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
- Understand how to use a **`Contextual Menu`** to delete multiple items from a list

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

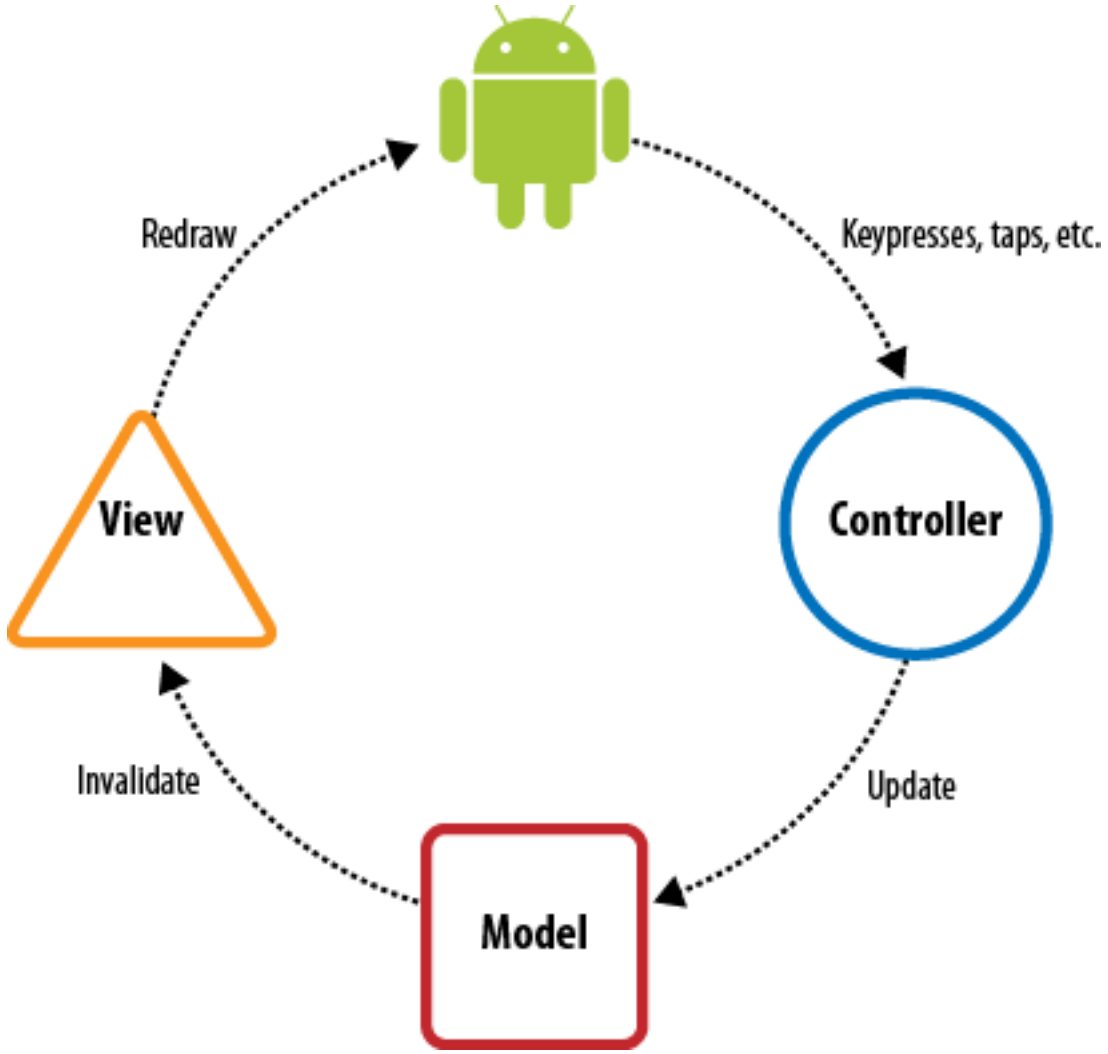
- Keyboard
- "Click" Tap via Touch (or Stylus)
- GPS or Network Location
- Accelerometer Motion
- Orientation / Compass / Altitude
- Vibration
- Sound / Music
- Environment Lighting
- Multi-touch & Gestures
- Device Security / Loss

User Interface

- Your app's user interface is everything that the user can see and interact with. Android provides a variety of pre-built UI components such as structured layout objects and UI controls that allow you to build the graphical user interface for your app. Android also provides other UI modules for special interfaces such as dialogs, notifications, and menus.

App Structure & The Android Framework

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



Some General UI Guidelines & Observations – (UIGOs)

- 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

UIGOs - 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

UIGOs - 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

UIGOs – Layouts (most common)

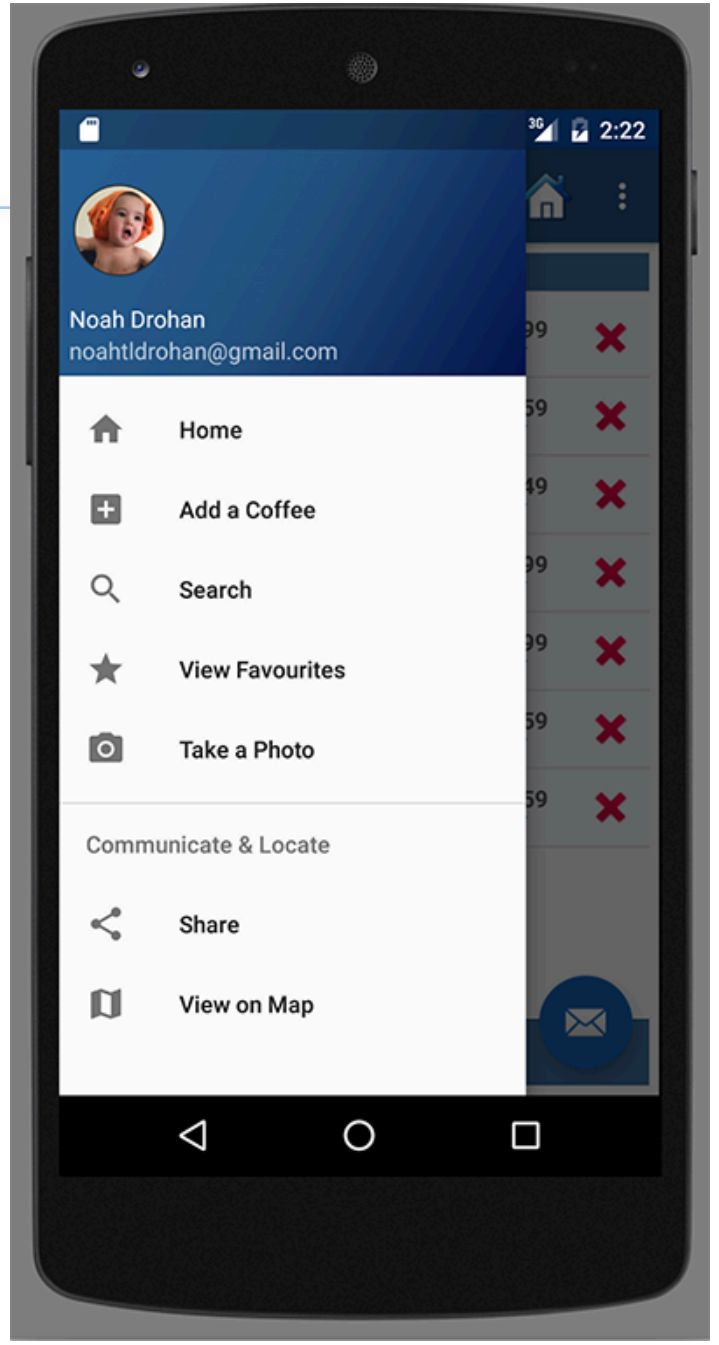
- **LinearLayout**: Each child view is placed after the previous one in a single row or column
- **RelativeLayout**: Each child view is placed in relation to other views in the layout or relative to its parent's layout
- **FrameLayout**: Each child view is stacked within a frame, relative to the top-left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

UIGOs - 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

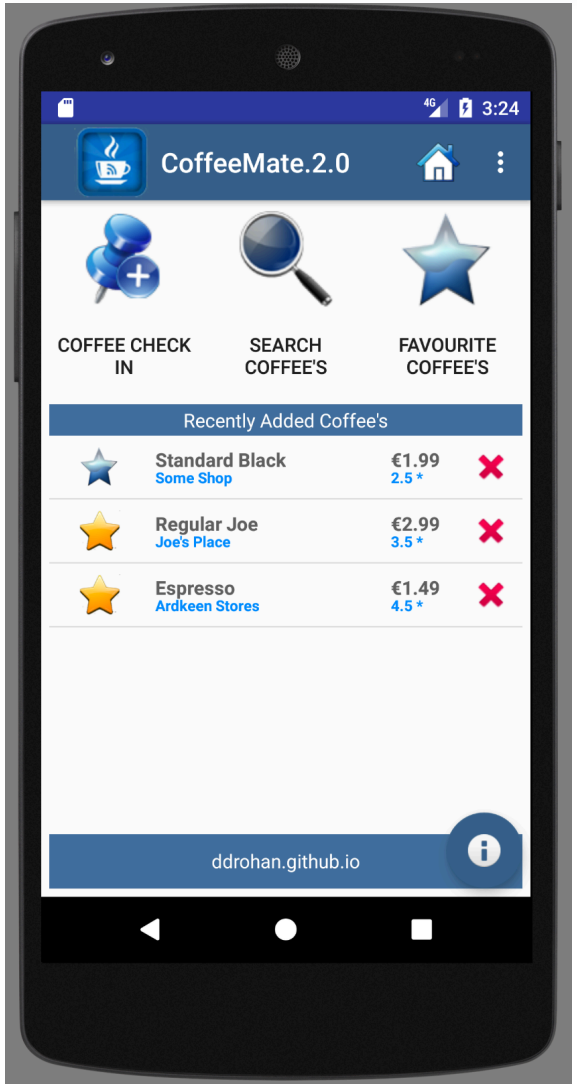
- *CoffeeMate* – an Android App to keep track of your Coffees, their details, and which ones you like the best (your favourites)
- App Features (with Google+ Sign-In)
  - List all your Coffees
  - View specific Coffee details
  - Filter Coffees by Name and Type
  - Delete a Coffee
  - List all your Favourite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

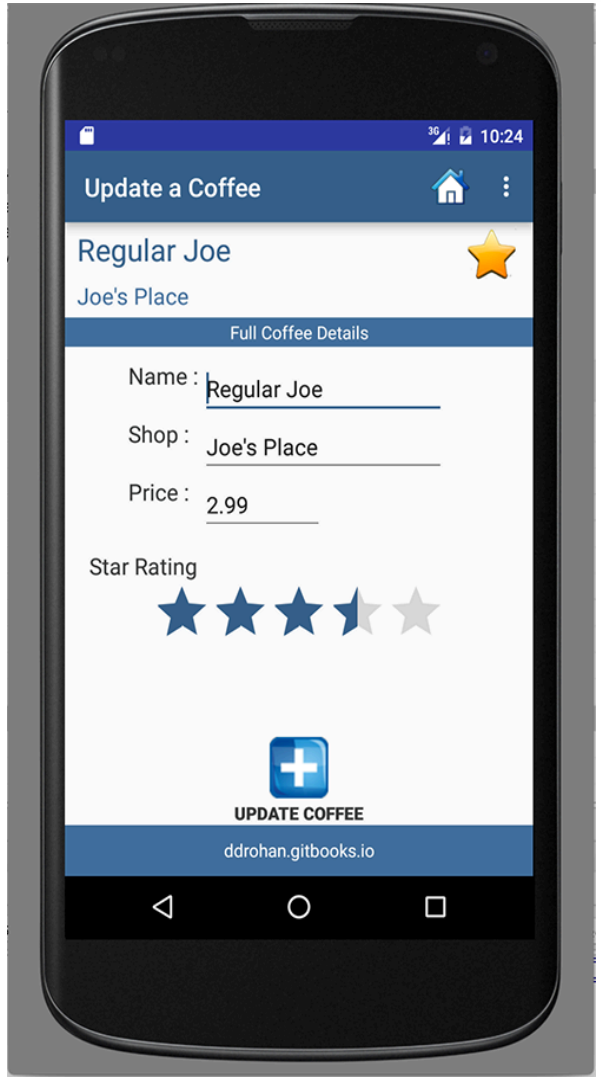


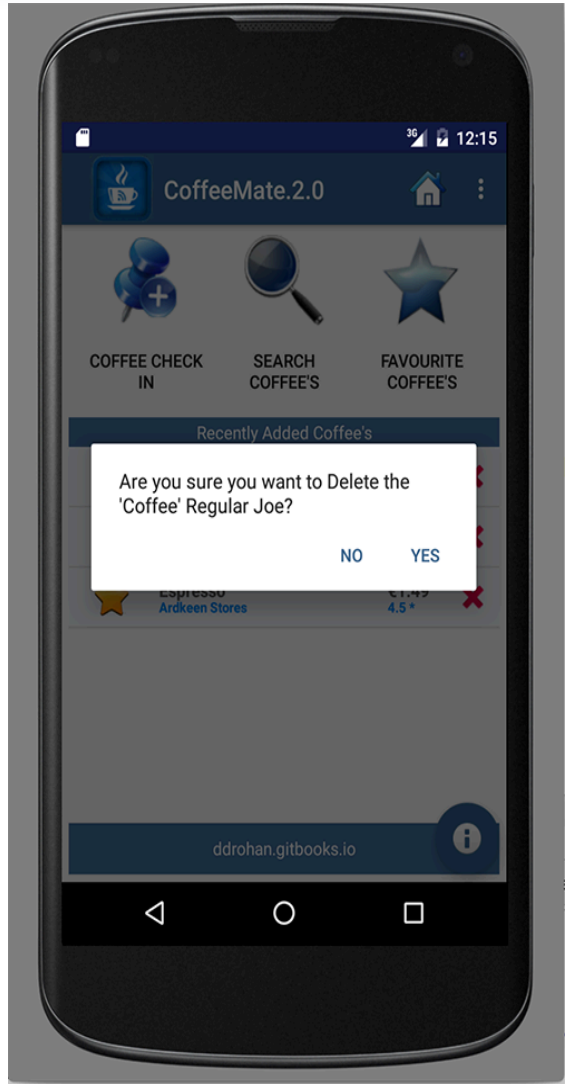
CoffeeMate 2.0

Using Fragments and Custom ArrayAdapters

CoffeeMate 2.0

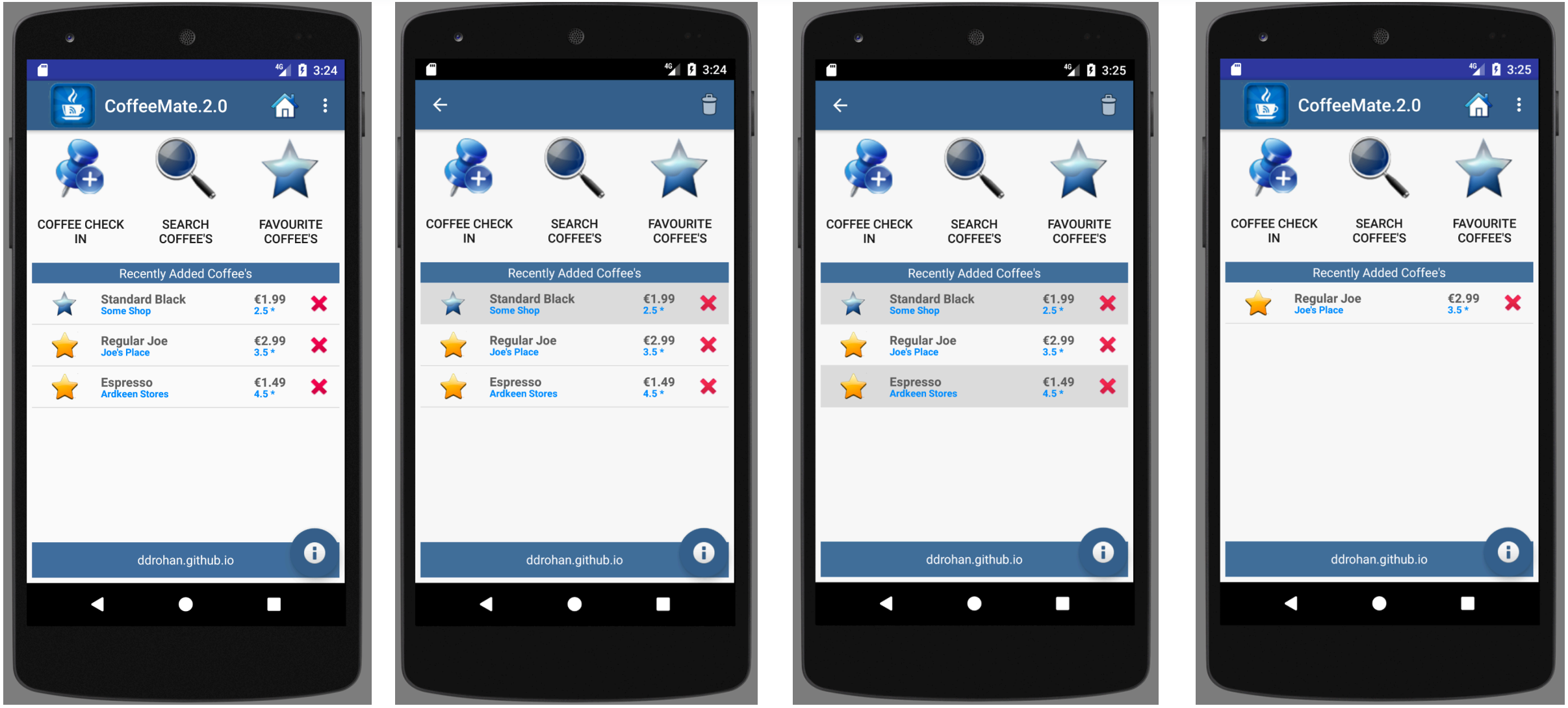






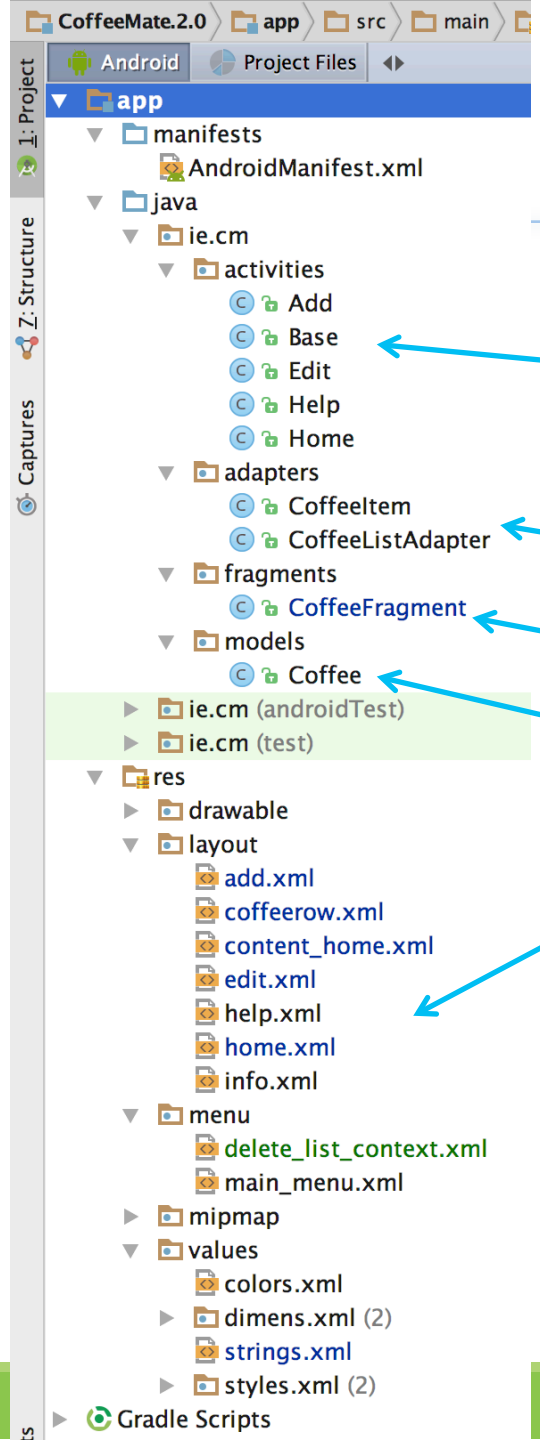
No Persistence in this Version

CoffeeMate 2.0



No Persistence in this Version

CoffeeMate 2.0



- 5 Activity source files
- 9 xml Layouts & Menus
- Custom Adapter classes
- 1 Fragment
- 1 Model

CoffeeMate 2.0

Using Fragments

Fragments - Recap

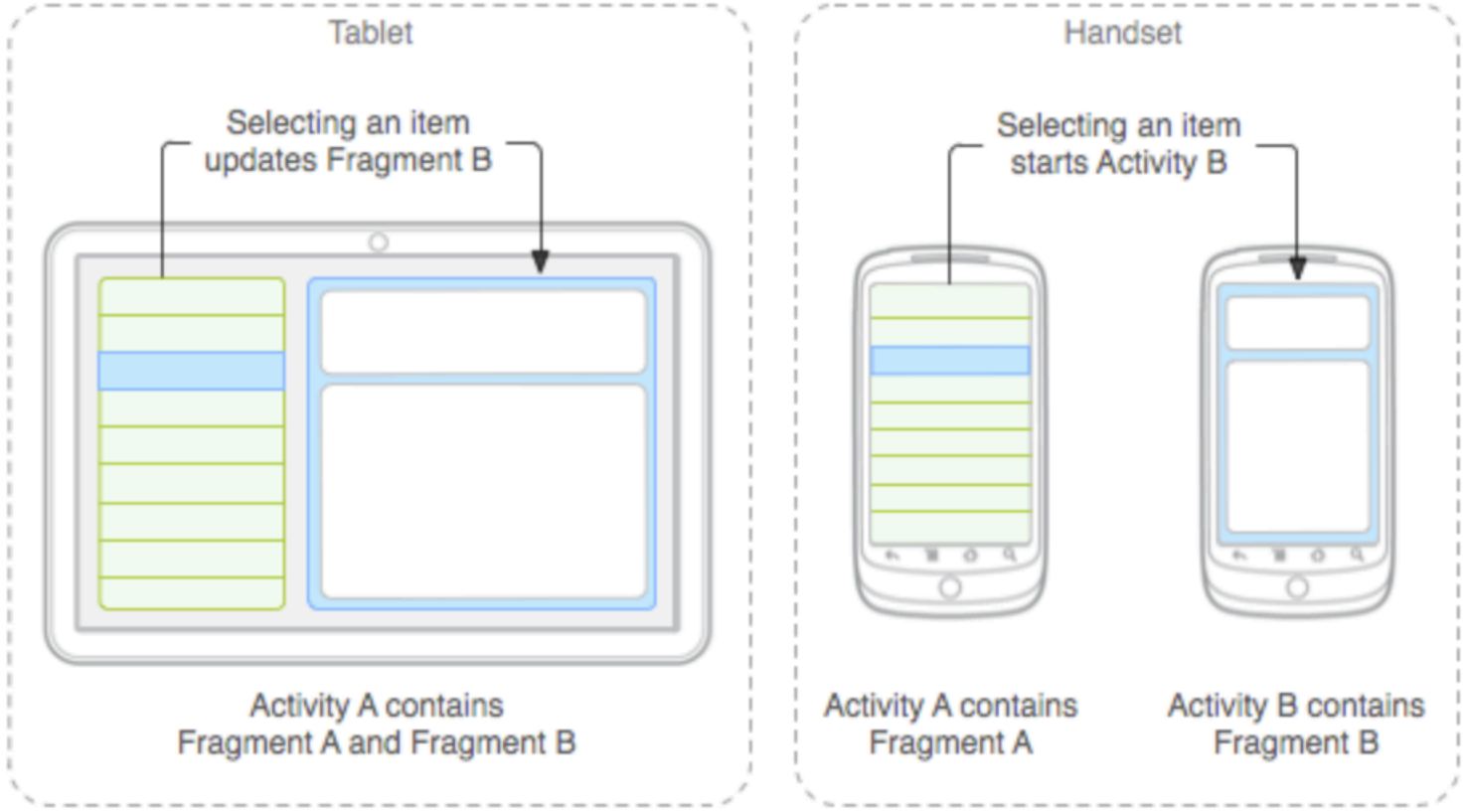
- Fragments represents a behaviour or a portion of a user interface *in an **Activity***.
- You can combine multiple fragments in a single activity and *reuse a single fragment in multiple activities*.
- Each Fragment has its own lifecycle (next slide).
- A fragment must always be embedded in an activity.
- 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.

Designing Fragments *

- You should design each fragment as a modular and reusable activity component.
- When designing your application to support both tablets and handsets, you can ***reuse your fragments*** in different layout configurations to optimize the user experience based on the available screen space.

- For example, on a handset, it might be necessary for separate fragments to provide a single-pane UI when more than one cannot fit within the same activity. (Next Slide)

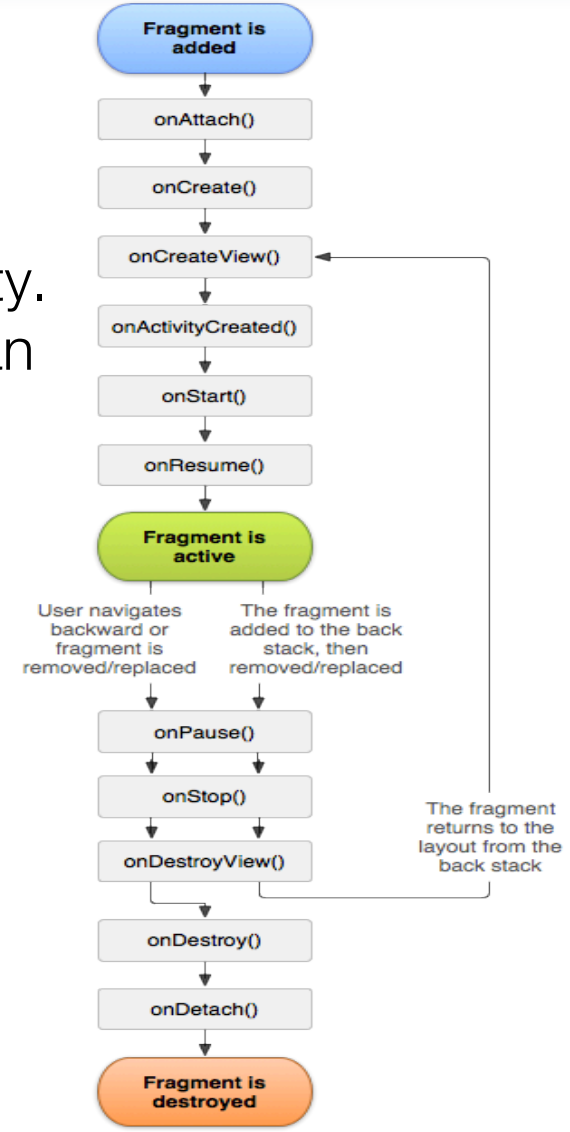
Designing Fragments



An example of how two UI modules defined by fragments can be combined into one activity for a tablet design, but separated for a handset design.

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()*



Fragment Managers & Transactions

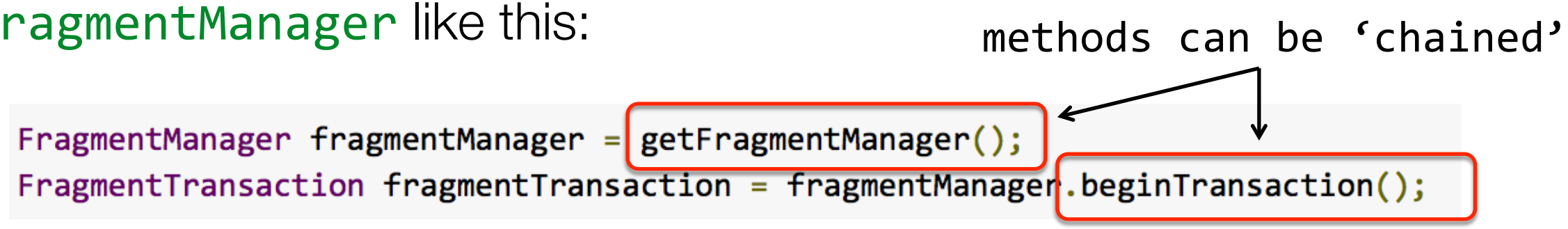
- A great feature about using fragments in your activity is the ability to add, remove, replace, and perform other actions with them, in response to user interaction.

- Each set of changes that you commit to the activity is called a transaction and you can perform one by using APIs in `FragmentTransaction`.

- You can also save each transaction to a back stack managed by the activity, allowing the user to navigate backward through the fragment changes (similar to navigating backward through activities).

Fragment Managers & Transactions *

- You can acquire an instance of `FragmentTransaction` from the `FragmentManager` like this:



- Each transaction is a set of changes that you want to perform at the same time. You can set up all the changes you want to perform for a given transaction using methods such as `add()`, `remove()`, and `replace()`.

- Then, to apply the transaction to the activity, you must call `commit()`.

Fragment Managers & Transactions

- Before you call `commit()`, however, you might want to call `addToBackStack()`, in order to add the transaction to a back stack of fragment transactions.

- This back stack is managed by the activity and allows the user to return to the previous fragment state, by pressing the Back button.

- For example, here's how you can replace one fragment with another, and preserve the previous state in the back stack: (next slide)

Fragment Managers & Transactions *

```
// Create new fragment and transaction
Fragment newFragment = new ExampleFragment();
FragmentTransaction transaction = getFragmentManager().beginTransaction();

// Replace whatever is in the fragment_container view with this fragment,
// and add the transaction to the back stack
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);

// Commit the transaction
transaction.commit();
```

- In this example, `newFragment` replaces whatever fragment (if any) is currently in the layout container identified by the `R.id.fragment_container` ID. By calling `addToBackStack()`, the replace transaction is saved to the back stack so the user can reverse the transaction and bring back the previous fragment by pressing the Back button.

CoffeeMate 2.0

Code Highlights (1)

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

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(){...}
}
```

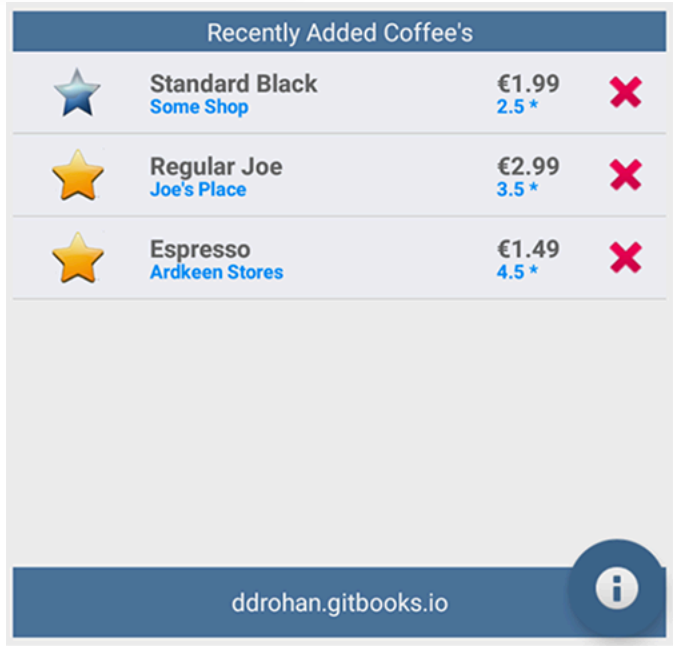
Creating a Fragment instance and adding it to our Home Activity (we'll take a close look at the Fragment class next)

Note how we've 'chained' the method calls

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); }
}
```

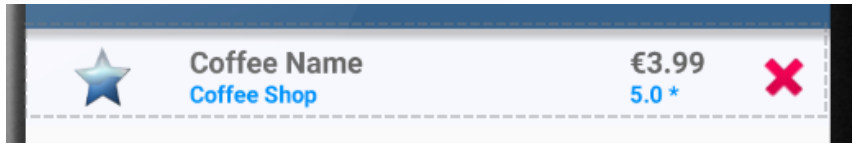
Our constructor, associating our data (our list of Coffees) with the view we want to bind to (coffeerow)

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

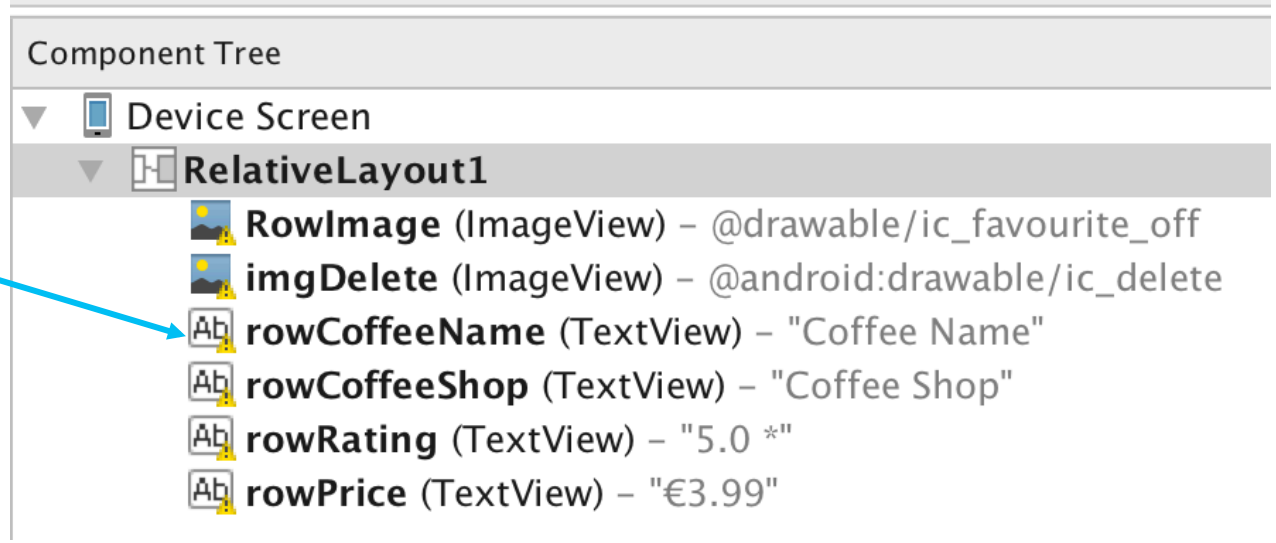
Inflating the 'Current Row'

Updating the 'Row' with Coffee Data

'Tagging' the Delete Image with a Coffee for Deleting

coffeerow (Our Custom Layout)

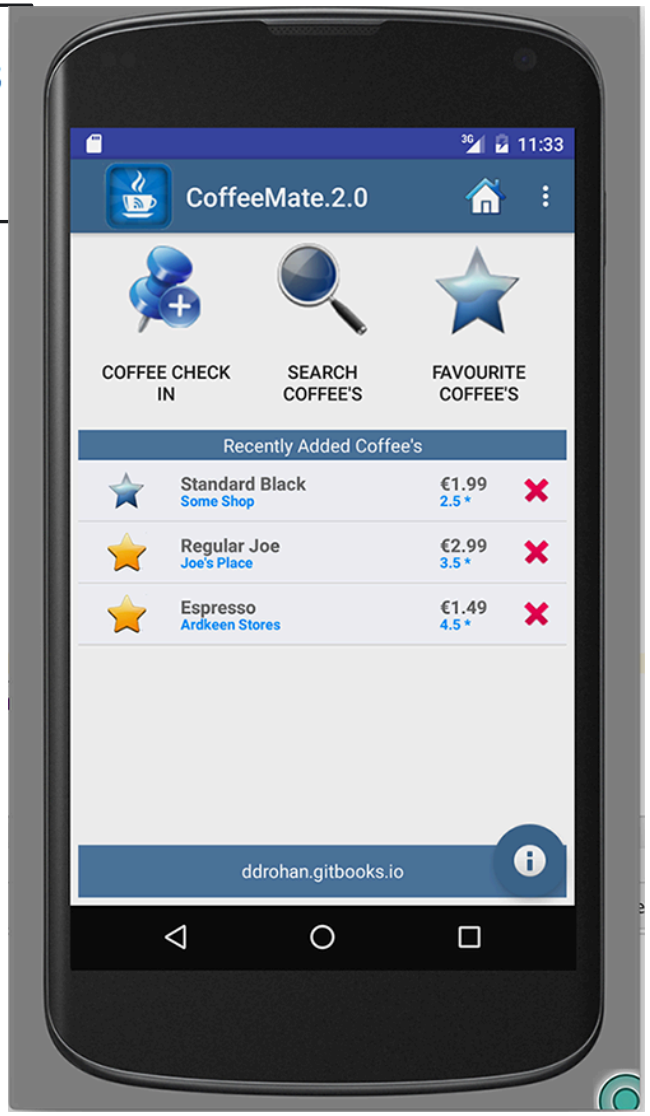
Each time *getView()* is called, it creates a new **CoffeeItem** and binds the individual Views (widgets) above, to each element of the object array in the **ArrayAdapter**.



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);
}
```

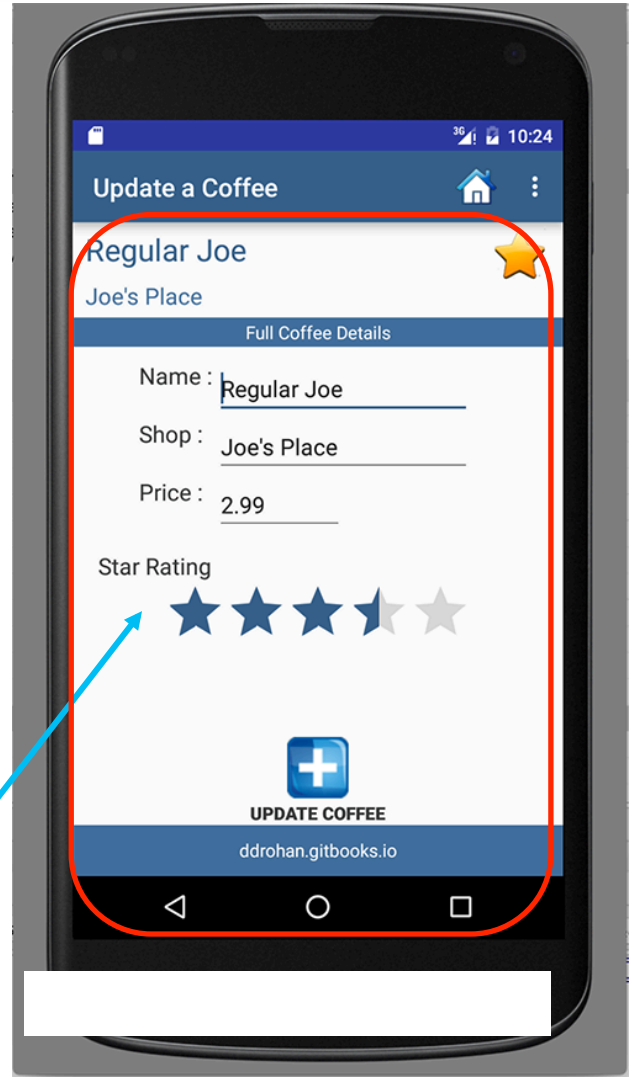
Remember we set the id of the 'row' (v) ? Here we retrieve it, and store it in a Bundle so we know which coffee to edit

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

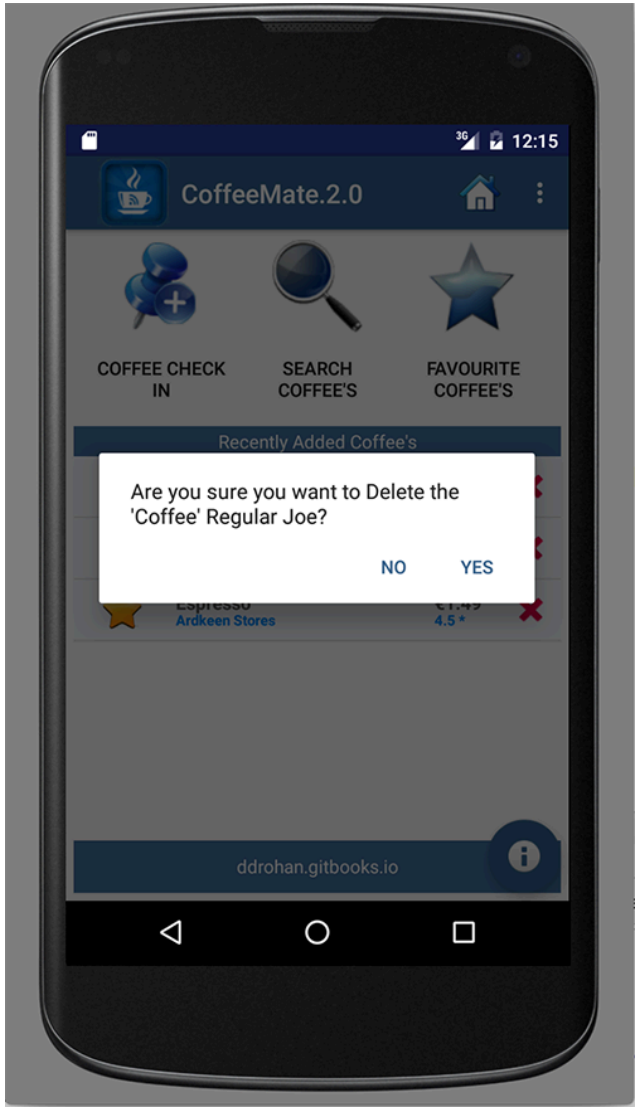


Delete a Coffee – class CoffeeFragment

```
@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 2.0

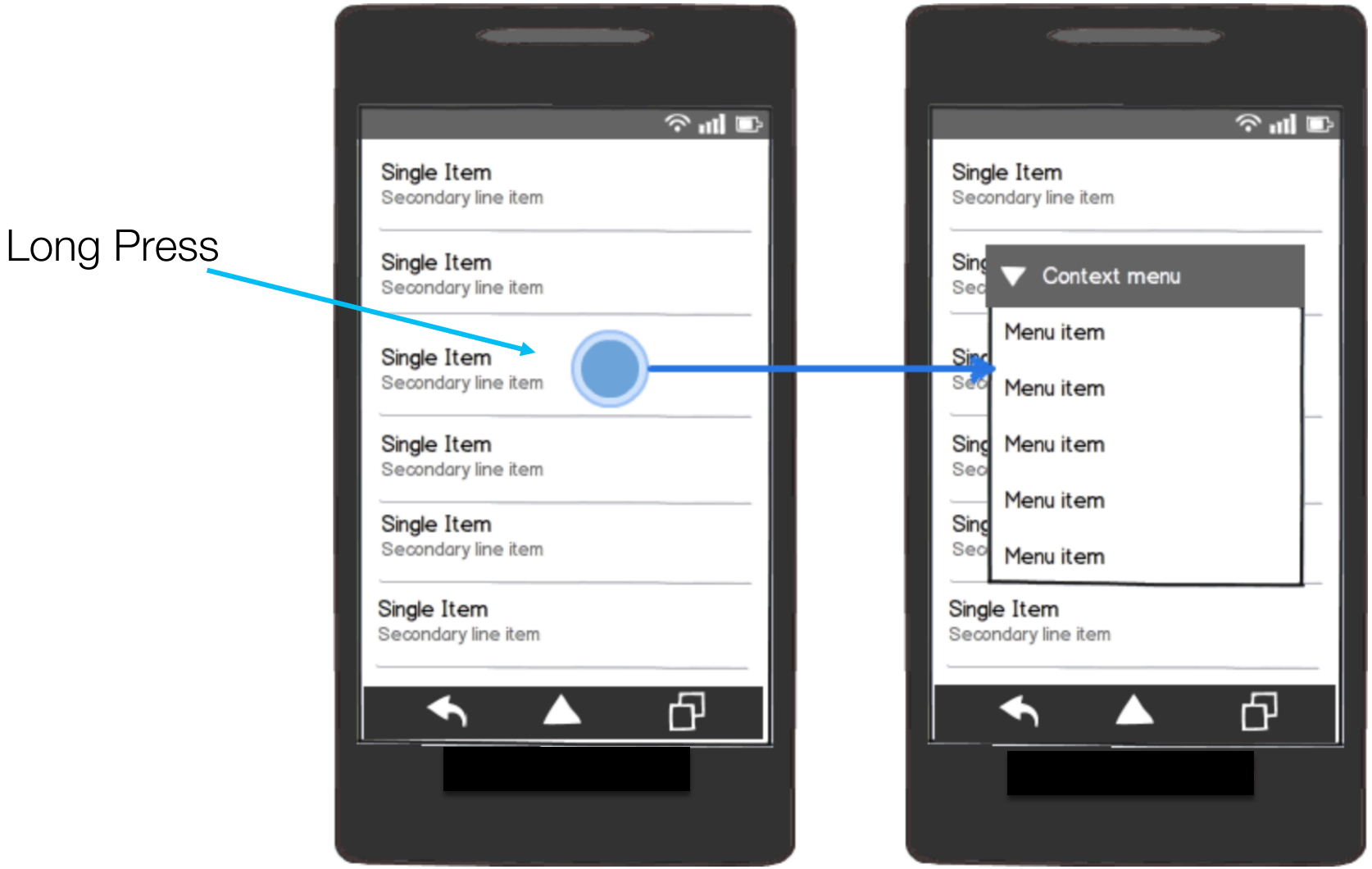
Using Contextual Menus

(for multiple selection & deletion)

Contextual ActionMode / Action Bar

- The Contextual Action Mode is a system implementation of ActionMode that focuses user interaction toward performing contextual actions.
- When a user enables this mode by selecting an item, a Contextual Action Bar appears at the top of the screen to present actions the user can perform on the currently selected item(s).
- A Contextual Action Bar (CAB) in Android is a temporary action bar that overlays the app's action bar for the duration of a particular sub-task.
- Triggered by **Long Press Gesture** - used to handle multi-select and contextual actions.
- **Prior to Android 3.0, a Floating Context Menu would have been displayed on the Long Press Gesture.**

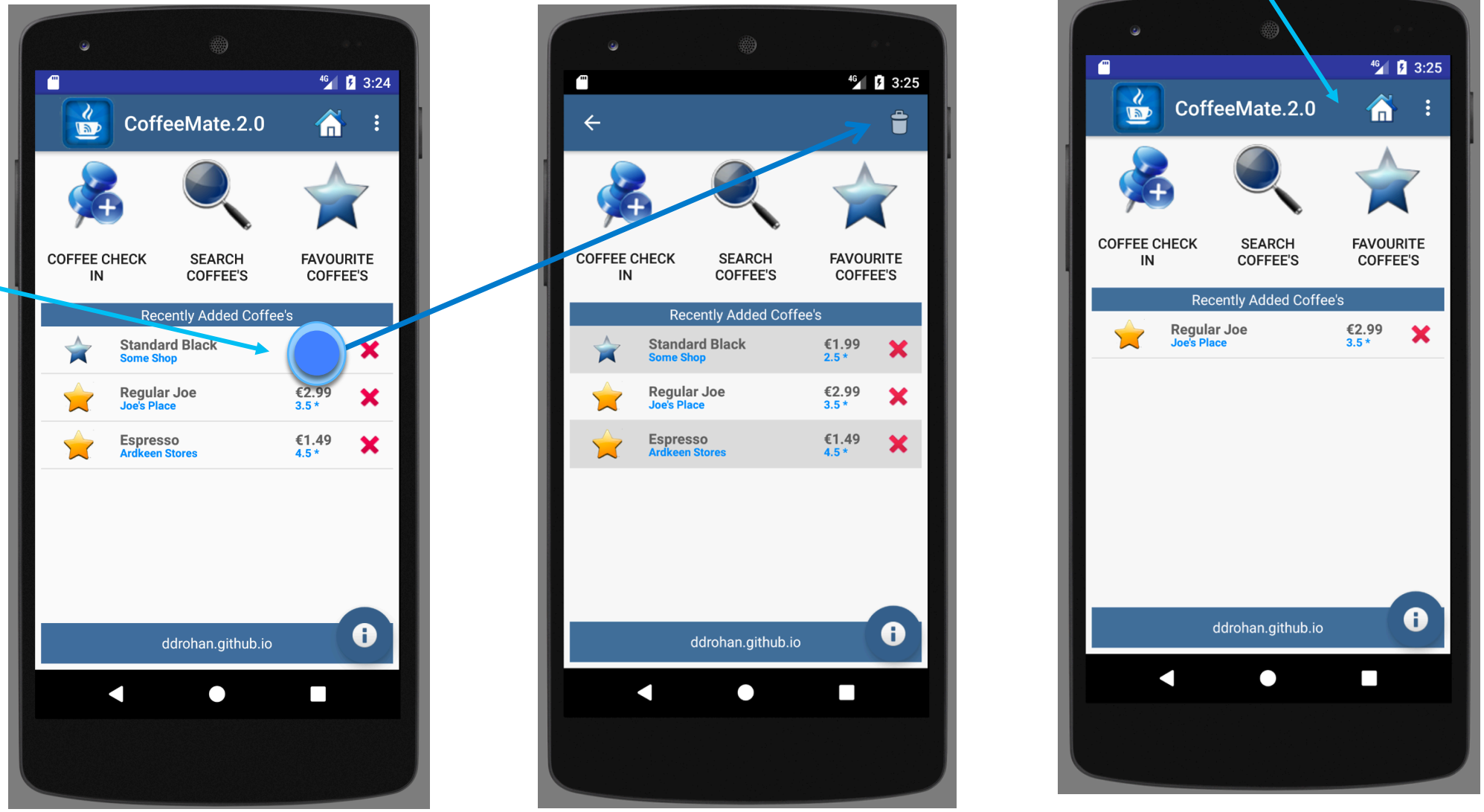
1. Floating Context Menu (Android 3.0-)



2. Contextual Action Bar (Android 3.0+)

Original Menu restored upon deletion

Long Press



Implementing a Contextual Action Bar

1. Design your resources/xml (Context Menu, background style, AppTheme style options).
2. Implement the **`AbsListView.MultiChoiceModeListener`** and set it to your ViewGroup (e.g. ListView).
3. Configure your ViewGroup for multiple selection of items.
4. Implement the necessary behaviour in the Listener Callbacks.

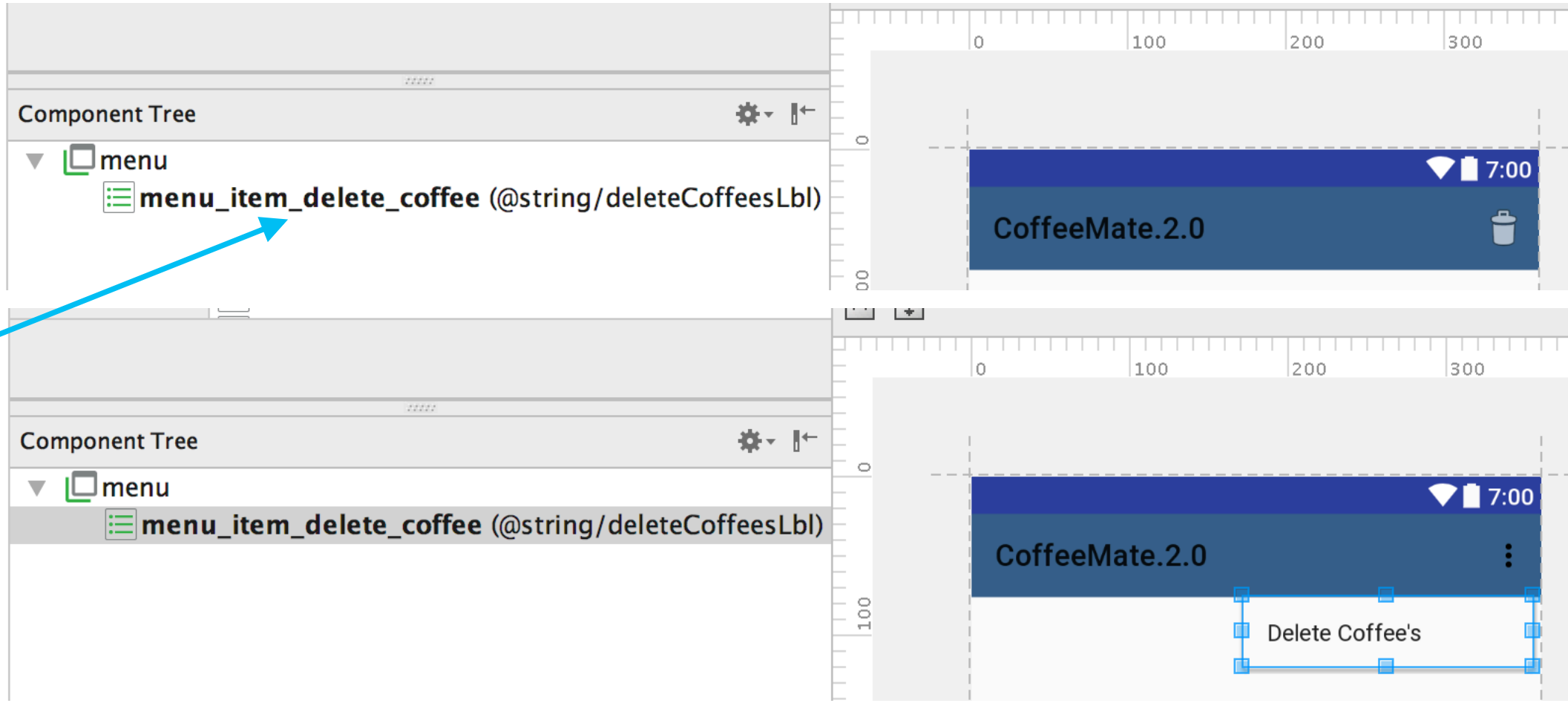
CoffeeMate 2.0

Code Highlights (4)

1. Design your resources / xml

- Firstly, decide what you want your context menu to look like

Menu ID



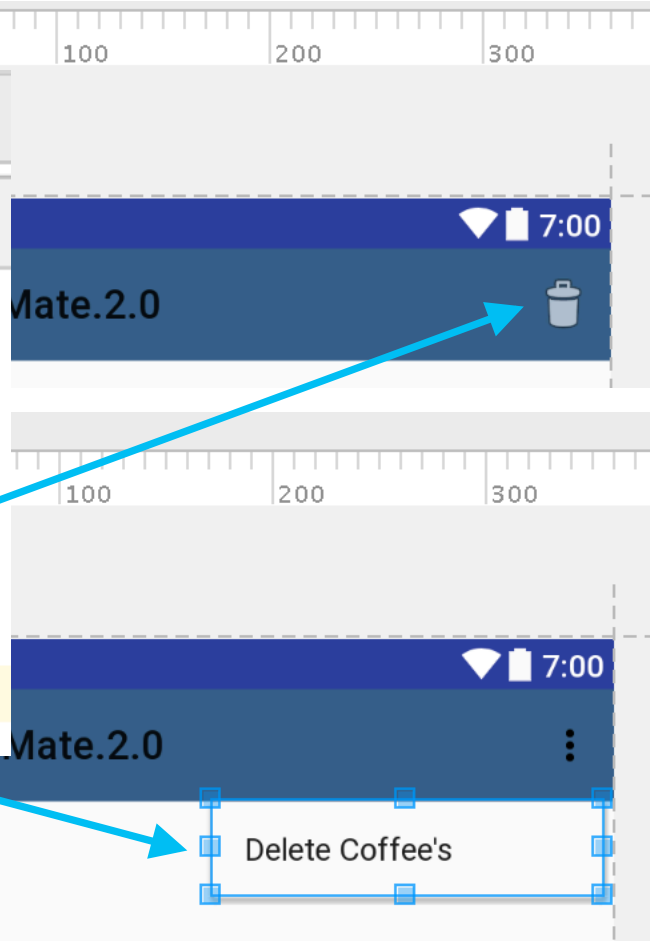
1. Design your resources / xml

- delete_list_context.xml

delete_list_context.xml

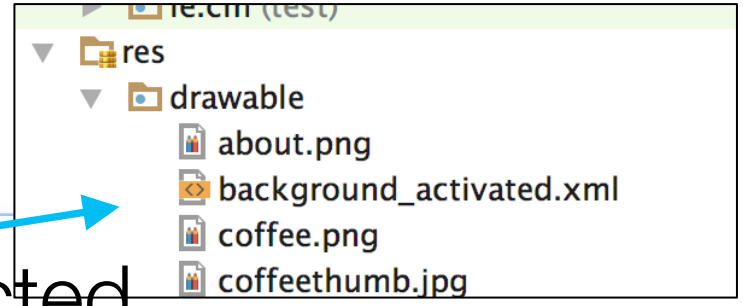
```
menu
1  <?xml version="1.0" encoding="utf-8"?>
2  <menu xmlns:app="http://schemas.android.com/apk/res-auto"
3      xmlns:android="http://schemas.android.com/apk/res/android">
4      <item
5          android:id="@+id/menu_item_delete_coffee"
6          android:icon="@android:drawable/ic_menu_delete"
7          android:title="Delete Coffee's" />
8  </menu>
```

Menu ID



1. Design your resources / xml

- And the items background style, when selected



background_activated.xml

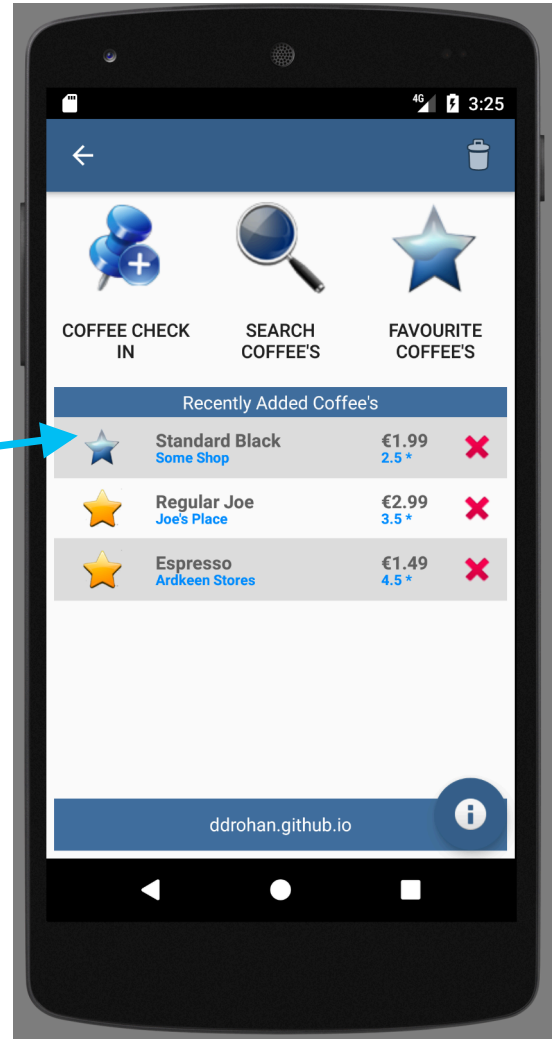
selector | item

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <selector xmlns:android="http://schemas.android.com/apk/res/android" >
3      <item
4          android:state_activated="true"
5          android:drawable="@color/appFooterBGColor"
6          />
7  </selector>
8
```

coffeerow.xml

RelativeLayout | ImageView

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:tools="http://schemas.android.com/tools"
4      android:id="@+id/RelativeLayout1"
5      style="@style/appLayoutTheme"
6      android:layout_width="fill_parent"
7      android:layout_height="46dp"
8      android:background="@drawable/background_activated"
9      android:padding="8dp" >
```



1. Design your resources / xml

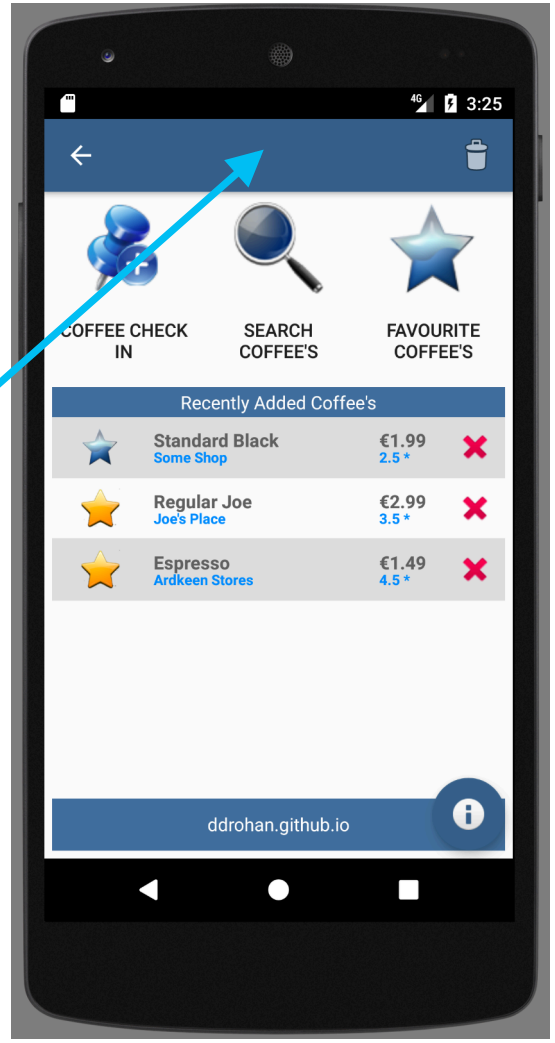
- And how to overlay the context menu on the main app menu (in styles.xml)

styles.xml

Edit all themes in the project in the theme editor.

resources | style | item

```
1  <resources>
2
3      <!-- Base application theme. -->
4      <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
5          <!-- Customize your theme here. -->
6          <item name="colorPrimary">@color/colorPrimary</item>
7          <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
8          <item name="colorAccent">@color/colorAccent</item>
9          <item name="windowActionModeOverlay">true</item>
10         <item name="actionModeBackground">@color/colorPrimary</item>
11     </style>
```



2. Implement `AbsListView.MultiChoiceModeListener`

- First, update your Activity/Fragment *

```
public class CoffeeFragment  extends ListFragment implements  OnClickListener,
                                                    AbsListView.MultiChoiceModeListener
{
```

- We'll look at implementing the callbacks a bit later (just accept the default implementations for now)

3. Configure your ViewGroup *

- Set your listener to the ViewGroup (or ListView in our case) like so and specify multiple selections is possible

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup parent, Bundle savedInstanceState) {
  View v = super.onCreateView(inflater, parent, savedInstanceState);

  listView = (ListView) v.findViewById(android.R.id.list);
  listView.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
  listView.setMultiChoiceModeListener(this);

  return v;
}
```

Note, Android's system List reference

4. Implement Listener Callbacks *

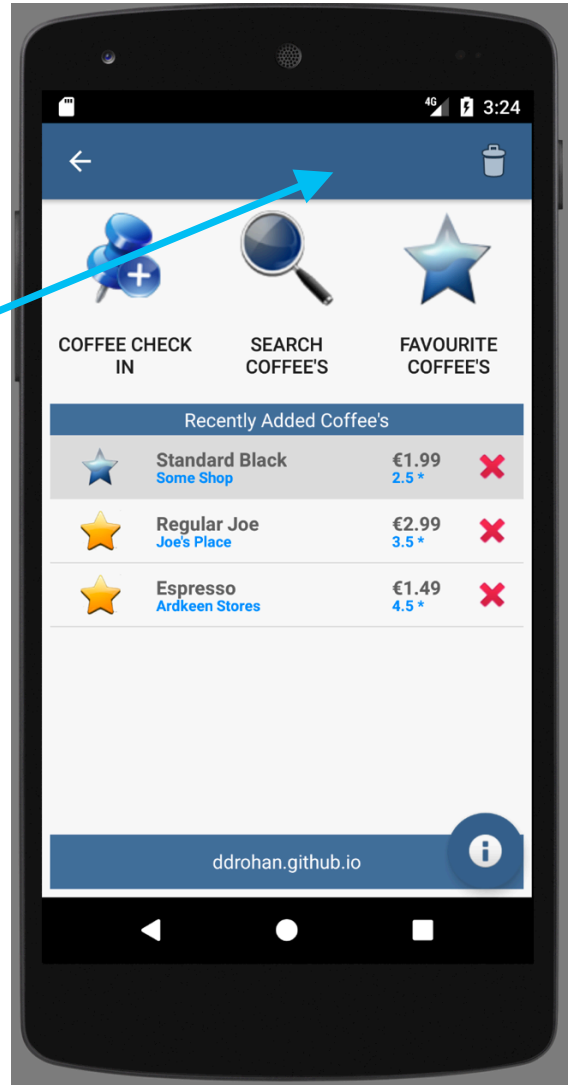
- Here's a full list of all the callback methods (and one helper)

```
/* ************ MultiChoiceModeListener methods (begin) *********** */
@Override
public boolean onCreateActionMode(ActionMode actionMode, Menu menu)
{...}

@Override
public boolean onPrepareActionMode(ActionMode actionMode, Menu menu) { return false; }

@Override
public boolean onActionItemClicked(ActionMode actionMode, MenuItem menuItem)
{...}

private void deleteCoffees(ActionMode actionMode)
{...}

@Override
public void onDestroyActionMode(ActionMode actionMode)
{}

@Override
public void onItemCheckedStateChanged(ActionMode actionMode, int position, long id, boolean checked)
{}
/* ************ MultiChoiceModeListener methods (end) *********** */
```

4. Implement Listener Callbacks

- Inflating the Context Menu

```
@Override
public boolean onCreateActionMode(ActionMode actionMode, Menu menu)
{
   MenuInflater inflater = actionMode.getMenuInflater();
   inflater.inflate(R.menu.delete_list_context, menu);
   return true;
}
```

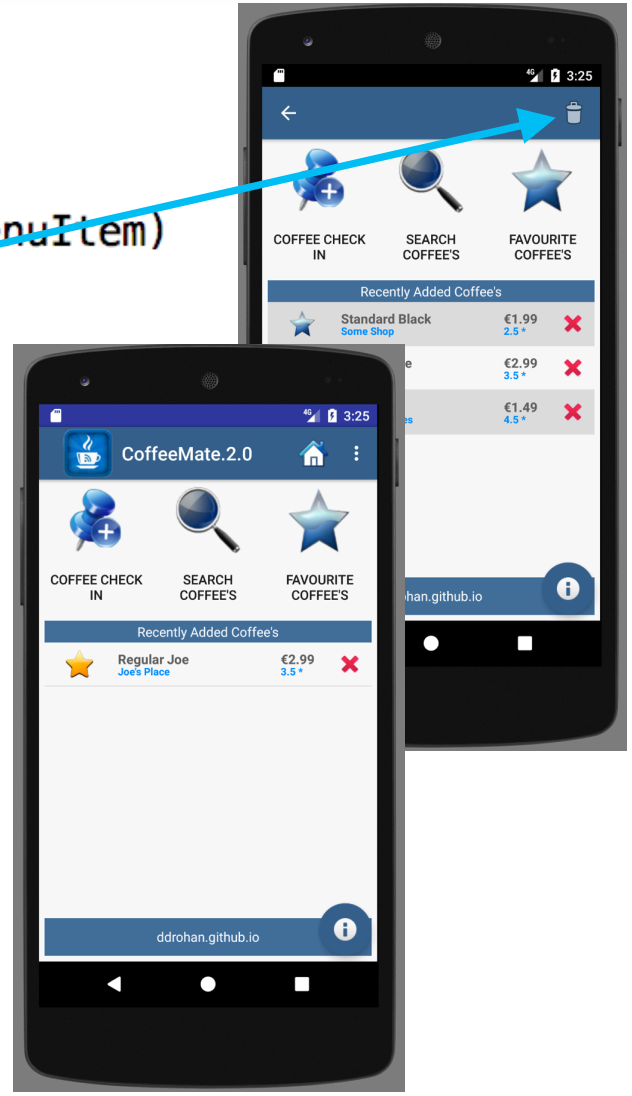


- This replaces or ‘overlays’ the current app menu

4. Implement Listener Callbacks

- Triggered when MenuItem selected

```
@Override
public boolean onActionItemClicked(ActionMode actionMode, MenuItem menuItem)
{
  switch (menuItem.getItemId())
  {
    case R.id.menu_item_delete_coffee:
      deleteCoffees(actionMode);
      return true;
    default:
      return false;
  }

}
```

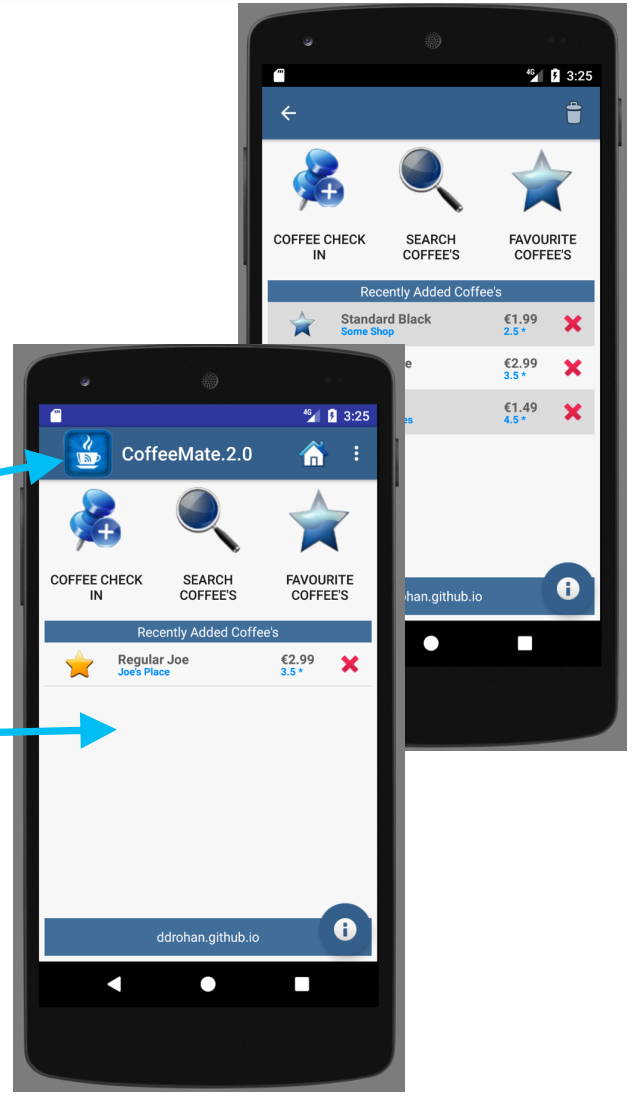


- In our case, we delete the selected coffees

4. Implement Listener Callbacks

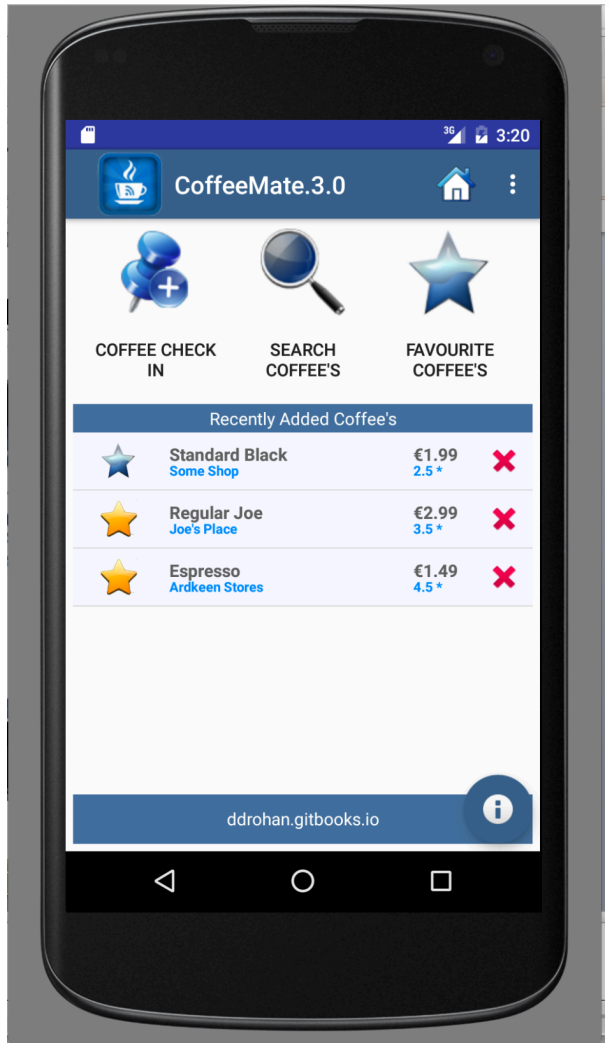
- Triggered when **MenuItem** selected

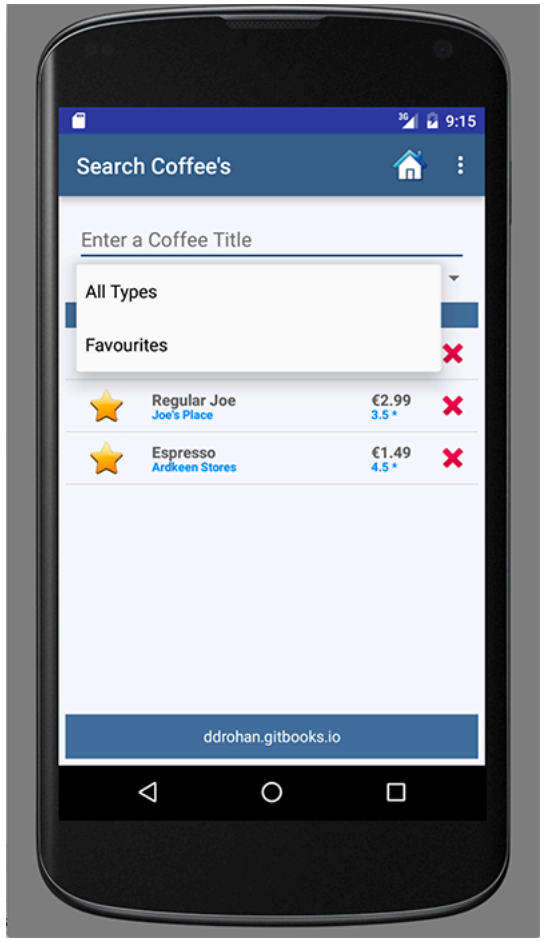
```
private void deleteCoffees(ActionMode actionMode)
{
  for (int i = listAdapter.getCount() - 1; i >= 0; i--)
  {
    if (listView.isItemChecked(i))
    {
      Base.coffeeList.remove(listAdapter.getItem(i));
    }
  }
  actionMode.finish();
  listAdapter.notifyDataSetChanged();
}
```

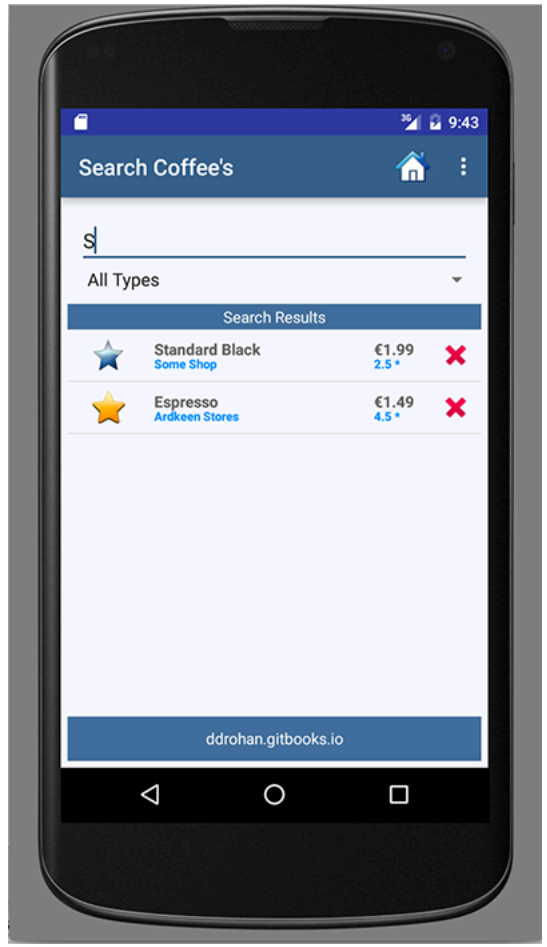


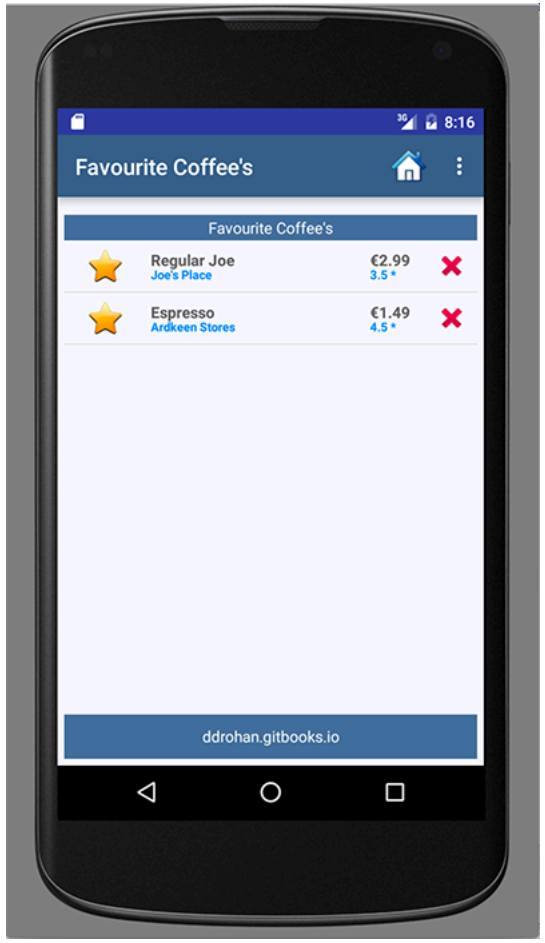
- In our case, we delete the selected coffees

CoffeeMate 3.0









Still no Persistence in this Version

Questions?