Services, Menus, Dialogs and Fragments

CE881: Mobile and Social Application Programming

Spyros Samothrakis

Febrary 01, 2016

Theme: "Social apps"

- The social network (movie)

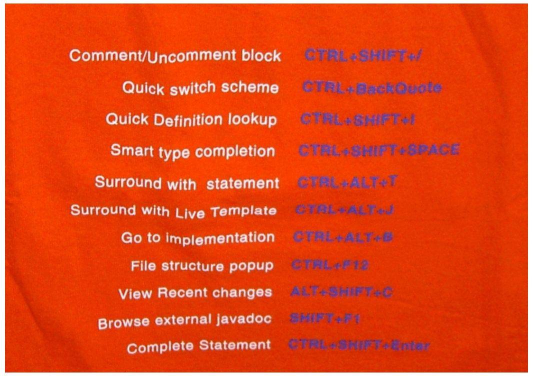
Apps

- Facebook
- Twitter
- Pinterest
- OkCupid
- Instagram

Where's the value?

In case I haven't annoyed you enough...

- Learn how to touch type
- Ctrl+Shift+A (Meta - search for shortcut/action)
- Ctrl+B (Go to declaration)
- Ctrl+U (Go to superclass)
- Ctrl+J (Insert template)



Comment/Uncomment block	CTRL+SHIFT+/
Quick switch scheme	CTRL+BackQuote
Quick Definition lookup	CTRL+SHIFT+I
Smart type completion	CTRL+SHIFT+SPACE
Surround with statement	CTRL+ALT+T
Surround with Live Template	CTRL+ALT+J
Go to implementation	CTRL+ALT+B
File structure popup	CTRL+F12
View Recent changes	ALT+SHIFT+C
Browse external javadoc	SHIFT+F1
Complete Statement	CTRL+SHIFT+Enter

http://stackoverflow.com/questions/294167/what-are-the-most-useful-intellij-idea-keyboard-

Progress Test

- Next week
- Sample progress test online
- 20 Questions
- 30 Minutes

What is a service?

- App components
- Stay in the background
- Provide a long-running support for the app

Why us them?

- Runs in the background as normal even if the app is minimised
- Not on it's own thread (unless explicitly programmed to do so)
- Exposes non-visual functionality to third parties
- Allows proper interprocess communication (if needed)

Can you think of some interesting services?

Declaring a service

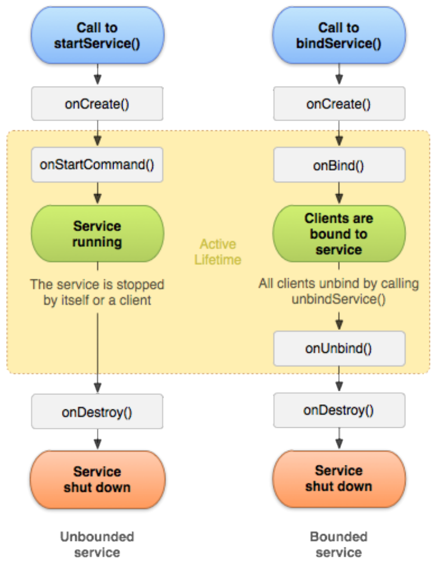
- See here for more details, we will go through some

```
<manifest ... >
  ...
  <application ... >
      <service android:name="com.bob.megaservice" />
      ...
  </application>
</manifest>
```

The service life cycle (1)

```
public class ExampleService extends Service {
    int mStartMode;       // indicates how to behave if the service is killed
    IBinder mBinder;      // interface for clients that bind
    boolean mAllowRebind; // indicates whether onRebind should be used

    @Override
    public void onCreate() {
        // The service is being created
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // The service is starting, due to a call to startService()
        return mStartMode;
    }
    @Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with bindService()
        return mBinder;
    }
    @Override
    public boolean onUnbind(Intent intent) {
        // All clients have unbound with unbindService()
        return mAllowRebind;
    }
    @Override
    public void onRebind(Intent intent) {
        // A client is binding to the service with bindService(),
        // after onUnbind() has already been called
    }
    @Override
    public void onDestroy() {
        // The service is no longer used and is being destroyed
    }
```

The Service Lifecycle (2)



Two types of service

- Default Service
  - Does not handle threads, must be done manually
- Intent Service
  - Handles requests one by one

```
public class HelloIntentService extends IntentService {

  /**
   * A constructor is required, and must call the super IntentService(String)
   * constructor with a name for the worker thread.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
   * The IntentService calls this method from the default worker thread with
   * the intent that started the service. When this method returns, IntentService
   * stops the service, as appropriate.
   */
  @Override
  protected void onHandleIntent(Intent intent) {
      // Normally we would do some work here, like download a file.
      // For our sample, we just sleep for 5 seconds.

  }
}
```

More on services

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

- Asynchronous - When the service is finished, call some global variable
- How about remote calls - or long running service? ?
  - To be used if you require that the service is accessed by third party apps
  - Provide a messaging interface

Bound service

```
public class MessengerService extends Service {


    /**
     * Handler of incoming messages from clients.
     */
    class IncomingHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_REGISTER_CLIENT:
                    mClients.add(msg.replyTo);
                    break;
                case MSG_UNREGISTER_CLIENT:
                    mClients.remove(msg.replyTo);
                    break;
                case MSG_SET_VALUE:
                    // dome something
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }

    final Messenger mMessenger = new Messenger(new IncomingHandler());

    ....
    @Override
    public IBinder onBind(Intent intent) {
        return mMessenger.getBinder();
    }
```

How to communicate with a remote service

```
<service android:name=".app.MessengerService"
        android:process=":remote" />


// within an Activity
private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
            IBinder service) {

        mService = new Messenger(service);

        try {
            Message msg = Message.obtain(null,
                    MessengerService.MSG_REGISTER_CLIENT);
            msg.replyTo = mMessenger;
            mService.send(msg);

            // Give it some value as an example.
            msg = Message.obtain(null,
                    MessengerService.MSG_SET_VALUE, this.hashCode(), 0);
            mService.send(msg);
        } catch (RemoteException e) {
            // In this case the service has crashed before we could even
            // do anything with it; we can count on soon being
            // disconnected (and then reconnected if it can be restarted)
            // so there is no need to do anything here.
        }

        // As part of the sample, tell the user what happened.
        Toast.makeText(Binding.this, R.string.remote_service_connected,
                Toast.LENGTH_SHORT).show();
    }
```

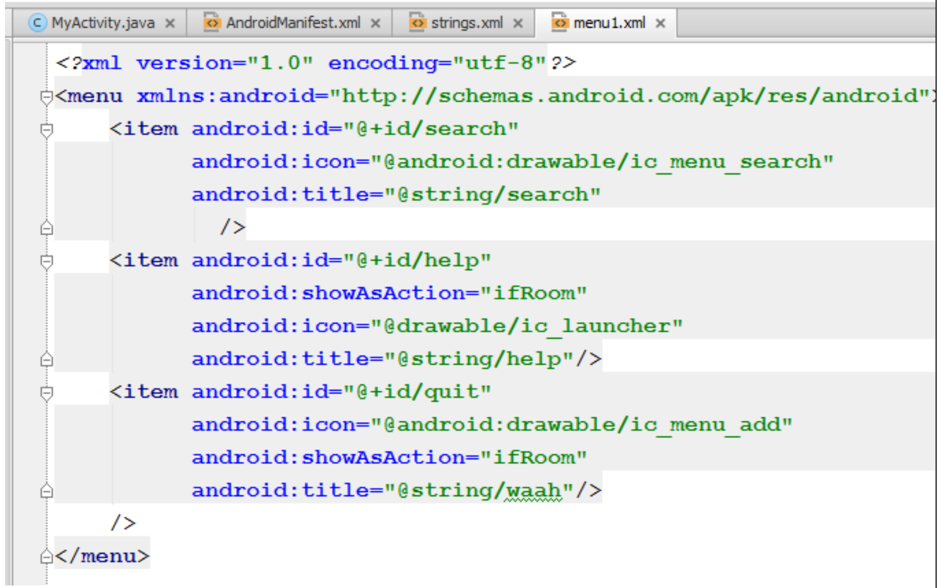
Types of Menu

- Options Menu
  - Will pop up when the menu “button” is pressed on an app
  - The location of the “button” will depend on the device: on modern nexus devices it appears as a column of dots in the ActionBar
- Popup Menu
  - Appears when an item within a view is clicked, where the item handles the relevant event
- Context Menu
  - Appears on items that handle a long-click event
  - Menus can be declared in XML or in Java

Menus: all about selection

- A menu presents one or more items for a user to select
- When the item is selected an action should be taken
- Menus are added to parent views
  - Write a method to handle the appropriate event
  - It is common for the same method to handle many menu item selections
  - Then use a switch statement to detect which item was selected

Creating a Menu in XML



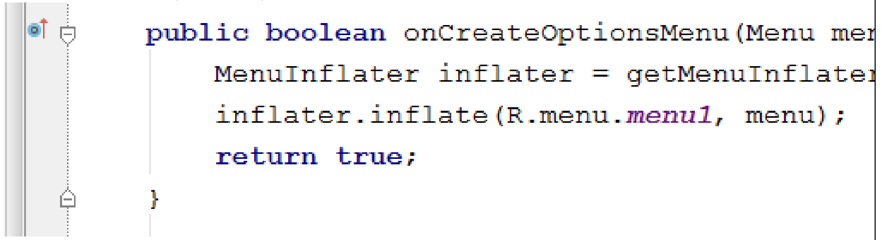
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    <item android:id="@+id/search"
          android:icon="@android:drawable/ic_menu_search"
          android:title="@string/search"
            />
    <item android:id="@+id/help"
          android:showAsAction="ifRoom"
          android:icon="@drawable/ic_launcher"
          android:title="@string/help"/>
    <item android:id="@+id/quit"
          android:icon="@android:drawable/ic_menu_add"
          android:showAsAction="ifRoom"
          android:title="@string/waah"/>
    />
</menu>
```

Questions

android:id=“@id/help” vs android:id=“@+id/help”

Then loading it in Java

- Override the onCreateOptionsMenu method
- Use a MenuInflater to build the menu
- Note: menu1 matches the name of the xml file (menu1.xml) in the folder res/menu/



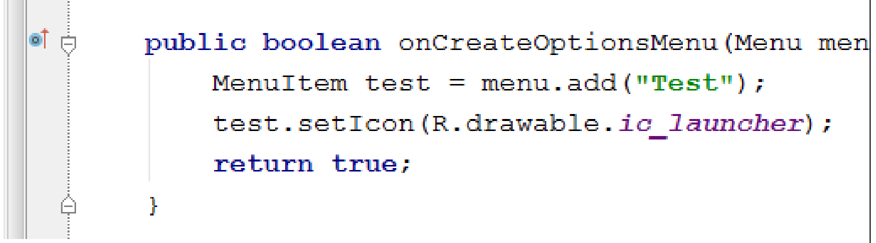
```
public boolean onCreateOptionsMenu(Menu mer
    MenuInflater inflater = getMenuInflater
    inflater.inflate(R.menu.menu1, menu);
    return true;
}
```

Creating a Menu in Java

- Override the **onCreateOptionsMenu** method
- Add the menu item and assign the return value to a reference variable of type MenuItem
- Call methods of the MenuItem object to modify its appearance or where it appears

Java Code

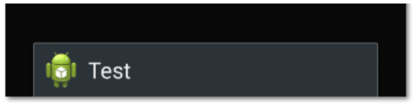
- In addition to adding a menu item labelled “Test” we also add an Icon to it

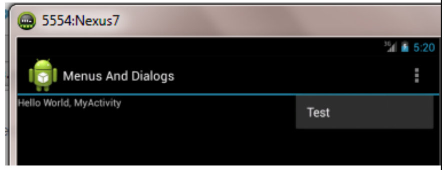


```
public boolean onCreateOptionsMenu(Menu men
    MenuItem test = menu.add("Test");
    test.setIcon(R.drawable.ic_launcher);
    return true;
}
```

Frustrating Differences

- The exact appearance of a menu differs with version of Android OS (or variations in UI added by manufacturer)
- E.g. the above Java code running on S4 (above) versus on a Nexus 7 emulator (below)





Adding Custom Menu Icons

- Normal practice is to either:
  - Use Android Platform Icons
  - Add your own in the drawable folder
    - Ideally there should be separate versions for each resolution
    - The IDE may do this for you
- Somehow you need to do it automatically for your sanity
- But it's also possible to draw your own at Runtime. . .

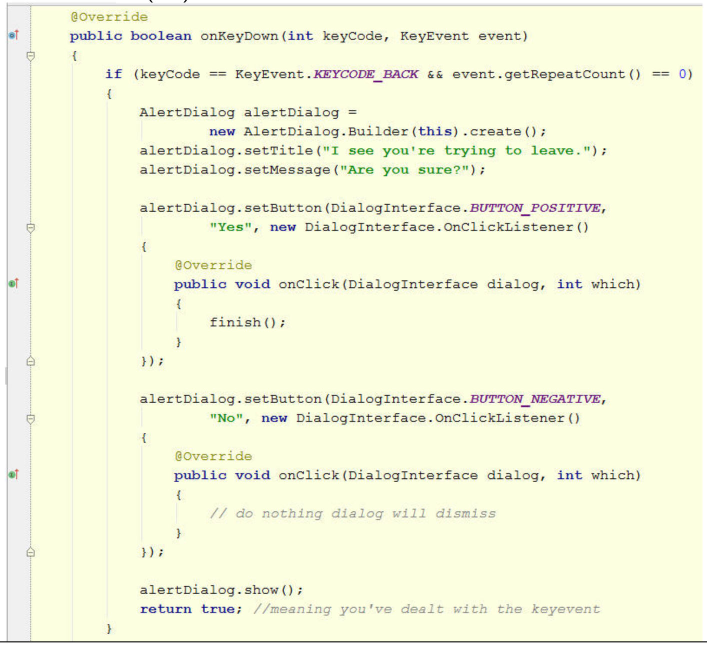
Dynamic Menu Icon Creation

- When setting up the menu icon:
- setIcon() can take a Drawable (Drawable is an Abstract Class)
- So do this:
  - **class MyIcon extends Drawable**
  - Then implement the draw(Canvas c) method
  - Doing this felt a bit "off" - it might be useful - and was an interesting exercise, but use with some caution

Dialogs (1)

- Dialogs can be built very easily using the AlertDialog builder.
- The following code assumes this is being called from a method of an Activity
  - (note the "this" object being passed to the AlertDialog.Builder(this) constructor)
- The rest of the code:
- Sets the title and message strings
- Sets handlers for the onClick events for each button
- Shows the Dialog

Dialogs (2)



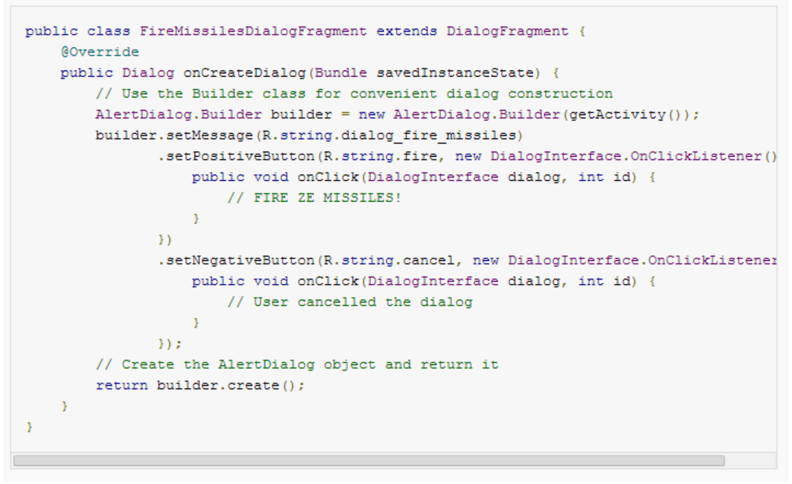
```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event)
{
    if (keyCode == KeyEvent.KEYCODE_BACK && event.getRepeatCount() == 0)
    {
        AlertDialog alertDialog =
                new AlertDialog.Builder(this).create();
        alertDialog.setTitle("I see you're trying to leave.");
        alertDialog.setMessage("Are you sure?");

        alertDialog.setButton(DialogInterface.BUTTON_POSITIVE,
                "Yes", new DialogInterface.OnClickListener()
        {
            @Override
            public void onClick(DialogInterface dialog, int which)
            {
                finish();
            }
        });

        alertDialog.setButton(DialogInterface.BUTTON_NEGATIVE,
                "No", new DialogInterface.OnClickListener()
        {
            @Override
            public void onClick(DialogInterface dialog, int which)
            {
                // do nothing dialog will dismiss
            }
        });

        alertDialog.show();
        return true; //meaning you've dealt with the keyevent
    }
```

Custom Dialogs

- Main idea:
  - Your custom Dialog class will extend DialogFragment
  - Use the AlertDialog.Builder as before
  - Override the *onCreateDialog* method within the subclass
  - Then create a new instance of your class and call its show method to show it

Example

see: http://developer.android.com/guide/topics/ui/dialogs.html



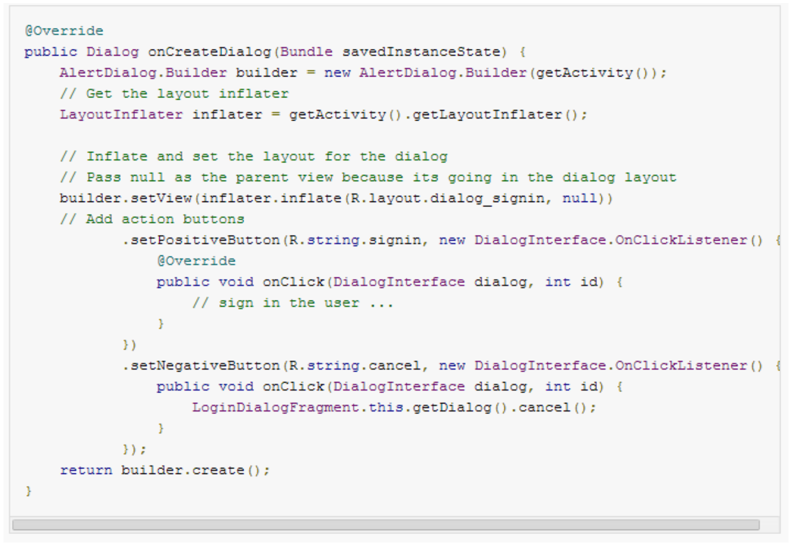
```
public class FireMissilesDialogFragment extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the Builder class for convenient dialog construction
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setMessage(R.string.dialog_fire_missiles)
               .setPositiveButton(R.string.fire, new DialogInterface.OnClickListener()
                   public void onClick(DialogInterface dialog, int id) {
                       // FIRE ZE MISSILES!
                   }
               })
               .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener
                   public void onClick(DialogInterface dialog, int id) {
                       // User cancelled the dialog
                   }
               });
        // Create the AlertDialog object and return it
        return builder.create();
    }
}
```

Specifying a Custom Layout

- Within the builder we can call setView to set a custom view
- Can use Views specified in XML or created dynamically in Java
- This is equivalent to the setContentView we've used in the onCreate method of an Activity
- The relevant line on the next slide (copied from the Android developer guide) is below
- Note that the code looks more complex than necessary due to method call chaining

```
public void onCreate(Bundle savedInstanceState)
{
    builder.setView(inflater.inflate(
    R.layout.dialog_signin, null)
}
```

Code



```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
    // Get the layout inflater
    LayoutInflater inflater = getActivity().getLayoutInflater();

    // Inflate and set the layout for the dialog
    // Pass null as the parent view because its going in the dialog layout
    builder.setView(inflater.inflate(R.layout.dialog_signin, null))
    // Add action buttons
           .setPositiveButton(R.string.signin, new DialogInterface.OnClickListener() {
               @Override
               public void onClick(DialogInterface dialog, int id) {
                   // sign in the user ...
               }
           })
           .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
               public void onClick(DialogInterface dialog, int id) {
                   LoginDialogFragment.this.getDialog().cancel();
               }
           });
    return builder.create();
}
```

Fragments

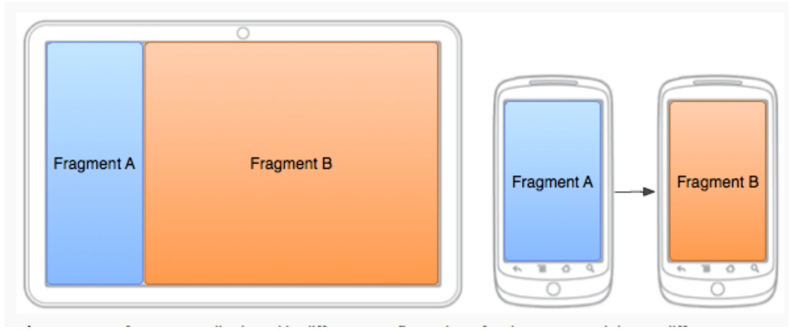
- Fragments offer a powerful way to compose Apps in a highly modular way
- Choice of Layout can easily depend on screen size
- Each Fragment has it's own lifecycle, tied to its parent activity's cycle
  - Composing an Activity from several fragments offers flexibility
  - The overhead is additional coding
  - These notes summarise the main points:
- See examples and lab exercise for more detail

Fragments and Android API Versions

- Fragments have native support from API 11 (Honeycomb, 3.0) onwards
  - If compatibility with earlier versions is required then the Android Support Library must be used
  - This also leads to some differences in the code
  - Fragments always belong to an Activity
- i.e. an Activity hosts a Fragment
  - From API 11 onwards, any Activity can host a Fragment
  - With earlier APIs and the Support Library, a FragmentActivity is needed (or a sub-class of this)

Example

http://developer.android.com/training/basics/fragments/fragment-ui.html

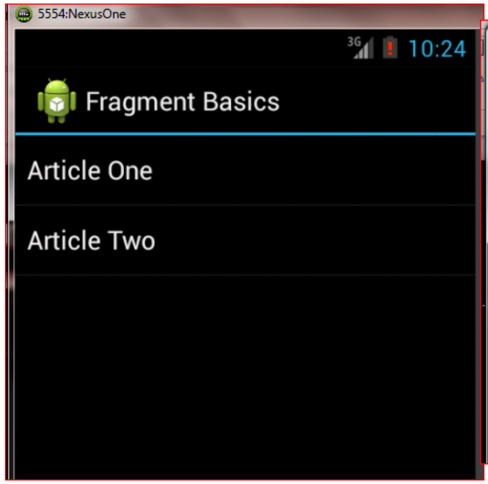


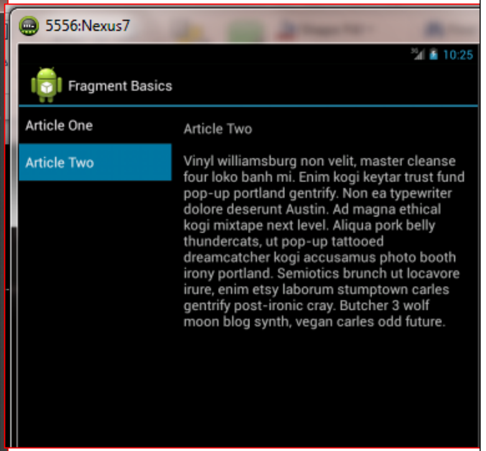
Defining Fragment GUIs

- Like other layouts, can be done in XML or in Java code
- XML trick:
  - Define two layouts, one for small screen, one for tablet (large screen)
  - The one for the tablet must be in a directory with the "large" qualifier e.g. two layouts could be:
    - res/layout/news_articles.xml
    - res/layout-large/news_articles.xml
  - When the layout is inflated the correct one will be chosen
- Following shows FragmentBasics example on Nexus One and Nexus 7 emulators

FragmentBasics Example

Note the different layouts





Vinyl williamsburg non velit, master cleanse four loko banh mi. Enim kogi keytar trust fund pop-up portland gentrify. Non ea typewriter dolore deserunt Austin. Ad magna ethical kogi mixtape next level. Aliqua pork belly thundercats, ut pop-up tattooed dreamcatcher kogi accusamus photo booth irony portland. Semiotics brunch ut locavore irure, enim etsy laborum stumptown carles gentrify post-ironic cray. Butcher 3 wolf moon blog synth, vegan carles odd future.

Fragments in Java Code

- When Fragments are added to XML layouts they cannot be removed in code
- Alternative: Fragments can be added or removed with the appropriate Java
  - This enables dynamic construction of GUIs
  - BUT NOTE: all additions and removals of Fragments MUST be done within a Fragment Transaction
  - Discussion: why is this so?

Implementing Fragments

- Similar to defining an Activity
- Except main override for Activity is onCreate()
- For Fragment use:

```
onCreateView()
   // check the savedInstanceState
   // inflate the layout
 onStart()
   // can now update view components
   // since layout is ready for use
   // perform any initialisation
   // and restoration of state
```

Managing Fragments

- The host activity is responsible for:
- Creating, adding and removing fragments
- Note:
  - To be visible each Fragment must be added to a View
  - Within the Fragment code, Save any state by overriding onPause() or onSavedInstanceState()

Programming with Fragments

- Note: communication between sibling fragments is not allowed
- Instead communicate via parent activity
- Also, see examples here:
  - http://developer.android.com/training/basics/fragments/fragment-ui.html
  - http://developer.android.com/training/animation/cardflip.html
- Discussion question: there is nothing to stop you trying this, but why do you think it is “not allowed”
- Do you need more than one activities in your app? Why not just stick to fragments?

XML Fragments - loading

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.example.news.ArticleListFragment"
            android:id="@+id/list"
            android:layout_weight="1"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
    <fragment android:name="com.example.news.ArticleReaderFragment"
            android:id="@+id/viewer"
            android:layout_weight="2"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
</LinearLayout>
```

XML Fragments - loading

```
public static class ExampleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.example_fragment, container, false);
    }
}
```

Programmatically

```
FragmentManager fragmentManager = getFragmentManager()
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

Fragment Transactions

- Why transactions?

Summary

- Services!
- With these Menus and Dialogs you can build sophisticated custom-designed User Interfaces for your apps
- Practice these ideas in the lab
- Use fragments
- Fragments are recommended for building apps in a scalable and flexible way
- Especially good for coping with different screen sizes
- They are reusable modules that always belong to a parent (host) Activity
- But are responsible for managing some lifecycle callbacks to initialise, save, and restore their state
- Some slides based on Simon's course