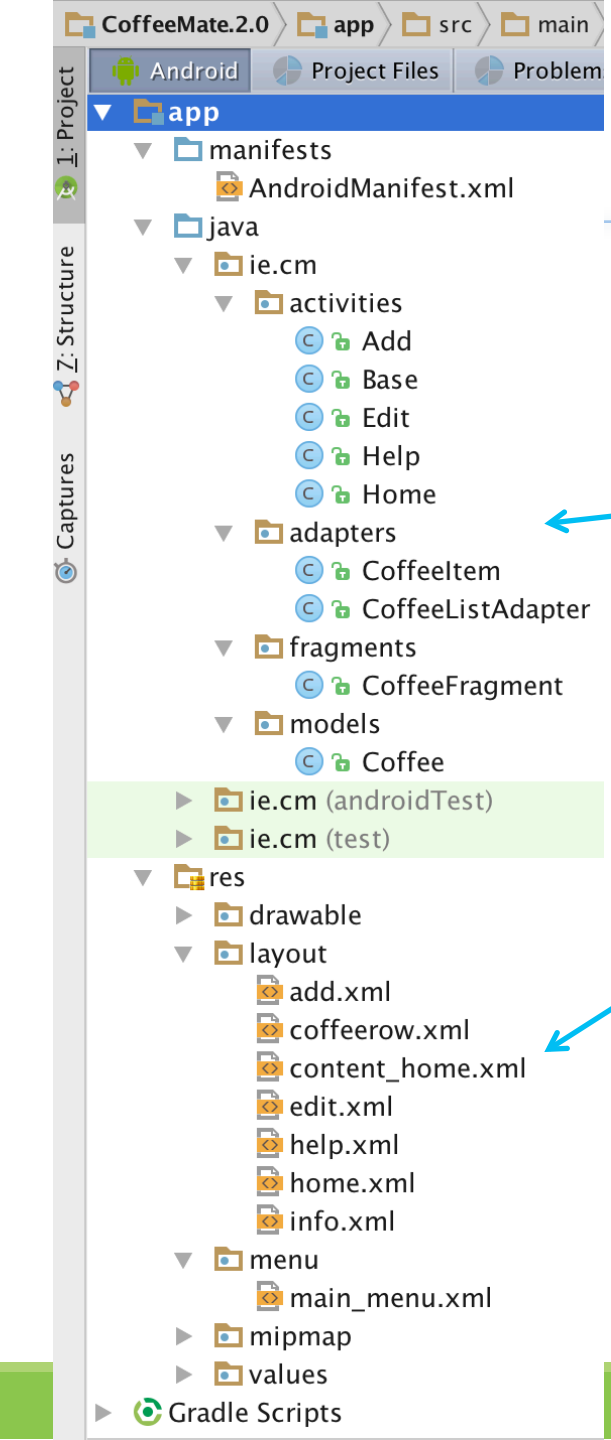






No Persistence in this Version

CoffeeMate 2.0



- 4 new java source files
- 2 new xml layouts

CoffeeMate 2.0

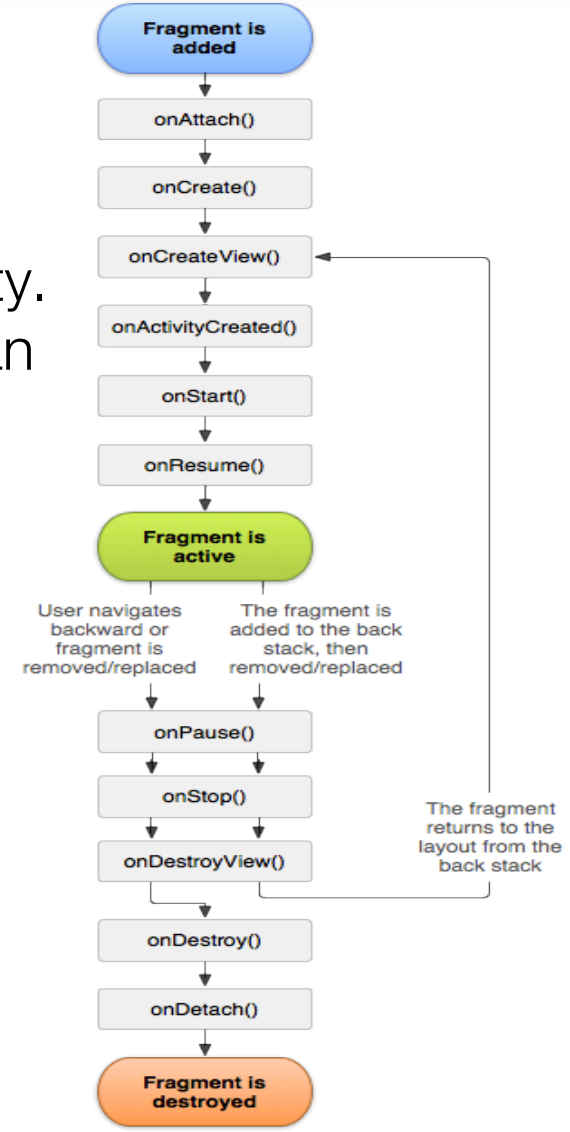
Using Fragments

Fragments - Recap

- Fragments represents a behaviour or a portion of a user interface *in an **Activity***.
- You can combine multiple fragments in a single activity and *reuse a single fragment in multiple activities*.
- Each Fragment has its own lifecycle (next slide).
- A fragment must always be embedded in an activity.
- You perform a ***fragment transaction*** to add it to an activity.
- When you add a fragment as a part of your activity layout, it lives in a *ViewGroup* inside the activity's view hierarchy and the fragment defines its own view layout.

The Fragment Life Cycle

- To create a fragment, you must subclass Fragment (or an existing subclass of it).
- Has code that looks a lot like an Activity. Contains callback methods similar to an activity, such as *onCreate()*, *onStart()*, *onPause()*, and *onStop()*.
- Usually, you should implement at least *onCreate()*, *onCreateView()* and *onPause()*



CoffeeMate 2.0

Code Highlights (1)

Revisit Base

```
public class Base extends AppCompatActivity {

    public static ArrayList<Coffee>     coffeeList = new ArrayList<~>();
    protected Bundle                    activityInfo; // Used for persistence (of sorts)
    protected CoffeeFragment            coffeeFragment; // How we'll 'share' our
                                                        // List of Coffees between Activities

    protected void goToActivity(Activity current,
                                Class<? extends Activity> activityClass,
                                Bundle bundle) {...}

    public void openInfoDialog(Activity current) {...}

    @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, menu);
        return true;
    }

    public void menuInfo(MenuItem m) { openInfoDialog(this); }

    public void menuHelp(MenuItem m) { goToActivity(this, Help.class, null); }

    public void menuHome(MenuItem m) { goToActivity(this, Home.class, null); }

    protected void toastMessage(String s) { Toast.makeText(this, s, Toast.LENGTH_SHORT).show(); }
}
```

A Bundle for passing data between activities

A reference to our Custom Fragment

Revisit Home

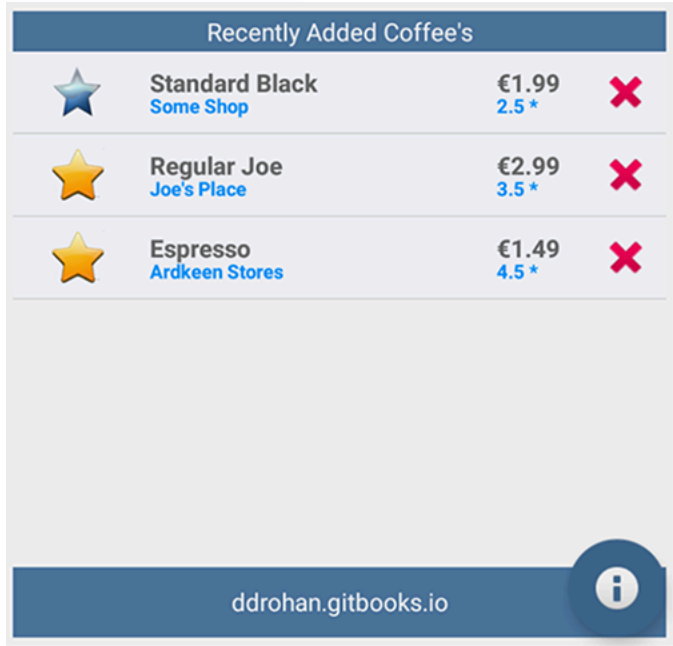
```
public class Home extends Base {

    TextView recentList;

    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    public void add(View v) { goToActivity(this,Add.class,null); }

    @Override
    protected void onResume() {
        super.onResume();

        if(coffeeList.isEmpty())
            recentList.setText("You have no Coffee's added, go have a coffee!");
        else
            recentList.setText("");

            coffeeFragment = CoffeeFragment.newInstance(); //get a new Fragment instance
            getFragmentManager()
                    .beginTransaction()
                    .replace(R.id.fragment_layout, coffeeFragment)
                    .commit(); // add/replace in the current activity


    }

    public void setupCoffees(){...}
}
```

Creating a Fragment instance and adding it to our Home Activity (we'll take a close look at the Fragment class next)

Our ‘CoffeeFragment’ Fragment

```
public class CoffeeFragment  extends ListFragment implements  OnClickListener
{
  protected         Base                activity;
  protected static  CoffeeListAdapter   listAdapter;
  protected         ListView            listView;

  public CoffeeFragment() {...}

  public static CoffeeFragment newInstance() {...}
@Override
  public void onAttach(Context context)
  {...}

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    listAdapter = new CoffeeListAdapter(activity, this, Base.coffeeList);
    setListAdapter (listAdapter);
  }

  @Override
  public void onStart() { super.onStart(); }
  @Override
  public void onClick(View view)
  {...}
  @Override
  public void onListItemClick(ListView l, View v, int position, long id)
  {...}
  public void onCoffeeDelete(final Coffee coffee)
  {...}
}
```

Note the type of Fragment we extend from



Adding a Custom Adapter to our Fragment to manage the list of coffees (more on this later)

Introducing Adapters (Big part of this Case Study)

- **`Adapters`** are bridging classes that bind data to **`Views`** (eg ListViews) used in the UI.
  - Responsible for creating the child Views used to represent each item within the parent View, and providing access to the underlying data
- Views that support adapter binding must extend the **`AdapterView`** abstract class.
  - You can create your own **`AdapterView`**-derived controls and create new **`Adapter`** classes to bind them.
- Android supplies a set of **`Adapters`** that pump data into native UI controls (next slide)

Introducing Adapters (cont'd)

- Because **`Adapters`** are responsible for supplying the data **AND** for creating the Views that represent each item, they can radically modify the appearance and functionality of the controls they're bound to.
- Most Commonly used Adapters
  - **`ArrayAdapter`**
    - uses generics to bind an **`AdapterView`** to an array of objects of the specified class.
    - By default, uses the **`toString()`** of each object to create & populate **`TextViews`**.
    - Other constructors available for more complex layouts (as we will see later on)
    - Can extend the class to use alternatives to simple **`TextViews`** (as we will see later on)
- See also **`SimpleCursorAdapter`** – attaches Views specified within a layout to the columns of Cursors returned from Content Provider queries.

CoffeeMate 2.0

Using Custom ArrayAdapters

Customizing the ArrayAdapter

- By default, the **`ArrayAdapter`** uses the **`toString()`** of the object array it's binding to, to populate the **`TextView`** available within the specified layout
- Generally, you customize the layout to display more complex views by..
  - Extending the **`ArrayAdapter`** class with a type-specific variation, eg

```
public class CoffeeListAdapter extends ArrayAdapter<Coffee> {
```

  - Override the **`getView()`** method to assign object properties to layout View objects. (see our case study example next)

The `getView()` Method

- Used to construct, inflate, and populate the View that will be displayed within the parent **`AdapterView`** class (eg a **`ListView`** inside our **`ListFragment`**) which is being bound to the underlying array using this adapter
- Receives parameters that describes
  - The position of the item to be displayed
  - The **`View`** being updated (or **`null`**)
  - The **`ViewGroup`** into which this new **`View`** will be placed
- Returns the new populated **`View`** instance as a result

- A call to **`getItem()`** will return the value (object) stored *at the specified index in the underlying array*

Adapters & ListViews

- A ListView receives its data via an Adapter. The adapter also defines how each row is the ListView is displayed.
- The Adapter is assigned to the list via the **`setAdapter()`** / **`setListAdapter()`** method on the ListView / ListFragment object.
- ListView calls the **`getView()`** method on the adapter for each data element. In this method the adapter determines the layout of the row and how the data is mapped to the Views (our widgets) in this layout.
- Your row layout can also contain Views which interact with the underlying data model via the adapter. E.G. our 'Delete' option – see later.

CoffeeMate 2.0

Code Highlights (2)

CoffeeListAdapter

```
public class CoffeeListAdapter extends ArrayAdapter<Coffee> {
  private Context context;
  private OnClickListener deleteListener;
  public List<Coffee> coffeeList;

  public CoffeeListAdapter(Context context, OnClickListener deleteListener,
                           List<Coffee> coffeeList) {
    super(context, R.layout.coffeerow, coffeeList);

    this.context = context;
    this.deleteListener = deleteListener;
    this.coffeeList = coffeeList;
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    CoffeeItem item = new CoffeeItem(context, parent, deleteListener,
            coffeeList.get(position));
    return item.view;
  }

  @Override
  public int getCount() { return coffeeList.size(); }
  public List<Coffee> getCoffeeList() { return this.coffeeList; }
  @Override
  public Coffee getItem(int position) { return coffeeList.get(position); }
  @Override
  public long getItemId(int position) { return position; }
  @Override
  public int getPosition(Coffee c) { return coffeeList.indexOf(c); }
}
```

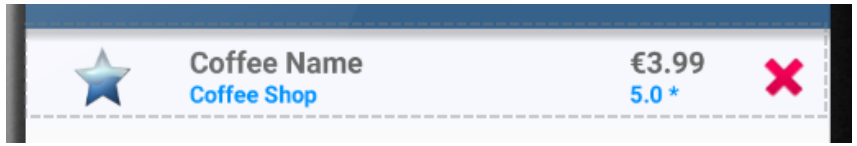
Our constructor, associating our data (our list of Coffees) with the view we want to bind to (coffeerow)

A reference for deleting a coffee

Every time this method is called (based on the position) we create a new 'CoffeeItem' – a new 'Row' to add to the Parent ViewGroup (the ListView)

CoffeeItem

This class represents a single row in our list




```
public class CoffeeItem {
    View view;

    public CoffeeItem(Context context, ViewGroup parent,
            OnClickListener deleteListener, Coffee coffee)
    {
        LayoutInflater inflater = (LayoutInflater) context
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        view = inflater.inflate(R.layout.coffeerow, parent, false);
        view.setId(coffee.coffeeId);

        updateControls(coffee);

        ImageView imgDelete = (ImageView) view.findViewById(R.id.imgDelete);
        imgDelete.setTag(coffee);
        imgDelete.setOnClickListener(deleteListener);
    }

    private void updateControls(Coffee coffee) {
        ((TextView) view.findViewById(R.id.rowCoffeeName)).setText(coffee.name);
        ((TextView) view.findViewById(R.id.rowCoffeeShop)).setText(coffee.shop);
        ((TextView) view.findViewById(R.id.rowRating)).setText(coffee.rating + " *");
        ((TextView) view.findViewById(R.id.rowPrice)).setText("€" +
                new DecimalFormat("0.00").format(coffee.price));

        ImageView imgIcon = (ImageView) view.findViewById(R.id.RowImage);

        if (coffee.favourite == true)
            imgIcon.setImageResource(R.drawable.ic_favourite_on);
        else
            imgIcon.setImageResource(R.drawable.ic_favourite_off);
    }
}
```

Setting the 'Rows' id to the Coffee id for Editing

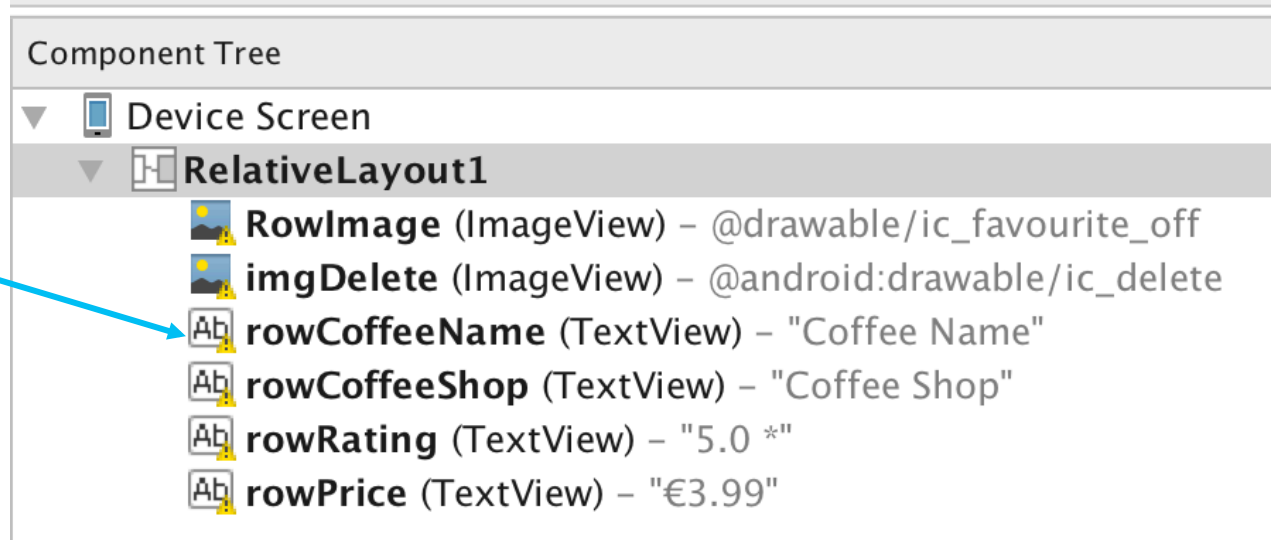
Inflating the 'Current Row'

Updating the 'Row' with Coffee Data

'Tagging' the Delete Image with a Coffee for Deleting

coffeerow (Our Custom Layout)

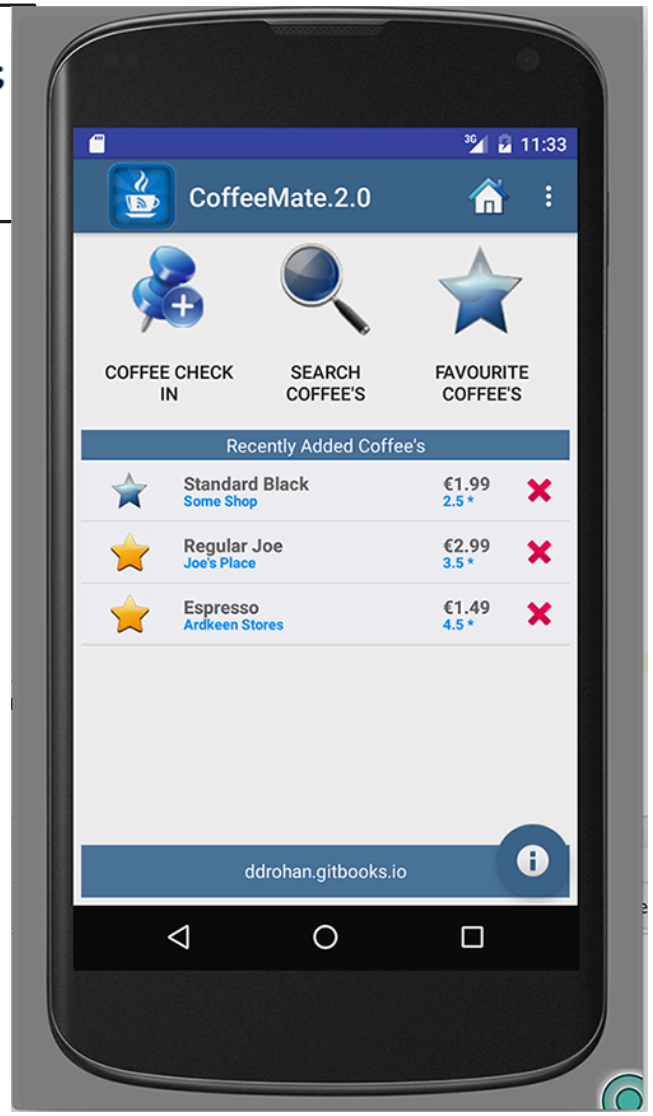
Each time *getView()* is called, it creates a new **CoffeeItem** and binds the individual Views (widgets) above, to each element of the object array in the **ArrayAdapter**.



Resulting ListView (inside our Fragment)

```
public void setupCoffees(){
    coffeeList.add(new Coffee("Standard Black", "Some Shop",2.5,1.99,0));
    coffeeList.add(new Coffee("Regular Joe", "Joe's Place",3.5,2.99,1));
    coffeeList.add(new Coffee("Espresso", "Ardkeen Stores",4.5,1.49,1));
}
```

Our Setup method initially gives us this list



CoffeeMate 2.0

Code Highlights (3)

Edit a Coffee – class CoffeeFragment

```
@Override
public void onListItemClick(ListView l, View v, int position, long id)
{
  Bundle activityInfo = new Bundle();
  activityInfo.putInt("coffeeID", v.getId());

  Intent goEdit = new Intent(getActivity(), Edit.class);
  goEdit.putExtras(activityInfo);
  getActivity().startActivity(goEdit);
}
```

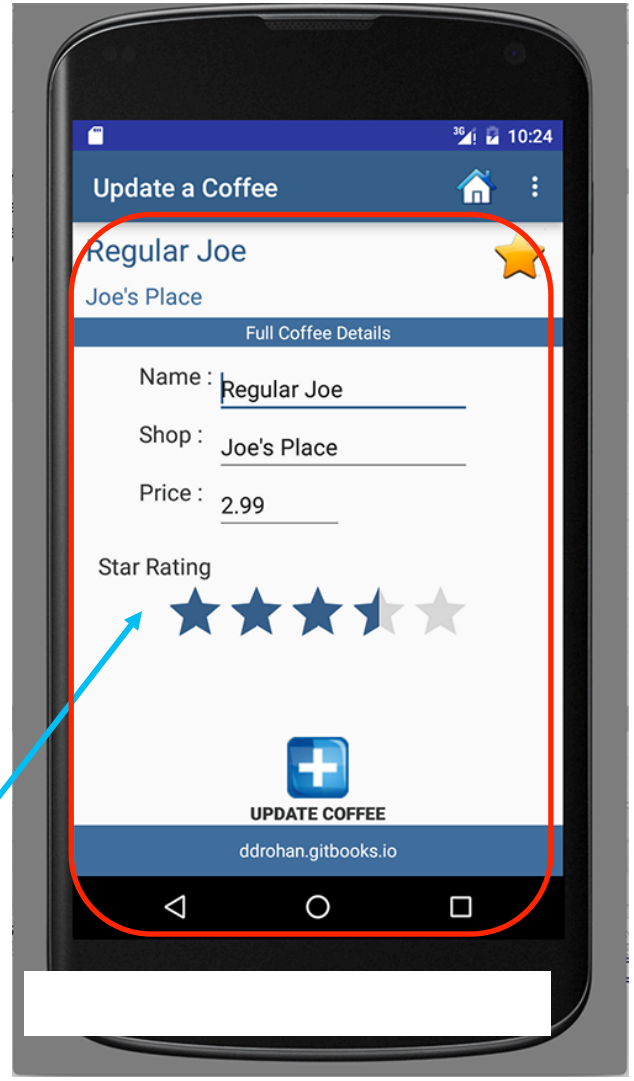
Remember we set the id of the 'row' (v) ? Here we retrieve it, and store it in a Bundle so we know which coffee to edit

Edit a Coffee – class Edit

```
public class Edit extends Base {
    private Context context;
    private Boolean isFavourite;
    private Coffee aCoffee;
    private ImageView favouriteImage;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        context = this;
        setContentView(R.layout.edit);

        activityInfo = getIntent().getExtras();
        aCoffee = getCoffeeObject(activityInfo.getInt("coffeeID"));

        ((TextView)findViewById(R.id.coffeeNameTextView)).setText(aCoffee.name);
        ((TextView)findViewById(R.id.coffeeShopTextView)).setText(aCoffee.shop);

        ((EditText)findViewById(R.id.nameEditText)).setText(aCoffee.name);
        ((EditText)findViewById(R.id.shopEditText)).setText(aCoffee.shop);
        ((EditText)findViewById(R.id.priceEditText)).setText(""+aCoffee.price);
        ((RatingBar) findViewById(R.id.coffeeRatingBar)).setRating((float)aCoffee.rating);

        favouriteImage = (ImageView) findViewById(R.id.favouriteImageView);

        if (aCoffee.favourite == true) {
            favouriteImage.setImageResource(R.drawable.ic_favourite_on);
            isFavourite = true;
        } else {
            favouriteImage.setImageResource(R.drawable.ic_favourite_off);
            isFavourite = false;
        }
    }
```

Retrieving the “id” of our selected coffee from the bundle and finding it in the arraylist

Assigning our Coffee object details to the widgets on our layout

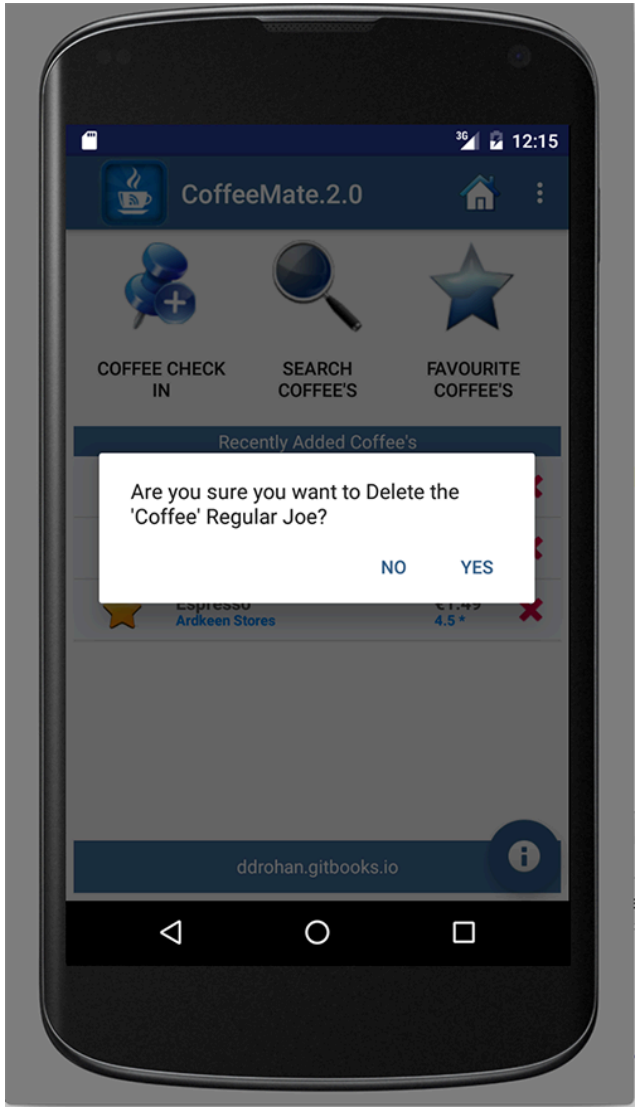


Delete a Coffee – class CoffeeFragment

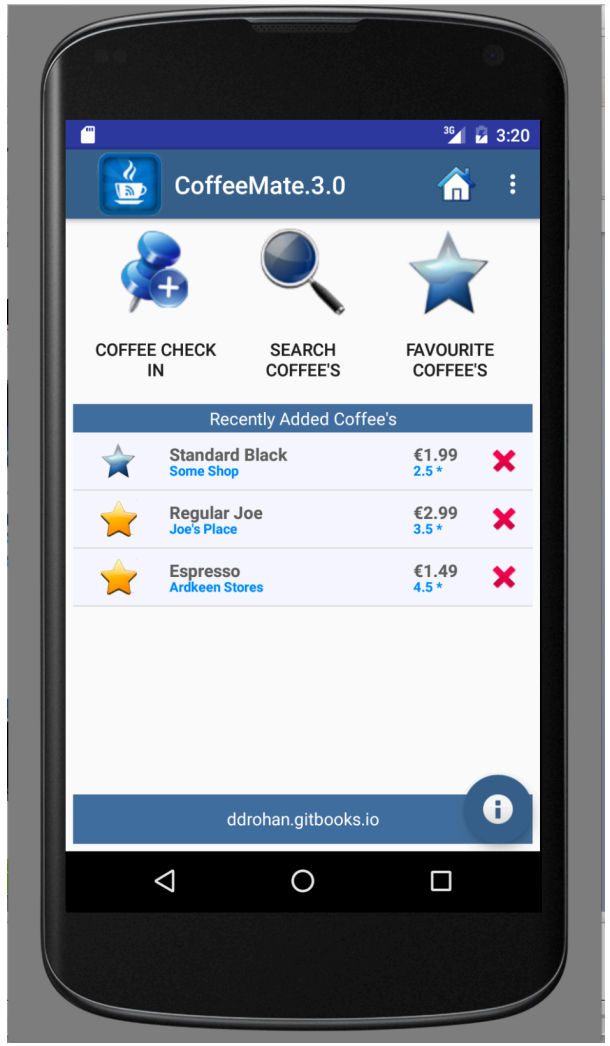
```
@Override
public void onClick(View view)
{
  if (view.getTag() instanceof Coffee)
  {
    onCoffeeDelete ((Coffee) view.getTag());
  }
}


public void onCoffeeDelete(final Coffee coffee)
{
  String stringName = coffee.name;
  AlertDialog.Builder builder = new AlertDialog.Builder(activity);
  builder.setMessage("Are you sure you want to Delete the \'Coffee\' " + stringName + "?");
  builder.setCancelable(false);

  builder.setPositiveButton("Yes", (dialog, id) → {
      Base.coffeeList.remove(coffee); // remove from our list
      listAdapter.coffeeList.remove(coffee); // update adapters data
      listAdapter.notifyDataSetChanged(); // refresh adapter
  }).setNegativeButton("No", (dialog, id) → { dialog.cancel(); });
  AlertDialog alert = builder.create();
  alert.show();
}
```

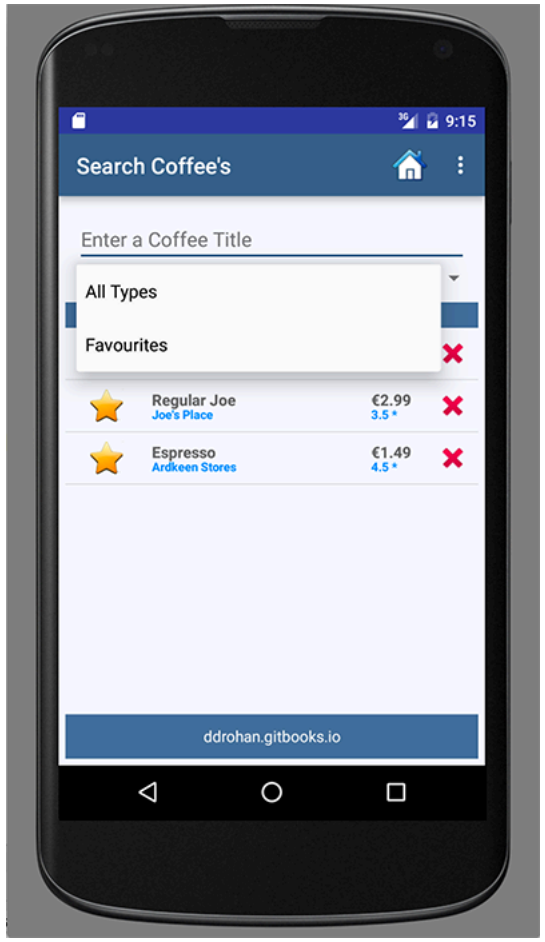
If the Views 'Tag' is a Coffee Object, we know the delete image was clicked, so we can delete the coffee

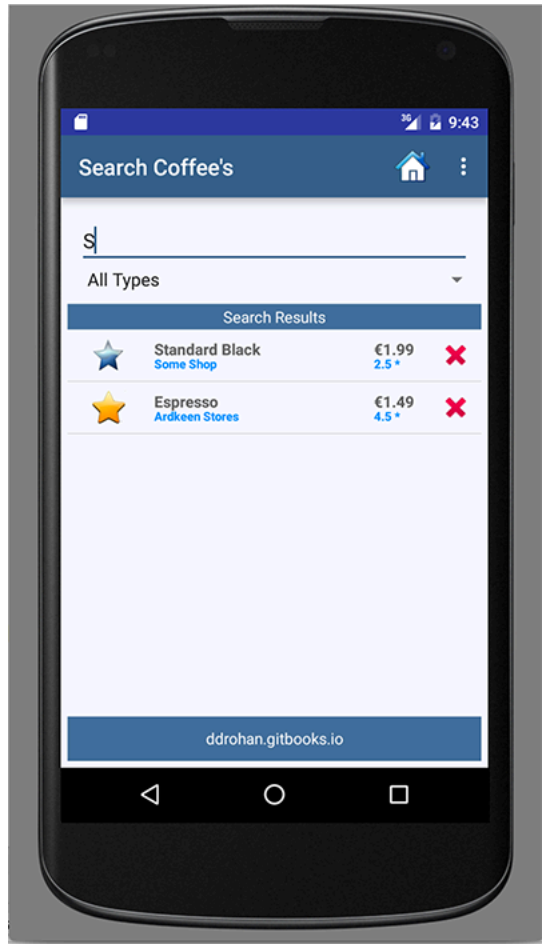
As well as removing the coffee from our global list, we need to remove it from the adapter too
(or otherwise create a whole new adapter reference)

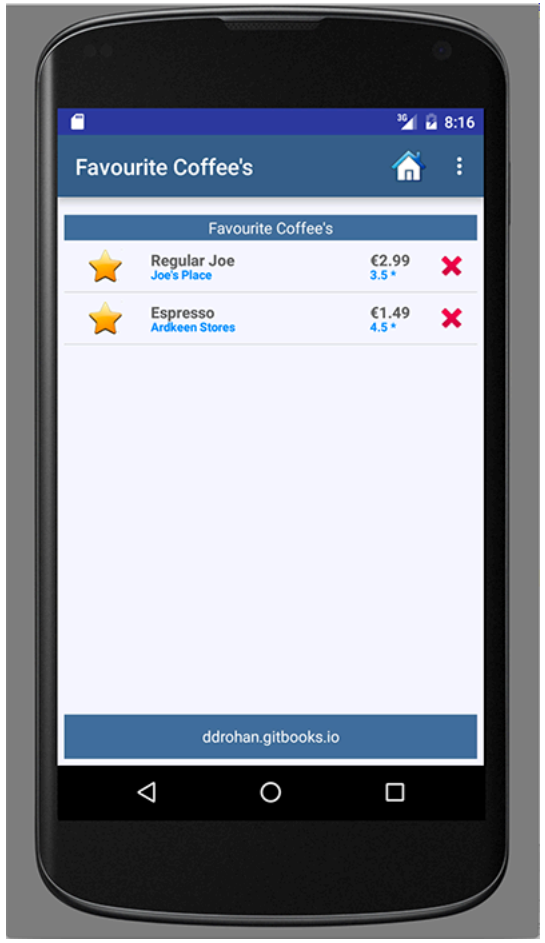


CoffeeMate 3.0









Still no Persistence in this Version

Questions?