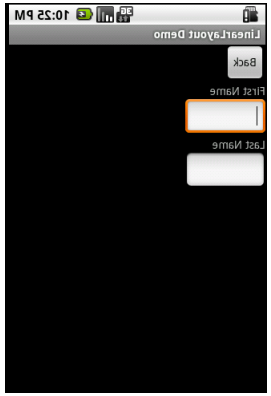
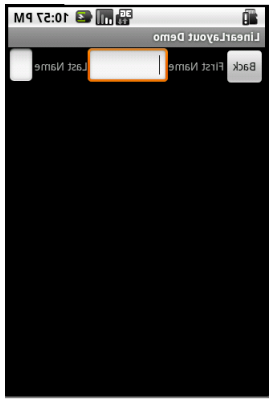
UI: Understanding Layouts & Fragments





Mobile Applications
Jay Urbain, Ph.D.

Credits:

- http://developer.android.com/index.html
- Lars Vogel, http://www.vogella.com/articles/Android/article.html#fragments_tutorial
- Sheridan Saint-Michel
- Meier, Reto, Professional Android 4 Application Development.

Objectives

- Using Views and layouts
- Optimizing layouts
- Creating resolution-independent user interfaces
- Extending, grouping, creating, and using Views
- Fragments
- Using Fragments for portrait and landscape mode

Android UI

- The individual elements of an Android UI are arranged on screen by means of a variety of *Layout Managers* derived from the *ViewGroup* class.
- Android provides several common UI controls, widgets, and Layout Managers.
- Modify standard Views— or create composite or entirely new Views— to provide your own user experience.

Android UI

- **View—**
  - Base class for all visual interface elements.
  - All UI controls, including the layout classes, are derived from View.
- **View Group—**
  - Extends View class, can contain *multiple* child Views.
  - Extend the ViewGroup class to create compound controls made up of child Views.
  - Extend to create *Layout Managers*.
- **Fragment—**
  - Used to encapsulate portions of your UI. Introduced in Android 3.0 (API level 11).
  - Useful for layouts with different screen sizes, portrait/landscape views, and creating reusable UI elements.
  - Each Fragment includes its own UI layout and receives the related input events, but is tightly bound to the Activity in which it is embedded.
  - *Note: Similar to UI View Controllers in iPhone development.*
- **Activity—**
  - Represent the window, or screen, being displayed.
  - Equivalent of Forms in traditional Windows desktop development.
  - To display a UI, you assign a View (usually a layout or Fragment) to an Activity.

UI Fundamentals

- A new Activity starts with an empty screen onto which you place your UI.
- *setContentView()* method accepts either a layout's resource ID or a single View instance.

```
@Override
public void onCreate( Bundle savedInstanceState) {
super.onCreate( savedInstanceState );
   setContentView( R.layout.main); // accepts single View instance
}
```

UI Fundamentals

- Using layout resources decouples your ***presentation layer*** from the ***application logic***.

  You can obtain a reference to each of the Views within a layout using the *findViewById()* method:

  TextView myTextView = (TextView) findViewById(R.id.myTextView);

- **Can also use programmatic approach**:

```
@Override public void onCreate( Bundle savedInstanceState) {
   super.onCreate( savedInstanceState);
   TextView myTextView = new TextView( this);
   setContentView( myTextView);  // accepts single View instance
   myTextView.setText(" Hello, Android");
}
```

Fragments

- If you're using **Fragments** to encapsulate portions of your Activity's UI, the View inflated within your Activity's *onCreate()* handler will be a layout that describes the relative position of each of your Fragments (or their containers).
- The UI used for each Fragment is defined in its own layout and inflated within the Fragment itself.
- Once a Fragment has been inflated into an Activity, the Views it contains become part of that Activity's View hierarchy.

Layouts (Layout Mangers)

- Layouts are extensions of the ViewGroup class.
- Used to position child Views within your UI.

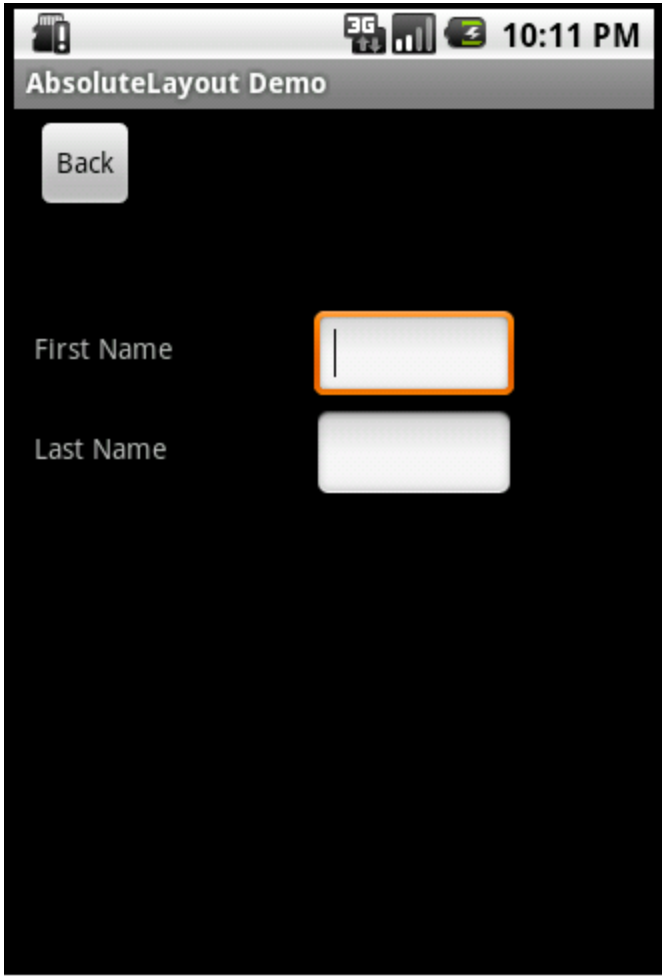
- AbsoluteLayout
- FrameLayout
- LinearLayout
- RelativeLayout
- TableLayout

Demo Layouts!

Absolute Layout

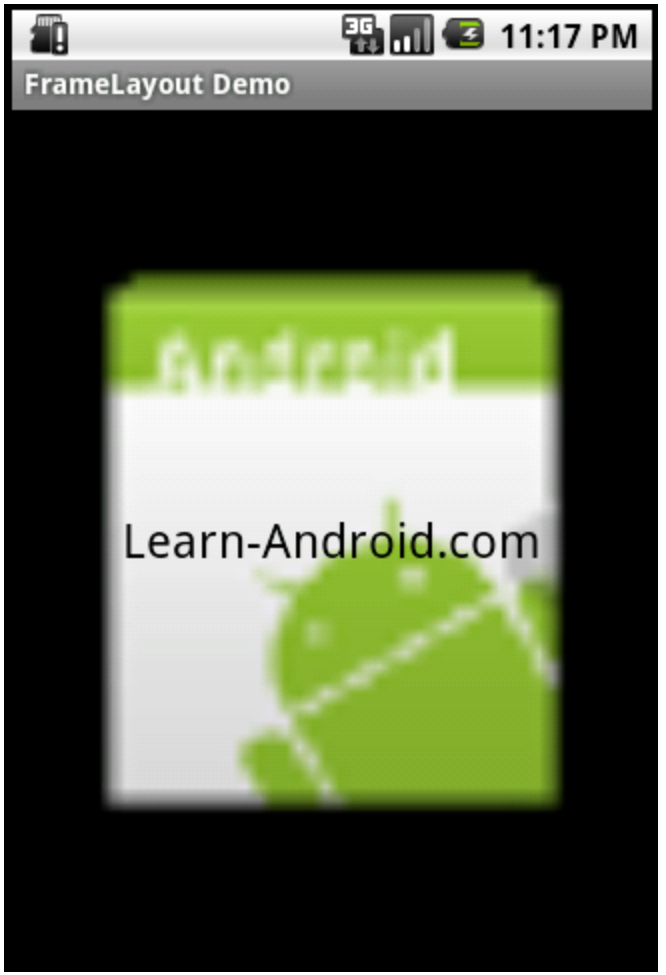
- Place each control at an absolute position.
- Specify exact x and y coordinates on the screen for each control.
- Deprecated, since it makes an inflexible UI that is much more difficult to maintain.

```
<AbsoluteLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:id="@+id/backbutton"
        android:text="Back"
        android:layout_x="10px"
        android:layout_y="5px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView
        android:layout_x="10px"
        android:layout_y="110px"
        android:text="First Name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```



Frame Layout

- Designed to display a single item at a time.
- Each element is positioned based on the top left of the screen.
- Elements that overlap will be displayed overlapping.
- Assign gravity.

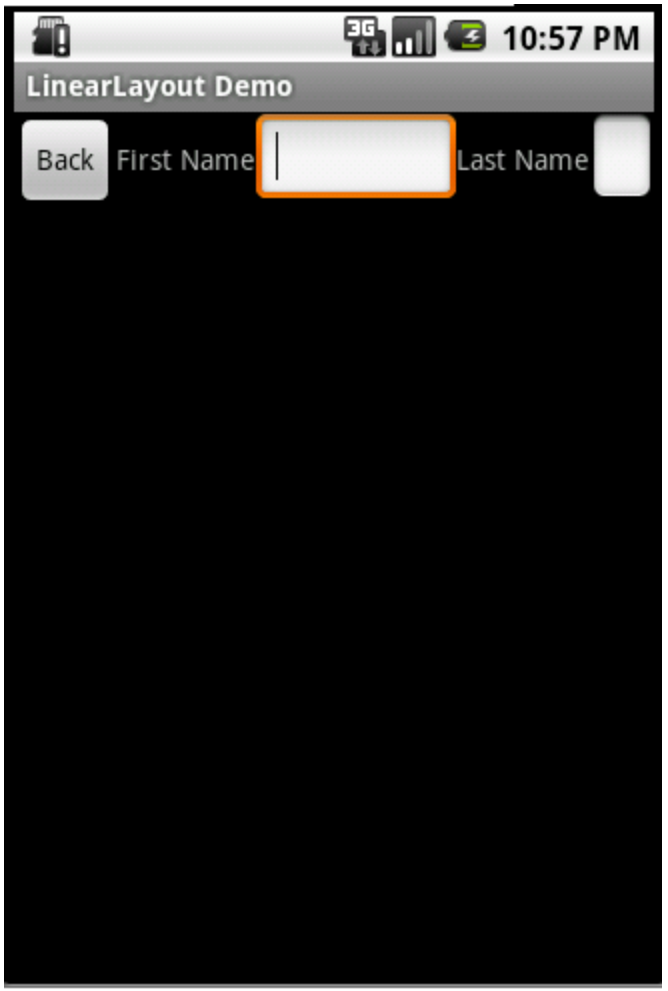



```
<FrameLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android">
        <ImageView
                android:src="@drawable/icon"
                android:scaleType="fitCenter"
                android:layout_height="fill_parent"
                android:layout_width="fill_parent"/>
        <TextView
                android:text="Learn-Android.com"
                android:textSize="24sp"
                android:textColor="#000000"
                android:layout_height="fill_parent"
                android:layout_width="fill_parent"
                android:gravity="center"/>
</FrameLayout>
```

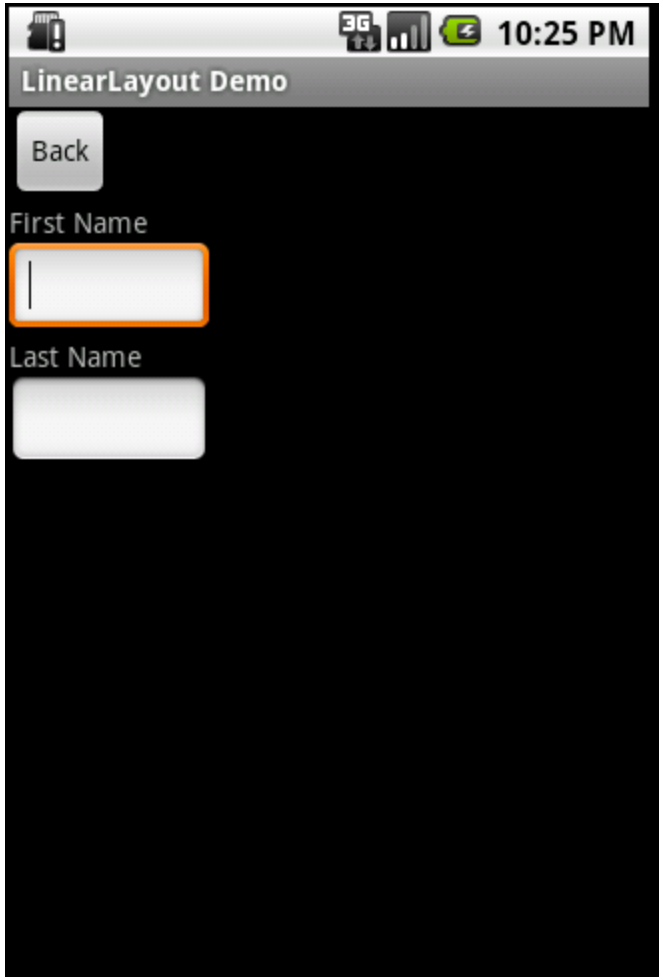
LinearLayout

- Organizes elements along a single line.
- Specify whether that line is *vertical* or *horizontal* using **android:orientation**.
- E.g., horizontal:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="fill_parent"
    android:layout_height="fill_parent">
     <Button
        android:id="@+id/backbutton"
        android:text="Back"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView
        android:text="First Name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <EditText
        android:width="100px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView
        android:text="Last Name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <EditText
        android:width="100px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```



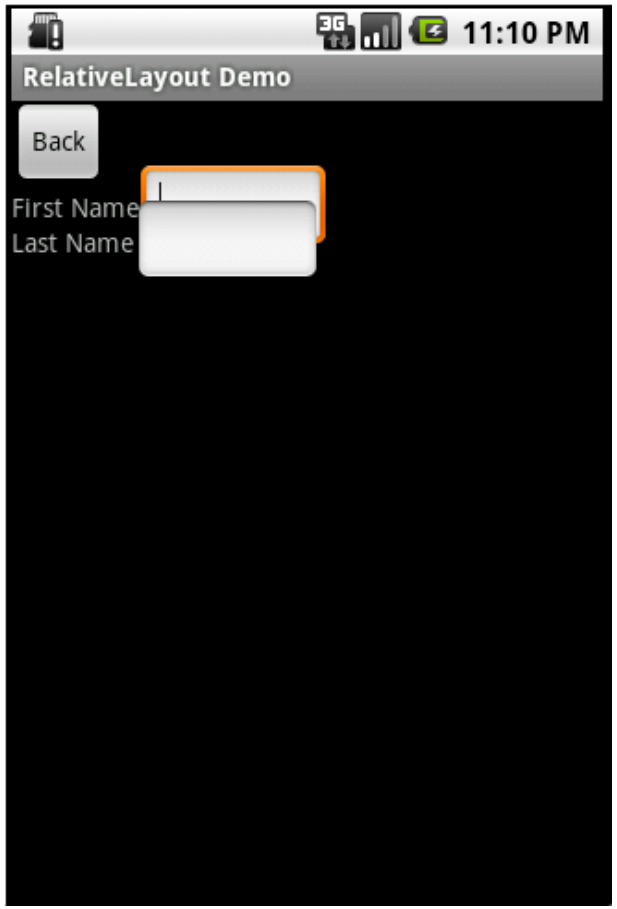
Vertical vs. Horizontal LinerarLayouts



RelativeLayout

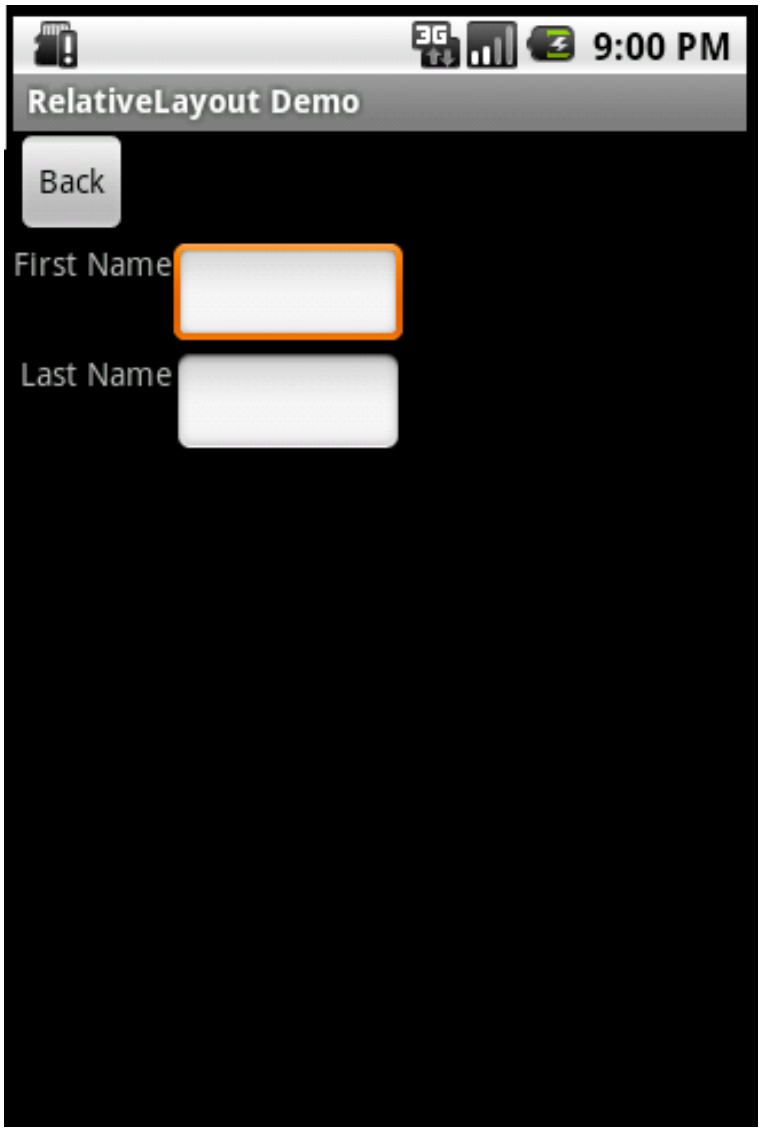
- Lays out elements relative to one another, and with the parent container.
- Arguably one of the most complicated, but also one of the most useful layouts.
- Need several properties to actually get the layout we want.

```
<RelativeLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android">
        <Button
                android:id="@+id/backbutton"
                android:text="Back"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
        <TextView
                android:id="@+id/firstName"
                android:text="First Name"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_below="@id/backbutton" />
        <EditText
                android:width="100px"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_toRightOf="@id/firstName"
                android:layout_alignBaseline="@id/firstName" />
        <TextView
                android:id="@+id/lastName"
                android:text="Last Name"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_below="@id/firstName" />
        <EditText
                android:width="100px"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_toRightOf="@id/lastName"
                android:layout_alignBaseline="@id/lastName" />
</RelativeLayout>
```



RelativeLayout

```
<RelativeLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android">
        <Button
                android:id="@+id/backbutton"
                android:text="Back"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
        <TextView
                android:id="@+id/firstName"
                android:text="First Name"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_below="@id/backbutton" />
        <EditText
                android:id="@+id/editFirstName"
                android:width="100px"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_toRightOf="@id/firstName"
                android:layout_below="@id/backbutton"/>
        <EditText
                android:id="@+id/editLastName"
                android:width="100px"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_below="@id/editFirstName"
                android:layout_alignLeft="@id/editFirstName"/>
        <TextView
                android:id="@+id/lastName"
                android:text="Last Name"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_toLeftOf="@id/editLastName"
                android:layout_below="@id/editFirstName" />
</RelativeLayout>
```

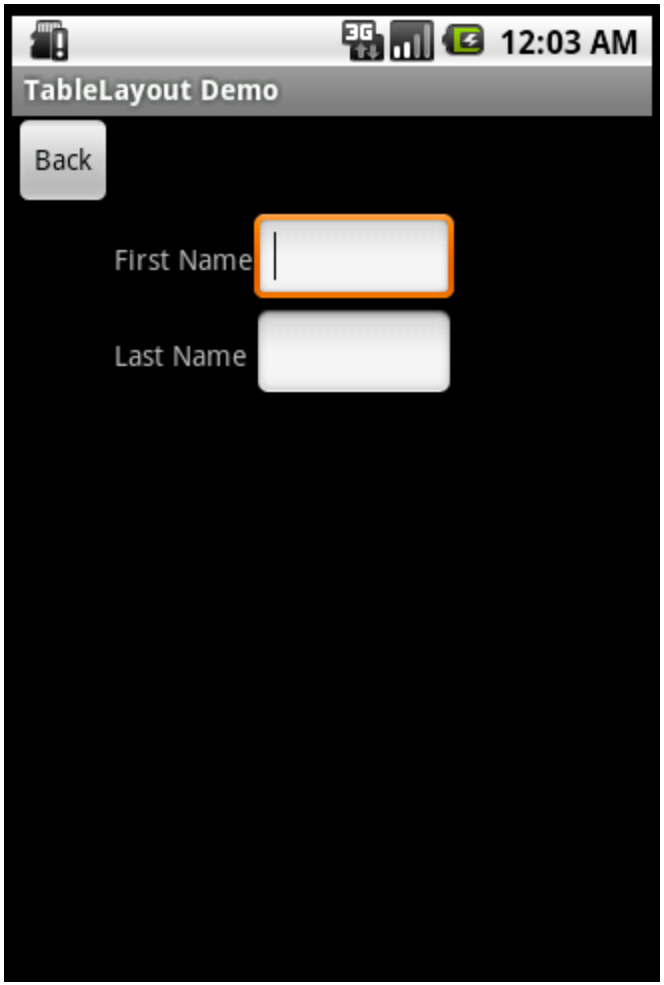


TableLayout

- Organizes content into rows and columns.
- Rows are defined in the layout (XML), and the columns are determined automatically by Android.
- Example: A row with two elements and a row with five elements would have a layout with 2 rows and 5 columns.
- Specify an element should occupy more than one column:

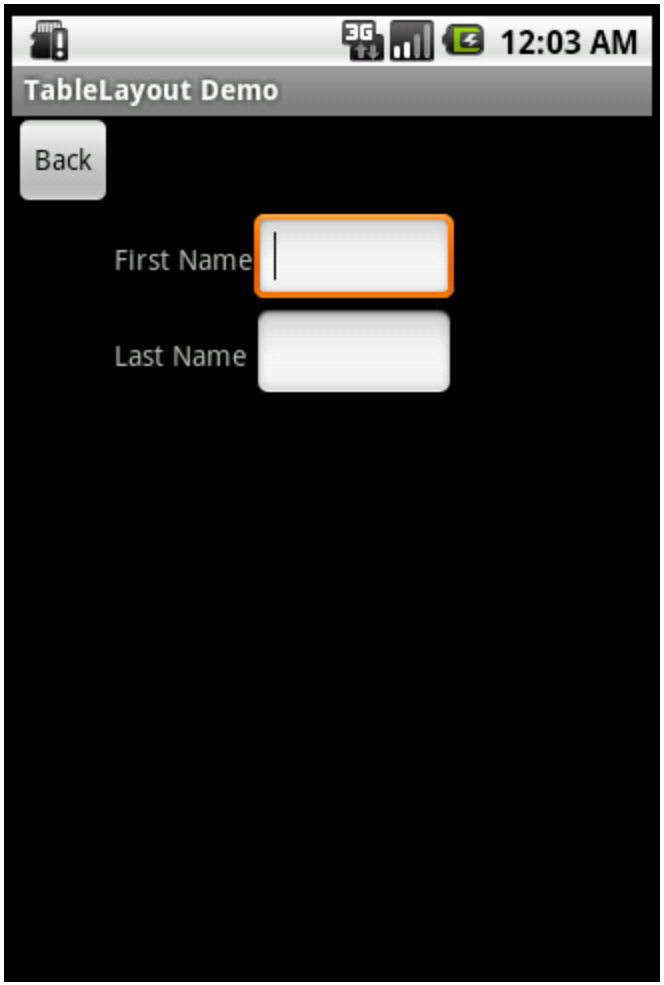
  android:layout_span =”3”.
- By default, each element is placed in the first unused column in the row.
- Specify the column an element should occupy:

  android:layout_column=“1”.



TableLayout

```
<TableLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android">
        <TableRow>
                <Button
                android:id="@+id/backbutton"
                android:text="Back"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
        </TableRow>
        <TableRow>
                <TextView
                android:text="First Name"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_column="1" />
                <EditText
                android:width="100px"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
        </TableRow>
        <TableRow>
                <TextView
                android:text="Last Name"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_column="1" />
                <EditText
                android:width="100px"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
        </TableRow>
</TableLayout>
```



Alternate Layouts

- When using LinearLayout, Android will shrink elements when they don't all fit on the screen.
- You can use alternate layouts for different screen orientations.

Example: Use alternate layouts for portrait and landscape.

- res/layout-land – The alternate layout for a landscape UI.
- res/layout-port – The alternate layout for a portrait UI
- res/layout-square – The alternate layout for a square UI.

Alternate Layouts

- Create a folder named *layout-land* under the *res* folder and place your XML under the new folder. The XML file should have the same name it has in the layout folder, in this case *linear_layout.xml*.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="fill_parent"
    android:layout_height="fill_parent">
     <Button
        android:id="@+id/backbutton"
        android:text="Back"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView
        android:text="First Name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <EditText
        android:width="100px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView
        android:text="Last Name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <EditText
        android:width="100px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Fragments

- Components that run in the context of an Activity.
- Fragments encapsulate functionality so that its "*easier*" to reuse, and easier to support different sized devices.
- Fragments have their own lifecycle and their own user interface.
- Can be defined via layout files or via coding.
- If an Activity stops, its Fragments will also be stopped; if an Activity is destroyed its Fragments will be destroyed.

Advantages of Using Fragments

- Fragments make it easy to reuse components in different layouts.
- *E.g. you can build single-pane layouts for handsets (phones) and multi-pane layouts for tablets.*

Creating different layouts with Fragments

Two approaches for creating different layouts with *Fragments*:

1 - Use one activity which displays two *Fragments* for tablets and only one for handsets devices.

- Switch the *Fragments* in the activity whenever necessary.
- Requires Fragment *not* to be declared in the layout file as such *Fragments* cannot be removed during runtime.
- Requires an update of the action bar if the action bar status depends on the fragment.

OR

2 - Use separate activities to host each fragment on a handset.

- Tablet UI uses two *Fragments* in an activity, use the same activity for handsets, but supply an alternative layout that includes just one fragment.
- When you need to switch *Fragments*, start another activity that hosts the other fragment.
- Considered a more flexible approach.

Defining Fragments

Extend either **android.app.Fragment** class or one of its subclasses:

- **ListFragment**
- **DiaglogFragment**
- **PreferenceFragment**
- **WebViewFragment**

Extending Fragment

```
package com.example.android.rssfeed;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class DetailFragment extends Fragment {

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
      Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_rssitem_detail,
        container, false);
    return view;
  }

  public void setText(String item) {
    TextView view = (TextView) getView().findViewById(R.id.detailsText);
    view.setText(item);
  }
}
```

Adding Fragments statically

- To use a Fragment you can statically add it to an XML layout.

- To check if the *Fragment* is already part of your layout you can use the FragmentManager class.

```
@Override
public void onRssItemSelected(String link) {
  DetailFragment fragment = (DetailFragment) getFragmentManager()
          .findFragmentById(R.id.detailFragment);
      if (fragment != null && fragment.isInLayout()) {
        fragment.setText(link);
      }
}
```

Fragment life cycle

- A Fragment has its own life cycle. But it is always connected to the life cycle of the Activity which uses the fragment.
- *onCreate()* method is called after *the onCreate()* method of the Activity, but before the *onCreateView()* method of the Fragment.
- *onCreateView()* method called when the Fragment should create its **UI**. Inflate a layout via the *inflate()* method of Inflator object.
- *onActivityCreated()* called after the *onCreateView()* method when the host Activity is created. Instantiate objects requiring a Context object.
- *Fragments* don't subclass the Context. Use the *getActivity()* method to get the parent *Activity*.
- The *onStart()* method is called once the fragment gets visible.
- If an *Activity* stops, its *Fragments* are also stopped; if an *Activity* is destroyed its *Fragments* are also destroyed.

Application Communication

- To increase reuse of *Fragments* they should not communicate directly with each other.
- Communication of the *Fragments* should be done via the host *Activity*.
- Fragment should define an interface as an *inner type* and require that the *Activity* implement and use this interface.
- This way, the Fragment does not need any knowledge about the *Activity* that uses it.
- The *onAttach()* method can be used to check if the *Activity* correctly implements this interface.

Checking Activity Interface

- Assume you have a *Fragment* which should communicate a value to its parent *Activity*. Implement as follows:

```
@Override
  public void onAttach(Activity activity) {
    super.onAttach(activity);
    if (activity instanceof OnItemSelectedListener) {
      listener = (OnItemSelectedListener) activity;
    } else {
      throw new ClassCastException(activity.toString()
          + " must implemenet MyListFragment.OnItemSelectedListener");
    }
  }
```

Persisting data in Fragments

- To store application data, you can persist data in:
  - SQlite database
  - File
  - The Application object - application will need to handle the storage.

Modifying Fragments at runtime

- FragmentManager class and the FragmentTransaction class allow you to add, remove and replace fragments in the layout of your *Activity*.

- *Fragments* can be dynamically modified via transactions. To dynamically add *Fragments* to an existing layout you typically define a container in the XML layout file in which you add a *Fragment*.

```
FragmentTransaction ft = getFragmentManager().beginTransaction();
ft.replace(R.id.your_placehodler, new YourFragment());
ft.commit();
```

- A new *Fragment* will replace an existing *Fragment* that was previously added to the container.

Animations for Fragment transition

- During a Fragment transaction you can define transition based on the Property Animation API via the *setCustomAnimations()* method.
- You can also use several standard animations provided by Android via the *setTransition()* method call.
- Defined via the constants starting with: FragmentTransaction.TRANSIT_FRAGMENT_*.
- Both methods allow you to define an entry animation and an exist animation.

Fragments for background processing

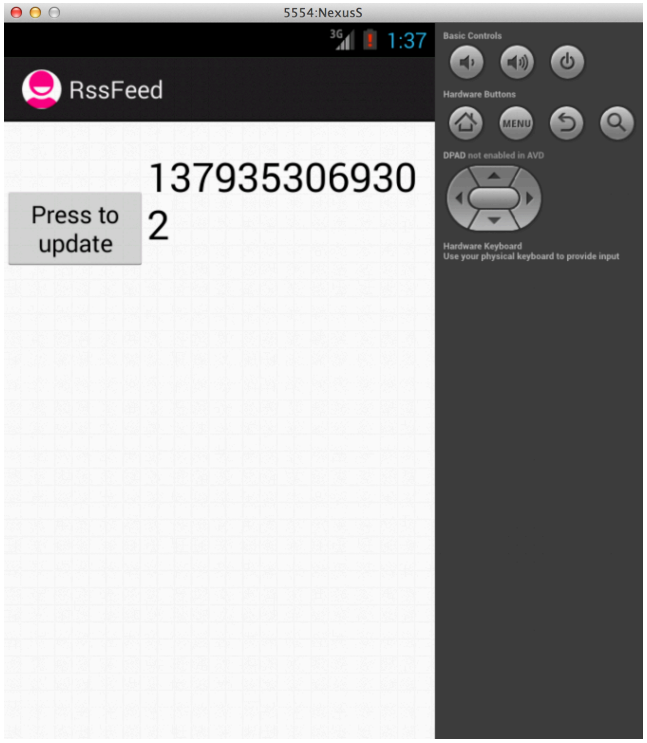
Headless fragments:

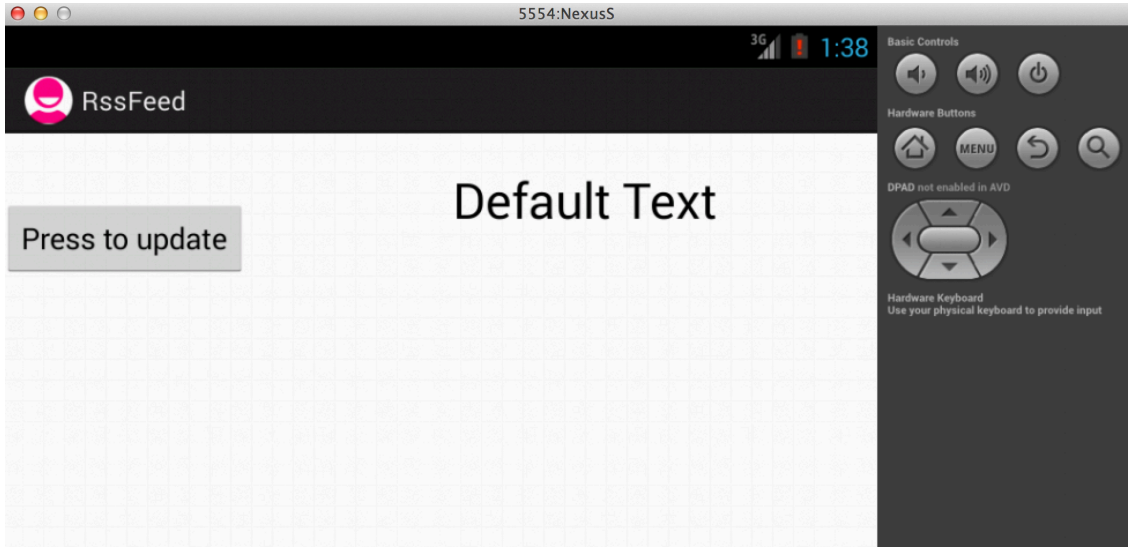
- Fragments can be used without defining a user interface.
- To implement a headless fragment simply return null in the *onCreateView()* method of your fragment.
- Can use headless for background processing for related to an Activity.

Contributing to the ActionBar

- Fragments can also contribute entries to the ActionBar.
- Call *setHasOptionsMenu()* in the *onCreate()* method of the Fragment.
- Android calls the *onCreateOptionsMenu()* method in the Fragment class, and adds its menu items to the ones added by the Activity.

Fragment demo 1 – RssFeed





Fragment demo 1 – RssFeed

Functionality:

- Both fragments displayed in both *landscape* & *portrait* modes.
- If you press the button both the ListFragment and DetailFragment should get updated.

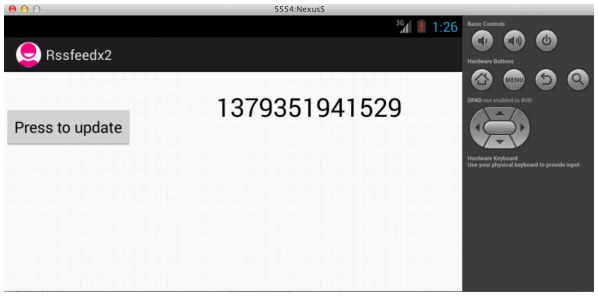
Development steps:

- Create standard layouts: *fragment_rssitem_detail.xml*, *fragment_rsslist_overview.xml*, *activity_rssfeed.xml*.
- Create fragment classes: DetailFragment,java, MyListFragment.java.

Toggling between landscape and portrait modes:

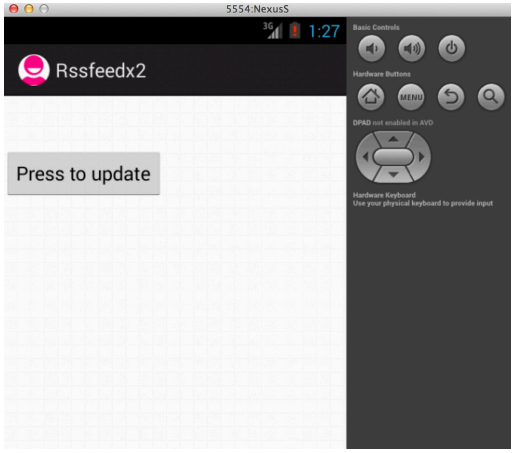
- ctrl+fn+F11 on Mac to change the landscape to portrait and vice versa.
- left-ctrl+F11on Windows 7.
- ctrl+F11 on Linux.

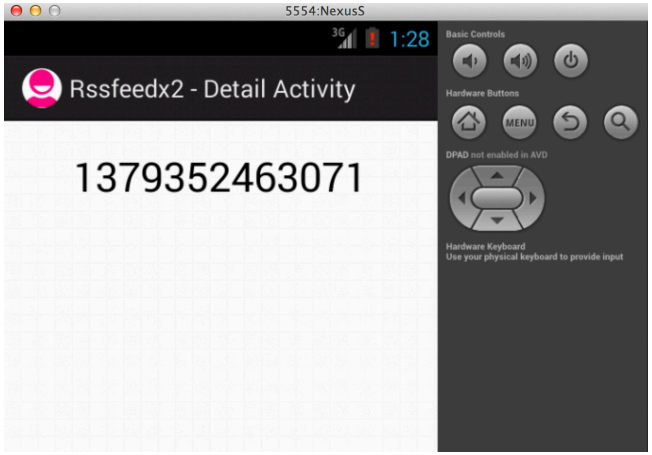
Fragment demo 2 – RssFeedx2



Landscape – display both list and detail fragments

Portrait – display list fragment in RssActivity, invoke DetailActivity to Display detail fragment





Fragment demo 2 – RssFeedx2

Functionality:

- RssFeedActivity should use a special layout file in portrait mode.
- In portrait mode, Android will check the *layout-port* folder for the portrait layout files. Otherwise it uses the *layout* folder.

Development steps:

- Create the *res/layout-port* folder.
- Create a *activity_rssfeed.xml* layout file in the *res/layout-port* folder for DetailActivity. Include only the list fragment within this layout.
- Also create the *activity_detail.xml* layout file for DetailActivity. Include only the detail fragment.
- Create the new DetailActivity – should check for landscape mode and return.
- Adjust RssFeedActivity to display the DetailActivity in case the other Fragment is not present in the layout.

http://stackoverflow.com/questions/4096169/onsaveinstancestate-and-onrestoreinstancestate

7

Usually you restore your state in `onCreate()`. It is possible to restore it in `onRestoreInstanceState()` as well, but not very common. (`onRestoreInstanceState()` is called after `onStart()`, whereas `onCreate()` is called before `onStart()`.

Use the put methods to store values in `onSaveInstanceState()`:

```
protected void onSaveInstanceState(Bundle icicle) {
  super.onSaveInstanceState(icicle);
  icicle.putLong("param", value);
}
```

And restore the values in `onCreate()`:

```
public void onCreate(Bundle icicle) {
  if (icicle != null){
    value = icicle.getLong("param");
  }
}
```

You do not have to store view states, as they are stored automatically.