



Lesson 7

Fragments

Victor Matos

Cleveland State University

Portions of this page are reproduced from work created and [shared by Google](#) and used according to terms described in the [Creative Commons 3.0 Attribution License](#).

Fragments

- Android is a multitasking OS and its hardware specs allow for real parallelism. However, at any given time only one activity per app can be 'visible' and 'active'. This fact is rather limiting considering the extensive screen area offered by larger devices (tablets, phablets, TV sets, etc). Fragments offer an escape solution.
- The **Fragment** class produces visual objects that can be *dynamically attached* to designated portions of the app's GUI. Each fragment object can expose its own views and provide means for the users to interact with the application.
- Fragments **must** exist within the boundaries of an Activity that acts as a 'home-base' or host.
- A host activity's GUI may expose any number of fragments. In this GUI each *fragment* could be visible and active.

Fragments

- Fragments behave as independent threads, usually they cooperate in achieving a common goal; however each can run its own I/O, events and business logic.
- Fragments could access '*global data*' held in the main activity to which they belong. Likewise, they could send values of their own to the main activity for potential dissemination to other fragments.
- Fragments have their own particular Life-Cycle, in which the **onCreateView** method does most of the work needed to make them.
- Fragments were first introduced in the Honeycomb SDK (API 11).

Fragments

ACTIVITY (Main Host Container)

Fragment1
(View 1)

Fragment2
(View 2)

Fragment3
(View 3)

A possible arrangement of Fragments attached to the main GUI of an app.

Fragment's Lifecycle

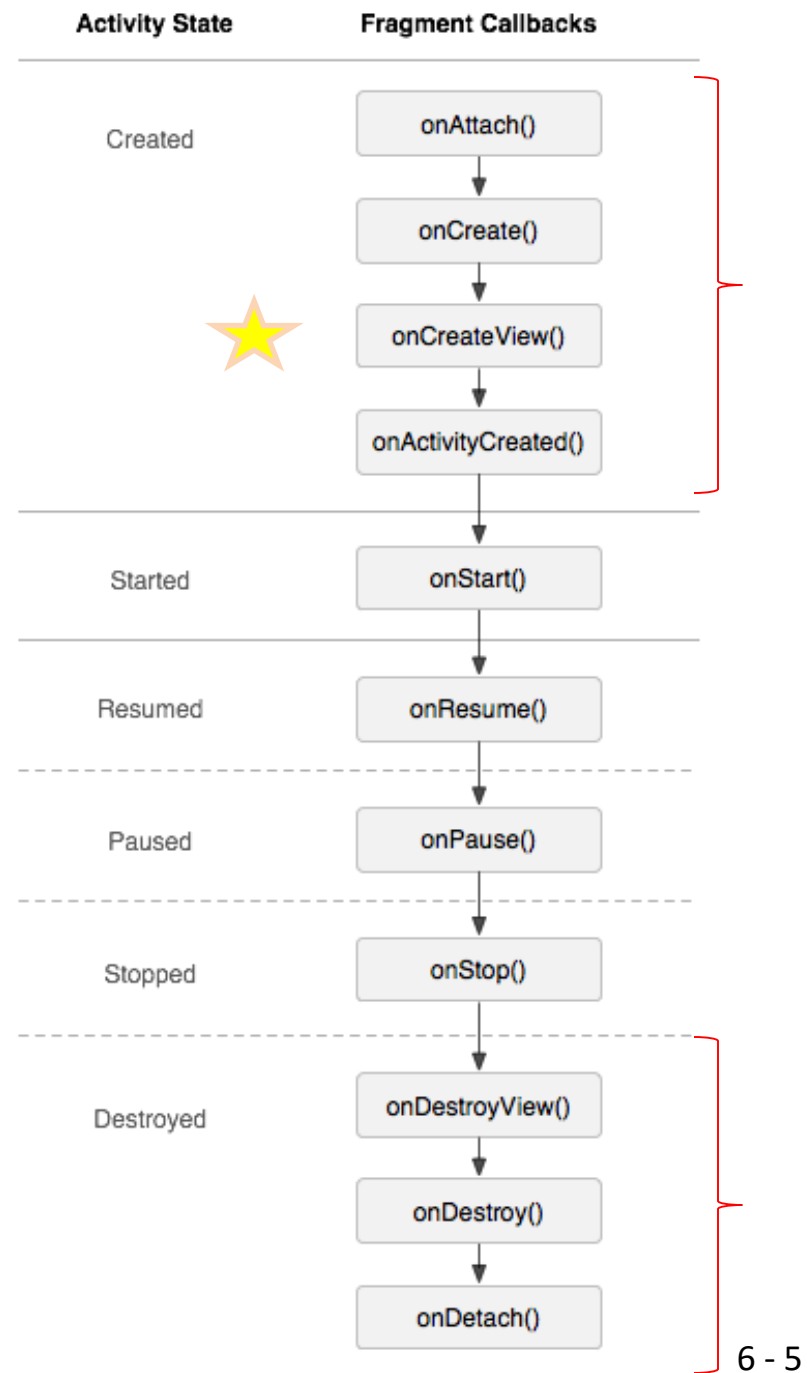
onAttach() Invoked when the fragment has been connected to the host activity.

onCreate() Used for initializing non-visual components needed by the fragment.

onCreateView() *Most of the work is done here.* Called to create the view hierarchy representing the fragment. Usually inflates a layout, defines listeners, and populates the widgets in the inflated layout.

onPause() The session is about to finish. Here you should commit state data changes that are needed in case the fragment is re-executed.

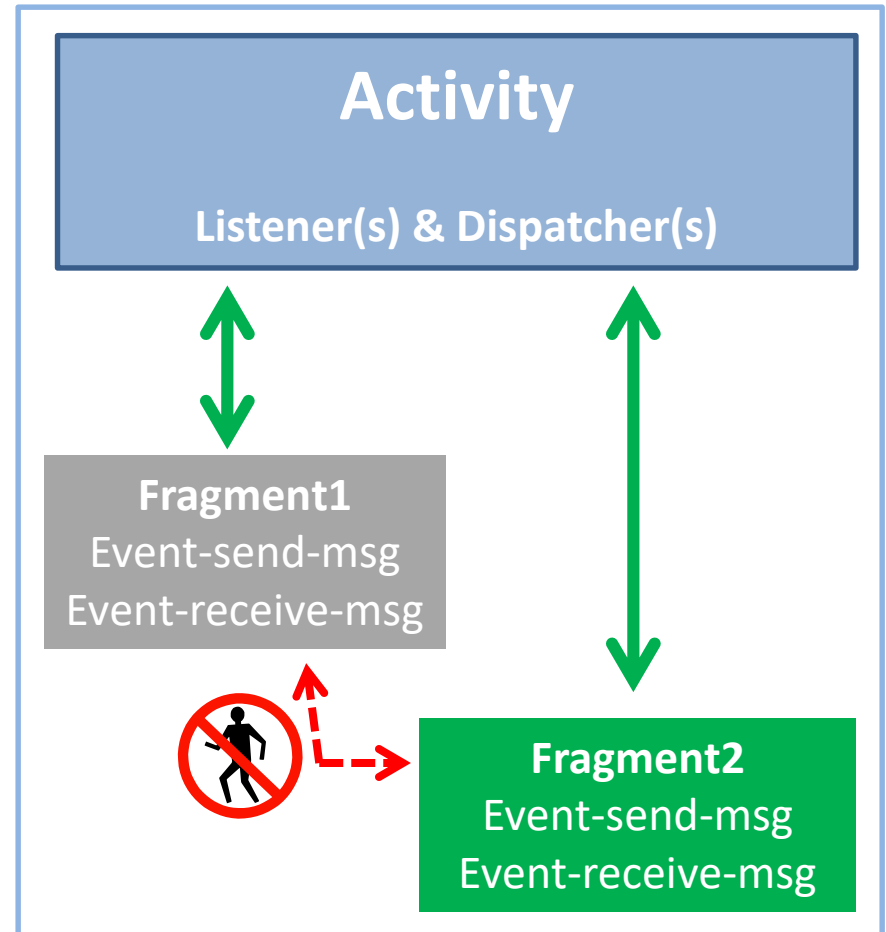
onDetach() Called when the inactive fragment is disconnected from the activity.



Fragments

Inter-Fragment Communication

- All Fragment-to-Fragment communication is done in a centralized mode through the home-base Activity.
- As a design principle; two Fragments should **NEVER** communicate directly.
- The home-base Activity and its fragments interact through listeners and events.
- When a fragment has some data for another fragment, it sends it to a listener in the main which in turn dispatches to a listener of the second fragment.



Fragments

Integrating the Home Activity and its Fragments

In general **fragments** appear on their enclosing Activity's GUI using one of the following attachment approaches

Dynamic Binding

The main activity defines a particular place on its GUI for fragments to be plugged in (or attached). Occupancy of designated areas is not permanent. Later on, the hosting Activity may replace a fragment with another (see **Example-1**)

Static Binding

The Activity's GUI declares a portion of its layout as a `<fragment>` and explicitly supplies a reference to the first type of fragment to be held there using the "android:name=fragmentName" clause. This simple association does not required you to call the constructors (or pass initial parameters). A static binding is permanent, fragments cannot be replaced at run time (see **Example-2**)

Multiple Fragments

The hosting activity may simultaneously expose any number of fragments using a combination of the strategies describe above. Fragments may interact with each other using the enclosing activity as a central *store-and-forward* unit (**Example-3**).

Fragments

Fragment-Activity: Dynamic Binding

- Fragments must be created inside a secure **FragmentTransaction** block.
- You may use the method **add()** to aggregate a fragment to the activity. Optionally any view produced by the fragment is moved into an UI container of the host activity.
- When you use the **replace** method to refresh the UI, the current view in the target area is *removed* and the new fragment is *added* to the activity's UI.
- A faceless fragment may also be *added* to an activity without having to produce a view hierarchy.

STEPS

1. Obtain a reference to the `FragmentManager`, initiate a transaction

```
FragmentTransaction ft= getFragmentManager().beginTransaction();
```

2. Create an instance of your fragment, supply arguments if needed.

```
FragmentBlue blueFragment= FragmentBlue.newInstance("some-value");
```

3. Place the fragment's view on the application's GUI.

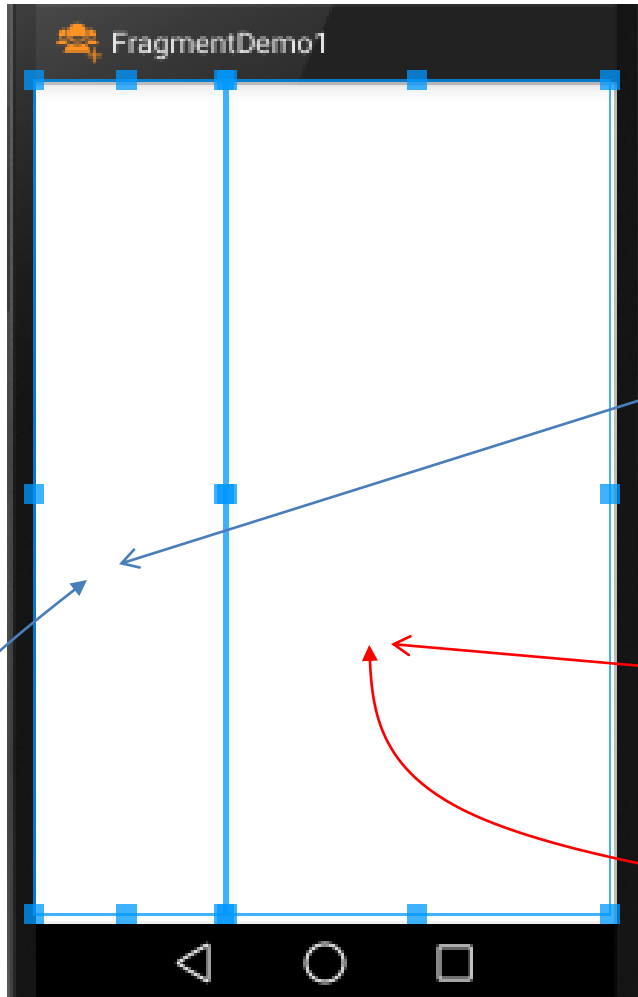
```
ft.replace(R.id.main_holder_blue, blueFragment);
```

4. Terminate the transaction.

```
ft.commit();
```


Fragments

Integrating the Home Activity and its Fragments



Example of dynamic binding. Instances of the **FragmentRed** and **FragmentBlue** classes are created at run-time and set to replace the left and right portions of the app's GUI.

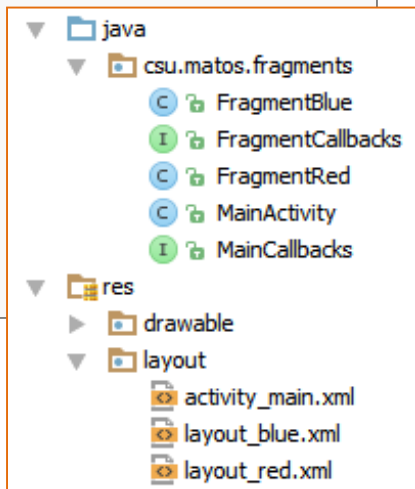
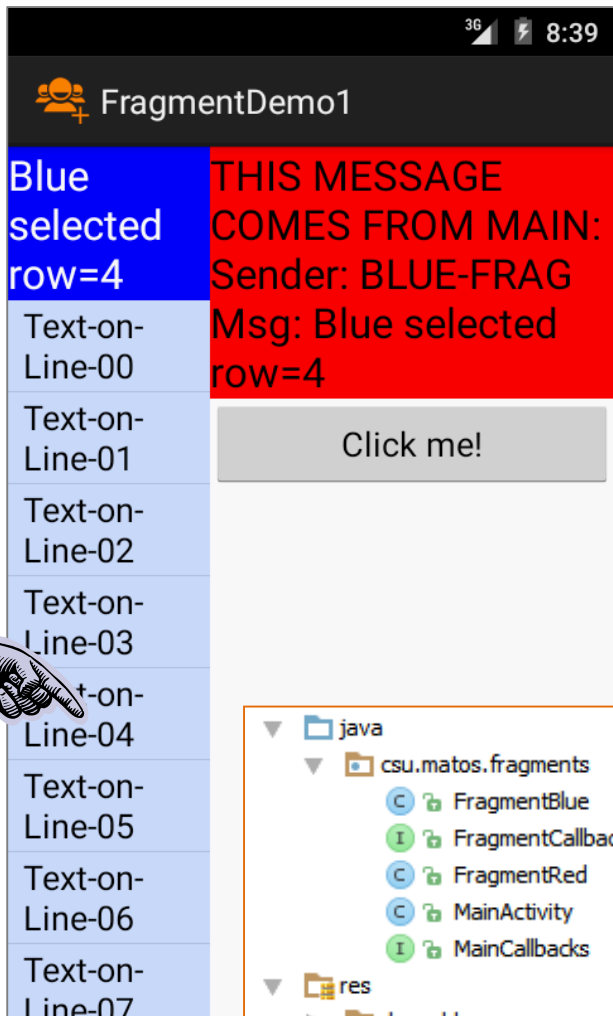
```
// create a new BLUE fragment - show it
ft = getFragmentManager().beginTransaction();
blueFragment = FragmentBlue.newInstance("new-blue");
ft.replace(R.id.main_holder_blue, blueFragment);
ft.commit();
```

```
// create a new RED fragment - show it
ft = getFragmentManager().beginTransaction();
redFragment = FragmentRed.newInstance("new-red");
ft.replace(R.id.main_holder_red, redFragment);
ft.commit();
```

main_holder_red

main_holder_blue

Example 1 – Dynamic Activity-Fragment Binding



This example shows a master-detail design. It is based on three classes:

- **MainActivity** (host),
- **FragmentBlue** (master) and
- **FragmentRed** (detail)

The **master** portion (on the left) presents a list of items. When the user selects one of them, a message is sent to the host **MainActivity** which in turn forwards the message to the detail fragment (on the right).

The **detail** fragment echoes the value of the selected row.

The interfaces **FragmentCallbacks** and **MainCallbacks** define the methods used to pass messages from the MainActivity to fragments and from fragments to MainActivity respectively.

Example 1 – Dynamic Activity-Fragment Binding

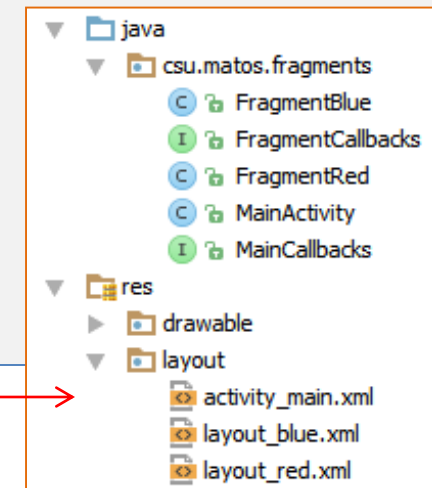
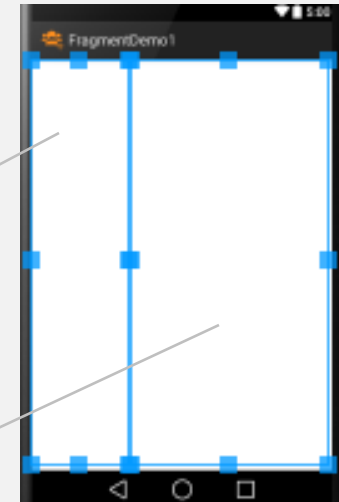
XML LAYOUT: activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:baselineAligned="false"
    android:orientation="horizontal" >

    <FrameLayout
        android:id="@+id/main_holder_blue"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:orientation="vertical" />

    <FrameLayout
        android:id="@+id/main_holder_red"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="2"
        android:orientation="vertical" />

</LinearLayout>
```



Example 1 – Dynamic Activity-Fragment Binding

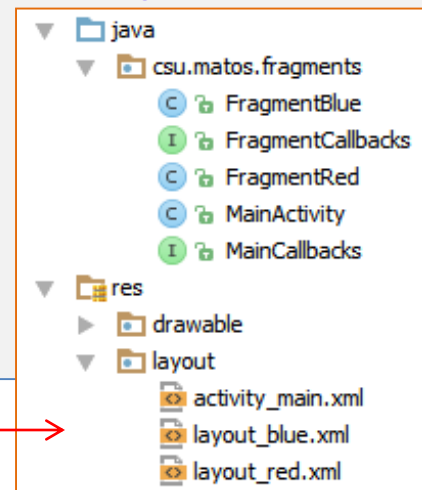
XML LAYOUT: layout_blue.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout_blue"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textView1Blue"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Blue Layout..."
        android:textColor="#ffffffff"
        android:background="#ff0000ff"
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <ListView
        android:id="@+id/listView1Blue"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
    </ListView>

</LinearLayout>
```



Example 1 – Dynamic Activity-Fragment Binding

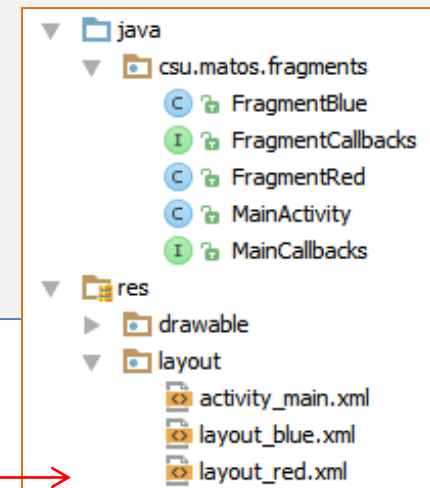
XML LAYOUT: layout_red.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout_red"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/textView1Red"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#ffff0000"
        android:text="Red Layout..."
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <Button
        android:id="@+id/button1Red"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:ems="10"
        android:text="Click me!" />

</LinearLayout>
```



```
//-----  
// GOAL: This example shows an Activity that includes two fragments.  
// Fragments inflate layouts and then get attached to their corresponding  
// layouts in the UI. The example includes two interfaces MainCallbacks  
// and FragmentCallbacks. They implement inter-process communication from  
// Main-to-fragments and from Fragments-to-Main.  
// -----  
public class MainActivity extends Activity implements MainCallbacks {  
  
    FragmentTransaction ft;  
    FragmentRed redFragment;  
    FragmentBlue blueFragment;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        // create a new BLUE fragment - show it  
        ft = getFragmentManager().beginTransaction();  
        blueFragment = FragmentBlue.newInstance("first-blue");  
        ft.replace(R.id.main_holder_blue, blueFragment);  
        ft.commit();  
  
        // create a new RED fragment - show it  
        ft = getFragmentManager().beginTransaction();  
        redFragment = FragmentRed.newInstance("first-red");  
        ft.replace(R.id.main_holder_red, redFragment);  
        ft.commit();  
    }  
}
```

```
// MainCallback implementation (receiving messages coming from Fragments)
@Override
public void onMsgFromFragToMain(String sender, String strValue) {
    // show message arriving to MainActivity
    Toast.makeText(getApplicationContext(),
        " MAIN GOT>> " + sender + "\n" + strValue, Toast.LENGTH_LONG)
        .show();

    if (sender.equals("RED-FRAG")) {
        // TODO: if needed, do here something on behalf of the RED fragment
    }

    if (sender.equals("BLUE-FRAG")) {
        try {
            // forward blue-data to redFragment using its callback method
            redFragment.onMsgFromMainToFragment("\nSender: " + sender
                + "\nMsg: " + strValue);
        } catch (Exception e) {
            Log.e("ERROR", "onStrFromFragToMain " + e.getMessage());
        }
    }
}
```

COMMENTS

1. Each fragment is safely created inside a TRANSACTION frame demarcated by: **beginTransaction ... commit**.
2. An invocation to the special *newInstance* constructor is used to supply any arguments a fragment may need to begin working.
3. Once created, the new fragment is used to **replace** whatever is shown at a designated area of the GUI (as defined in the *activity_main.xml* layout).
4. The method **onMsgFromFragToMain** implements the MainCallbacks interface. It accepts messages asynchronously sent from either *redFragment* or *blueFragment* to the enclosing MainActivity.
5. In our example, the row number selected from the *blueFragment* is forwarded to the *redFragment* using the fragment's callback method **onMsgFromMainToFragment**.


```
public class FragmentBlue extends Fragment {
    // this fragment shows a ListView
    MainActivity main;
    Context context = null;
    String message = "";

    // data to fill-up the ListView
    private String items[] = { "Text-on-Line-00", "Text-on-Line-01",
        "Text-on-Line-02", "Text-on-Line-03", "Text-on-Line-04",
        "Text-on-Line-05", "Text-on-Line-06", "Text-on-Line-07",
        "Text-on-Line-08", "Text-on-Line-09", "Text-on-Line-10", };

    // convenient constructor(accept arguments, copy them to a bundle, binds bundle to fragment)
    public static FragmentBlue newInstance(String strArg) {
        FragmentBlue fragment = new FragmentBlue();
        Bundle args = new Bundle();
        args.putString("strArg1", strArg);
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        try {
            context = getActivity(); // use this reference to invoke main callbacks
            main = (MainActivity) getActivity();
        } catch (IllegalStateException e) {
            throw new IllegalStateException(
                "MainActivity must implement callbacks");
        }
    }
}
```

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                          Bundle savedInstanceState) {

    // inflate res/layout_blue.xml to make GUI holding a TextView and a ListView
    LinearLayout layout_blue = (LinearLayout) inflater.inflate(R.layout.layout_blue, null);
    // plumbing - get a reference to textview and listview
    final TextView txtBlue = (TextView) layout_blue.findViewById(R.id.textView1Blue);
    ListView listView = (ListView) layout_blue.findViewById(R.id.listView1Blue);
    listView.setBackgroundColor(Color.parseColor("#ffccddff"));
    // define a simple adapter to fill rows of the listview
    ArrayAdapter<String> adapter = new ArrayAdapter<String>(context,
                                                         android.R.layout.simple_list_item_1, items);

    listView.setAdapter(adapter);
    // show listview from the top
    listView.setSelection(0);
    listView.smoothScrollToPosition(0);
    // react to click events on listview's rows
    listView.setOnItemClickListener(new OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View v, int position, long id) {
            // inform enclosing MainActivity of the row's position just selected
            main.onMsgFromFragToMain("BLUE-FRAG", "Blue selected row=" + position);
            txtBlue.setText("Blue selected row=" + position);
        }
    });
    // do this for each row (ViewHolder-Pattern could be used for better performance!)
    return layout_blue;

} // onCreateView
} // class
```

COMMENTS

1. The `Class.newInstance(...)` construction is a reflective method commonly used for creating instances of classes (regardless of the number of parameters).
2. Creating an Android fragment begins with the making of a new `Bundle` in which each of the supplied arguments is stored as a `<key,value>` entry. Then the bundle is bound to the fragment through the `.setArguments(...)` method. Finally the newly created fragment is returned.
3. In our example, the **onCreate** method verifies that the `MainActivity` implements the Java Interface defining methods needed to send data from the fragment to the `MainActivity`.
4. Fragments do most of their work inside of **onCreateView**. In this example, the predefined `layout_blue.xml` is inflated and plumbing is done to access its internal widgets (a `TextView` and a `ListView`).
5. A simple `ArrayAdapter` is used to fill the rows of the `ListView`.
6. An event handler is set on the `ListView`, so when the user clicks on a row its position is sent to the `MainActivity`'s listener **onMsgFromFragToMain**.

```
public class FragmentRed extends Fragment implements FragmentCallbacks {
    MainActivity main;
    TextView txtRed;
    Button btnRedClock;

    public static FragmentRed newInstance(String strArg1) {
        FragmentRed fragment = new FragmentRed();
        Bundle bundle = new Bundle();
        bundle.putString("arg1", strArg1);
        fragment.setArguments(bundle);
        return fragment;
    } // newInstance

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Activities containing this fragment must implement interface: MainCallbacks
        if (!(getActivity() instanceof MainCallbacks)) {
            throw new IllegalStateException( " Activity must implement MainCallbacks");
        }
        main = (MainActivity) getActivity(); // use this reference to invoke main callbacks
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // inflate res/layout_red.xml which includes a textview and a button
        LinearLayout view_layout_red = (LinearLayout) inflater.inflate(
            R.layout.layout_red, null);

        // plumbing - get a reference to widgets in the inflated layout
        txtRed = (TextView) view_layout_red.findViewById(R.id.textView1Red);
    }
}
```

```
// show string argument supplied by constructor (if any!)
try {
    Bundle arguments = getArguments();
    String redMessage = arguments.getString("arg1", "");
    txtRed.setText(redMessage);
} catch (Exception e) {
    Log.e("RED BUNDLE ERROR - ", "" + e.getMessage());
}
// clicking the button changes the time displayed and sends a copy to MainActivity
btnRedClock = (Button) view_layout_red.findViewById(R.id.button1Red);
btnRedClock.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {
        String redMessage = "Red clock:\n" + new Date().toString();
        txtRed.setText(redMessage);
        main.onMsgFromFragToMain("RED-FRAG", redMessage);
    }
});
return view_layout_red;
}

@Override
public void onMsgFromMainToFragment(String strValue) {
    // receiving a message from MainActivity (it may happen at any point in time)
    txtRed.setText("THIS MESSAGE COMES FROM MAIN:" + strValue);
}

} // FragmentRed
```

COMMENTS

1. This is very similar to the previous snippet describing the composition of `FragmentBlue`.
2. As before, *`newInstance`* is invoked to create an instance of this class.
3. The `FragmentRed` class uses `onCreate` to verify the `MainActivity` has implemented the methods needed to send messages to it.
4. Observe that `FragmentRed` asynchronously receives messages from the `MainActivity` by means of its **`onMsgFromMainToFragment`** listener.
5. In our example the position of the row selected by the `blueFragment` is accepted. This could be used to properly respond to that event, for instance providing details for the given selection.

Example 1 – Dynamic Binding - Callbacks

```
// method(s) to pass messages from fragments to MainActivity

public interface MainCallbacks {

    public void onMsgFromFragToMain (String sender, String strValue);

}
```

```
// method(s) to pass messages from MainActivity to Fragments

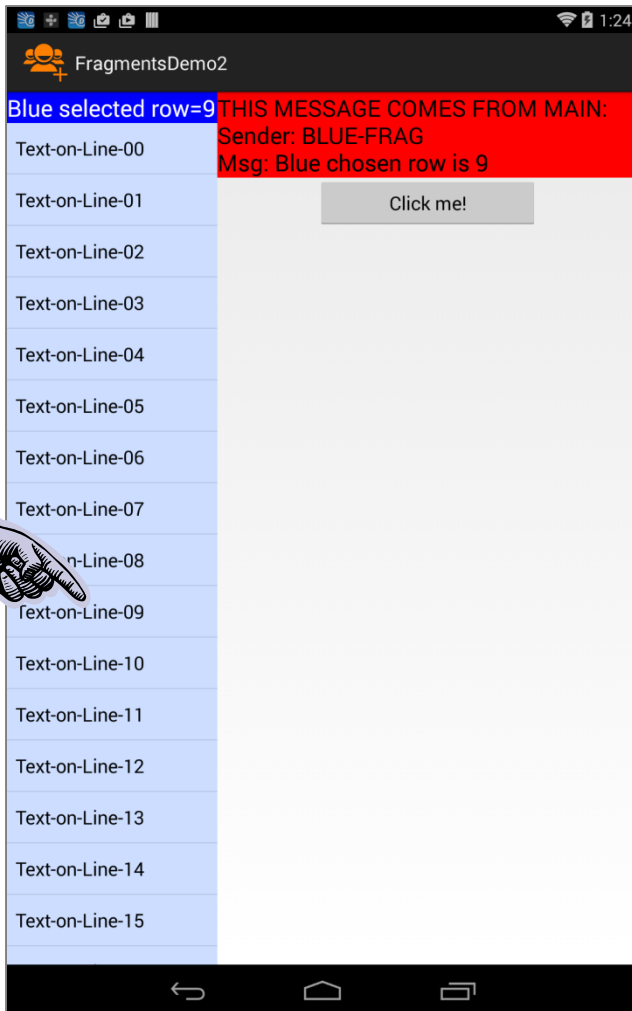
public interface FragmentCallbacks {

    public void onMsgFromMainToFragment(String strValue);

}
```

These Java Interfaces are used to enforce a formal **Inter-Process-Communication** behavior between the Fragments and the MainActivity. During their **onCreate** method each participant can check that the other has implemented the required listeners.

Example 2 – Static Activity-Fragment Binding



This example shows the same master-detail design introduced in the previous example. Like before, it is based on three classes:

- **MainActivity** (host),
- **FragmentRed** (master) and
- **FragmentBlue** (detail)

The main difference between Example1 and Example2 stems from the way the GUI defined by Example2's **MainActivity** statically ties the screen to particular Fragments.

The next pages will show the new *activity_main.xml* layout and the **MainActivity**. All the other components remain exactly the same.

Later on, you may break the GUI-Fragment bound. Just define a new Fragment instance and replace the appropriated GUI portion you want to modify.

Example 2 – Static Activity-Fragment Binding

- Static binding is simple and requires less programming than dynamic binding.
- This approach is appropriate for apps in which the interface retains the same fragment(s) for their entire session.
- Statically attached fragments cannot be removed (however other fragments can be *added* to the interface).
- The Activity's layout file uses the **<fragment>** element to mark the position and size of the area on which a fragment instance is to be injected.
- The following attributes can be used to identify the fragment in case it needs to be restarted (if none is provided the fragment is identified by the system's id of the fragment's container id)
 1. **android:name="AppPackageName.FragmentClassName"**
 2. **android:id="@id+/uniqueName" or
android:tag="string"**

Example 2 – Static Activity-Fragment Binding

XML LAYOUT: activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
```

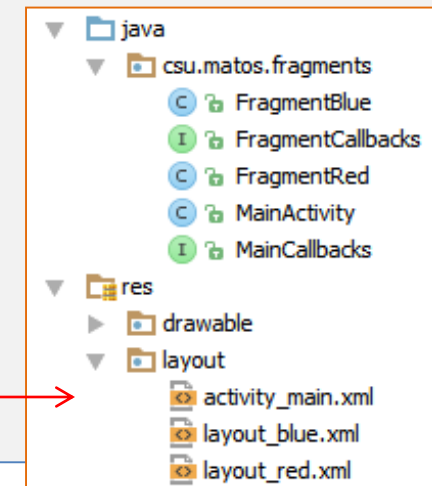
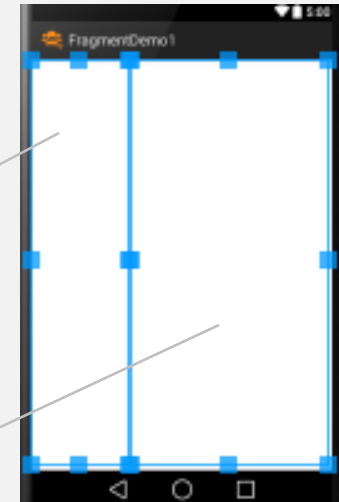
<fragment

```
    android:id="@+id/main_holder_blue"
    android:name="csu.matos.fragments.FragmentBlue"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1" />
```

<fragment

```
    android:id="@+id/main_holder_red"
    android:name="csu.matos.fragments.FragmentRed"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="2" />
```

```
</LinearLayout>
```




```
public class MainActivity extends Activity implements MainCallbacks {
    FragmentRed redFragment;
    FragmentBlue blueFragment ;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // NOTHING to do, fragments will be automatically created and added to the GUI
    }

    @Override
    public void onAttachFragment(Fragment fragment) {
        super.onAttachFragment(fragment);
        // get a reference to each fragment attached to the GUI
        if (fragment.getClass() == FragmentRed.class ){
            redFragment = (FragmentRed) fragment;

        }
        if (fragment.getClass() == FragmentBlue.class ){
            blueFragment = (FragmentBlue) fragment;

        }
    }
}
```



```
@Override
public void onMsgFromFragToMain(String sender, String strValue) {
    Toast.makeText(getApplicationContext(), " MAIN GOT MSG >> " + sender
        + "\n" + strValue, Toast.LENGTH_LONG).show();

    if (sender.equals("RED-FRAG")){
        //TODO: do here something smart on behalf of BLUE fragment
    }

    if (sender.equals("BLUE-FRAG")) {
        redFragment.onMsgFromActivity("\nSender: " + sender + "\nMsg: " + strValue);
    }

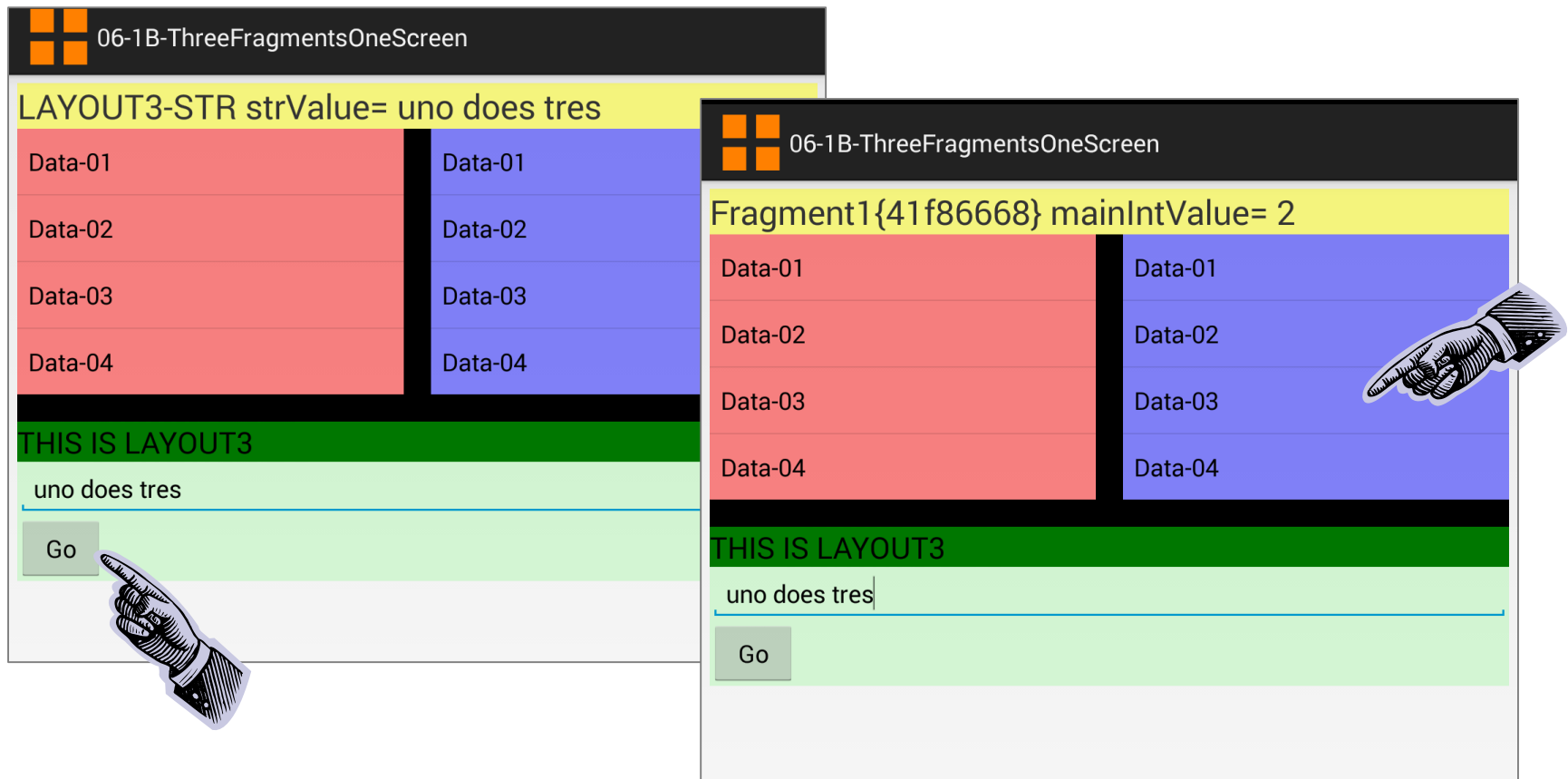
} //onMsgFromFragToMain
}
```

COMMENTS

1. In this example the **onCreate** method has nothing to do. Moreover, **onCreateView** is not even called, observe that the XML-layout clause **android:name="csu.matos.fragments.FragmentXYZ"** defines the specific fragment to be plugged in the activity's screen.
2. When a fragment is moved to the screen the **onAttachFragment** method is executed. This event is used here to keep a reference to the *redFragment* and the *blueFragment*.
3. Messages sent by the blueFragment to the MainActivity are caught in the **onMsgFromFragToMain** listener. As in the previous example, blueFragment messages are forwarded to the redFragment.

Example 3 – Multiple-Fragments-One-Screen

- This example is a minor variation of Example1. The MainActivity displays a screen simultaneously showing three independent fragments.
- All fragments are **visible** and **active**, providing multiple points of interaction with the user. There are two instances of a ListView fragment, and a bottom layout showing a design containing a TextView and a Button.



XML LAYOUT: activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayoutMain"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="6dp" >

    <TextView
        android:id="@+id/txtMsgMain"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#77ffff00"
        android:textSize="25sp" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <LinearLayout
            android:id="@+id/home1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:background="#77ff0000"
            android:orientation="vertical" />
```

Only the main layout is shown as the code for this example is literally the same from Example1.

XML LAYOUT: activity_main.xml

```
<View
    android:layout_width="20dp"
    android:layout_height="match_parent"
    android:background="#ff000000" />

<LinearLayout
    android:id="@+id/home2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:background="#770000ff"
    android:orientation="vertical" />
</LinearLayout>

<View
    android:layout_width="match_parent"
    android:layout_height="20dp"
    android:background="#ff000000" />

<LinearLayout
    android:id="@+id/home3"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical" />
</LinearLayout>
```

Only the main layout is shown as the code for this example is literally the same from Example1.

Example 4 – Saving Fragment State

A regular practice before leaving a fragment is to override its **onSaveInstanceState** method to persist any state data inside a system's managed bundle named *outState*.

Later, you may test for the existence of the saved state bundle and its contents inside any of the following the methods: **onCreate** , **onCreateView** , **onViewCreated**, or **onViewStateRestored**. This is know as a *warm-start*.

Observe that a fresh *cold-start* execution passes to the fragment a **null** bundle. Unlike Activities, Fragments do to have an **onRestoreInstanceState** method.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...
    stateData = getArguments().getString("arg1", "cold-start");

    if (savedInstanceState != null){
        stateData = savedInstanceState.getString("arg1", "warm-default");
    }
} //onCreate
...
@Override
public void onSaveInstanceState(Bundle outState) {
    ...
    outState.putString("arg1", stateData);
    super.onSaveInstanceState(outState);
} //onSaveInstanceState
```

Operations on Fragments

There are various operations that affect the presence and visibility of fragments dynamically bound to an activity. Those operations must be applied inside the scope of a **FragmentManager** object.

- add()** Add a fragment to an activity (generally showing a view). If the activity is re-started due to a configuration-change, previously created fragments that appeared on the UI via add() can be reused (better performance, no need to re-create the fragment).
- remove()** Remove a fragment from the activity. Fragment is destroyed (unless it was also added to the BackStack).
- replace()** A fragment currently on the UI is destroyed and replaced by another fragment.
- show() / hide()** Shows a previously hidden fragment (hidden but not destroyed).
- attach() / detach()** Attaches a fragment previously separated (detached) from the UI. Detached fragments are invisible but not destroyed.

Operations on Fragments

Consider the following code sample on which a sequence of opposite operations is applied to display a fragment.

```
FragmentTransaction ft = getFragmentManager().beginTransaction();

redFragment = FragmentRed.newInstance(intValue);
ft.add(R.id.main_holder, redFragment, "RED-TAG");

ft.hide(redFragment);
ft.show(redFragment);

ft.detach(redFragment);
ft.attach(redFragment);

ft.commit();
```

Using the BackStack to Recreate State

Android-OS introduced a special stack to help fragments keep state when the user navigates from one UI to the other.

The artifact is called the **BackStack** and allows push/pop operations to manage **FragmentTransactions**. The BackStack mirrors the behavior of the activity stack within a single activity

Remember that *all* Android devices include a **Back** button. **When this button is pressed in succession the app transitions to the previous screen shown by the app until it ends.** This mechanism provides a natural *historical* navigation (also known as *Back-Navigation*). Another important pattern of navigation known as *Child-to-HighAncestor* is discussed later.

Why should BackStack be used?

When the BackStack is used, the retrieved fragment is *re-used* (instead of re-created from scratch) and its state data *transparently* restored (no need for input/output state bundles). This approach leads to simpler and more efficient apps.

Using the BackStack to Recreate State

A typical sequence to create a fragment and add it to the BackStack follows:

```
FragmentTransaction ft = getFragmentManager().beginTransaction();
Fragment redFragment = FragmentRed.newInstance(intParameterValue);
ft.replace(R.id.main_holder, redFragment, "RED-FRAG");
> ft.addToBackStack("RED_UI");
ft.commit();
```

In this example a fragment transaction (*ft*) adds a *redFragment* to the main activity's UI. The fragment uses the optional tag/alias "RED-FRAG", as an alternative form of identification. Later, we may inspect the app's UI, and find the 'alias' of the fragment held inside the *main_holder* container.

Before the transaction commits, the statement

```
ft.addToBackStack("RED_UI");
```

pushes a reference of the current transaction's environment in the BackStack including the optional identification tag: "RED_UI". Later on, we may search through the BackStack looking for an entry matching the tag value. When found and popped, it resets the UI to the state held by that transaction.

Using the BackStack to Recreate State

Navigation

Retrieving entries from the BackStack can be done in various ways, such as:

- Pressing the **Back** button to trigger a historical navigation exposing in succession the previous User-Interfaces.
- Invoking the method **.popBackStackImmediate(...)** to selectively restore any particular BackStackEntry holding an UI already shown to the user.

```
// Remove current fragment's UI and show its previous screen
try {
    FragmentTransaction ft = getFragmentManager().beginTransaction();

    android.app.FragmentManager fragmentManager = getFragmentManager();

    1 → int bsCount = fragmentManager.getBackStackEntryCount();
    String tag = fragmentManager.getBackStackEntryAt(bsCount-1).getName();
    2 → int id = fragmentManager.getBackStackEntryAt(bsCount-1).getId();
    Log.e("PREVIOUS Fragment: ", "" + tag + " " + id);

    3 → fragmentManager.popBackStackImmediate(id, 1); //supply: id or tag

    ft.commit();

} catch (Exception e) {
    Log.e("REMOVE>>> ", e.getMessage() );
}
```

Using the BackStack to Recreate State

Navigating Through the BackStack

In the previous transaction we reproduce the behavior of the **Back** key when used for historical navigation.

1. The size of the BackStack is determined (`getBackStackEntryCount`)
2. The top element of the stack is inspected. First we obtain its tag and later its numerical id by calling the method:
`fragmentManager.getBackStackEntryAt(bsCount-1).getId()`.
3. The `.popBackStack(id, 1)` method removes BackStackEntries from the top of the BackStack until it finds the entry matching the supplied **id**. At this point the app's UI is updated showing the screen associated to the matching transaction previously held in the stack.

Using the BackStack to Recreate State

Navigating Through the BackStack

The following code clears the current BackStack. All fragment transactions pushed by calling the method `ft.addToBackStack(...)` are deleted. The app's UI is updated, removing all screens shown by fragments that put a reference to themselves in the BackStack.

This approach could be used to provide *Child-to-HighAncestor* navigation.

```
try {
    FragmentTransaction ft = getFragmentManager().beginTransaction();

    android.app.FragmentManager fragmentManager = getFragmentManager();

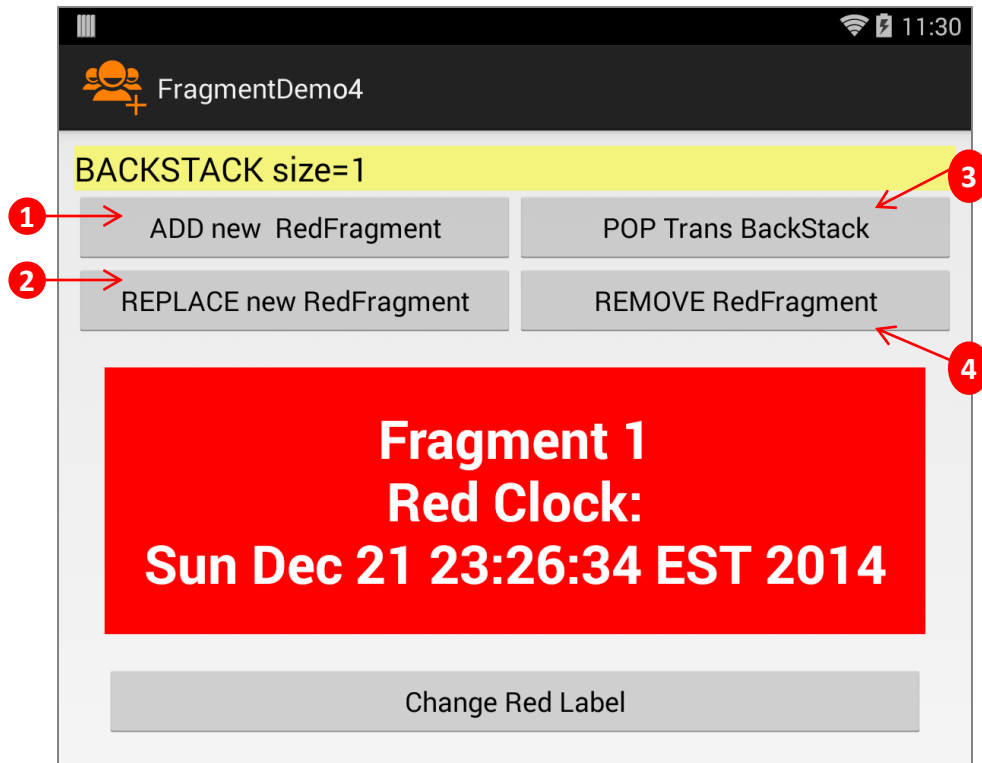
    fragmentManager.popBackStackImmediate(null,
        FragmentManager.POP_BACK_STACK_INCLUSIVE);

    ft.commit();

} catch (Exception e) {
    Log.e("CLEAR-STACK>>> ", e.getMessage() );
}
```


Example 5 - Using the BackStack

1 of x

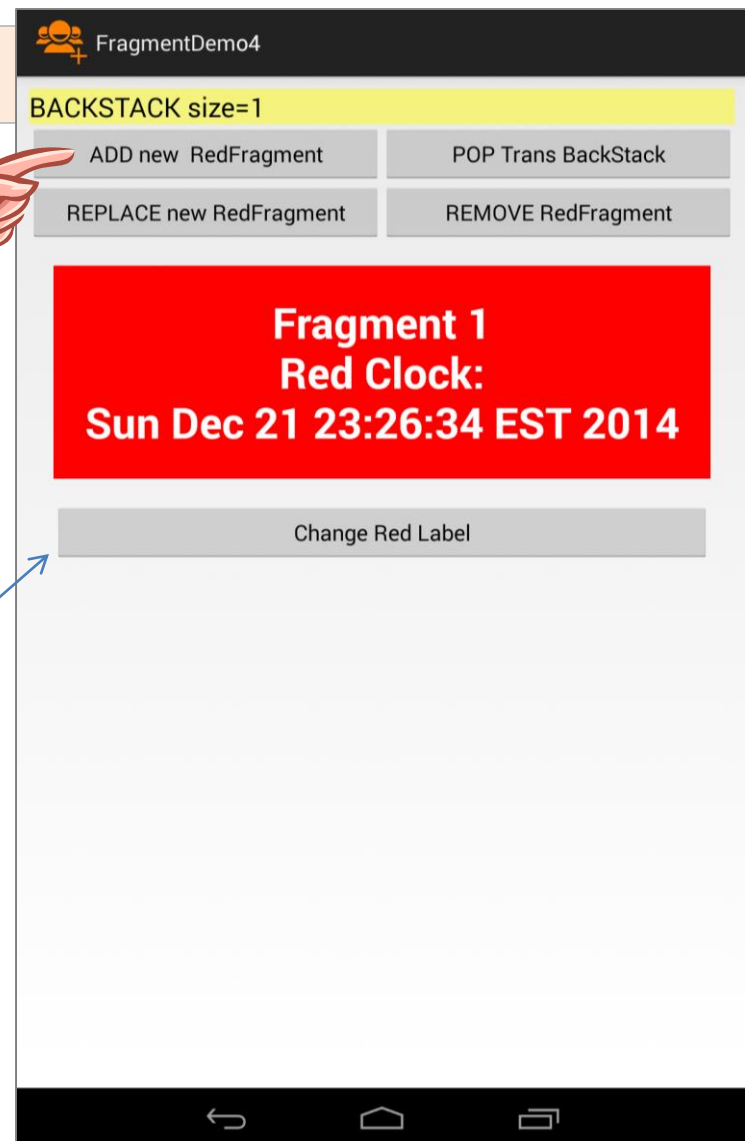
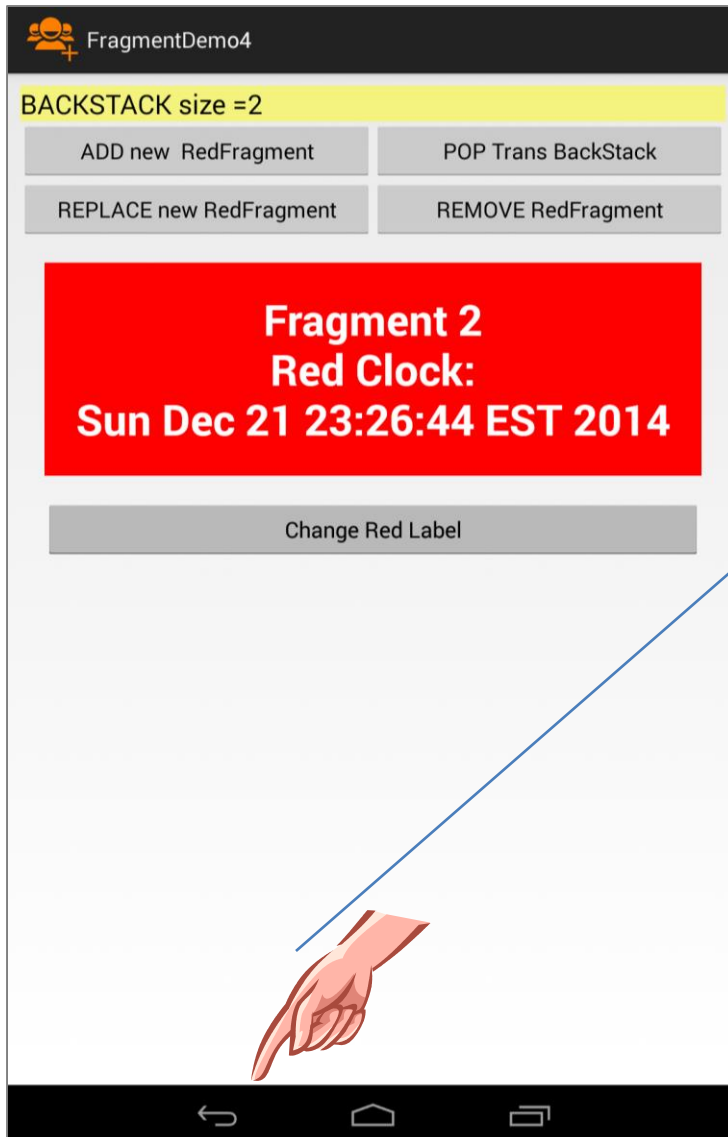


1. A new *redFragment* is created. Its view is attached to the activity's UI using the **add()** method. Finally its enclosing transaction is pushed on the BackStack.
2. As above, however; the fragment's view is attached to the activity's UI using the **replace()** method (old view is destroyed). The current transaction is also added to the BackStack.
3. Popping an entry from the BackStack removes the current app's UI and navigates back to the previously stored fragment's view. State data (if any) is shown as it was before leaving the view. The size of BackStack is reduced by one.

4. The "Remove" button activates a *findFragmentByTag* search. This first searches through fragments that are currently added to the manager's activity; if no such fragment is found, then all fragments currently on the back stack are searched. In our example, the current view is retired from the UI using **remove()** and the historically previous UI is presented.

Example 5 - Using the BackStack

1. A new *redFragment* is created and its enclosing transaction is added to the BackStack.



2. Pressing the **Back** button removes the current fragment from the UI and Back-Navigates to the previous fragment. Its state is preserved, so you do not need to refill its widgets.

Example 5. LAYOUT: activity_main.xml

1 of 3

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:baselineAligned="false"
    android:orientation="vertical"
    android:padding="10dp" >
```

<TextView

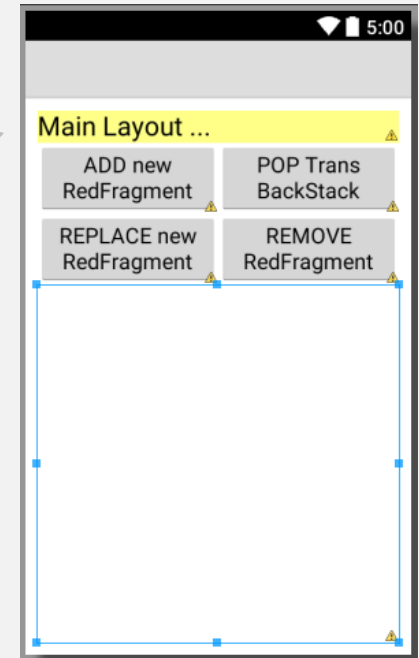
```
    android:id="@+id/textView1Main"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="#77ffff00"
    android:text="Main Layout ..."
    android:textAppearance="?android:attr/textAppearanceLarge" />
```

<LinearLayout

```
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:baselineAligned="false"
    android:orientation="horizontal" >
```

<Button

```
    android:id="@+id/button1MainShowRed"
    android:layout_width="150dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="ADD new RedFragment" />
```



<Button

```
    android:id="@+id/button2MainPop"  
    android:layout_width="150dp"  
    android:layout_height="wrap_content"  
    android:layout_weight="1"  
    android:text="POP Trans BackStack" />
```

</LinearLayout>

<LinearLayout

```
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:baselineAligned="false"  
    android:orientation="horizontal" >
```

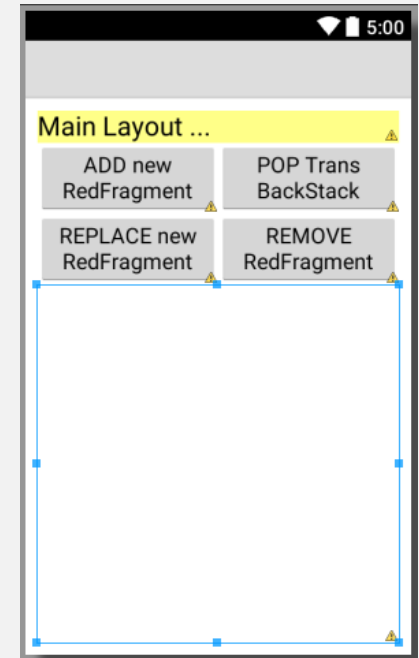
<Button

```
    android:id="@+id/button4MainReplace"  
    android:layout_width="150dp"  
    android:layout_height="wrap_content"  
    android:layout_weight="1"  
    android:text="REPLACE new RedFragment" />
```

<Button

```
    android:id="@+id/button3MainRemove"  
    android:layout_width="150dp"  
    android:layout_height="wrap_content"  
    android:layout_weight="1"  
    android:text="REMOVE RedFragment" />
```

</LinearLayout>



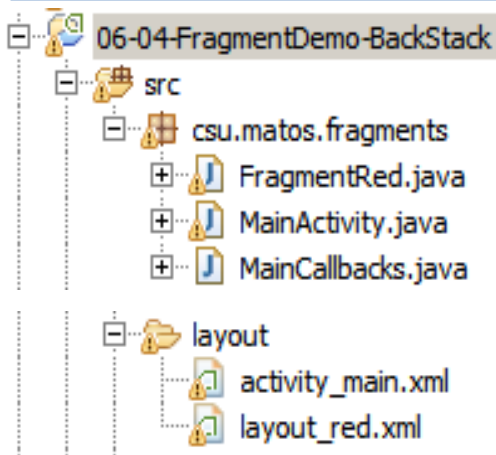
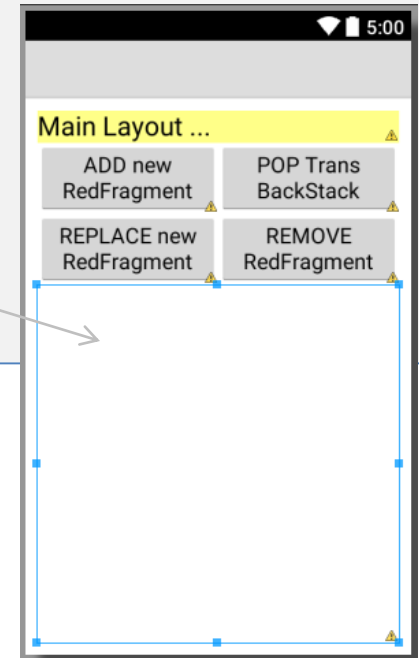
Example 5. LAYOUT: activity_main.xml

3 of 3

```
<FrameLayout
```

```
    android:id="@+id/main_holder"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_weight="2"  
    android:orientation="vertical" />
```

```
</LinearLayout>
```



MainCallbacks

```
package csu.matos.fragments;  
// method(s) to pass messages from fragments to MainActivity  
  
public interface MainCallbacks {  
    public void onMsgFromFragToMain ( String sender, String strValue);  
}
```

Example 5. LAYOUT: layout_red.xml

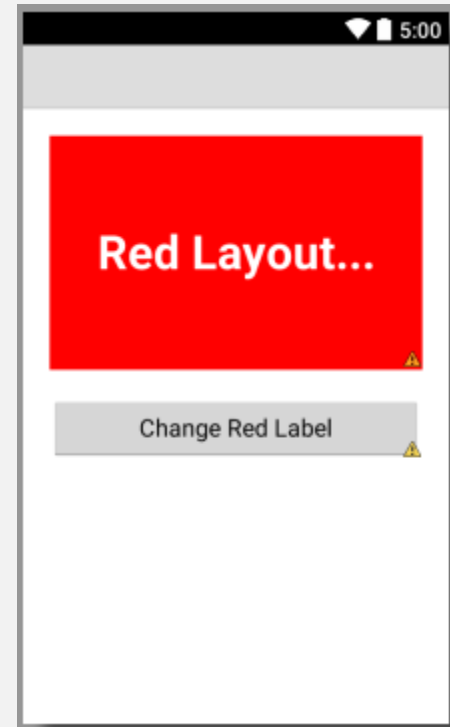
3 of 3

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout_red"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textView1Red"
        android:layout_width="match_parent"
        android:layout_height="175dp"
        android:layout_margin="20dp"
        android:background="#ffff0000"
        android:gravity="center"
        android:text="Red Layout..."
        android:textColor="@android:color/white"
        android:textSize="35sp"
        android:textStyle="bold" />

    <Button
        android:id="@+id/button1Red"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:layout_marginRight="20dp"
        android:text="Change Red Label" />

</LinearLayout>
```



Example 5. MainActivity

1 of 4

```
public class MainActivity extends Activity implements MainCallbacks, OnClickListener {

    FragmentTransaction ft;
    FragmentRed redFragment;
    TextView txtMsg;
    Button btnAddRedFragment;
    Button btnReplaceRedFragment;
    Button btnPop;
    Button btnRemove;
    int serialCounter = 0; //used to enumerate fragments
    String redMessage;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        txtMsg = (TextView) findViewById(R.id.textView1Main);
        btnAddRedFragment = (Button) findViewById(R.id.button1MainShowRed);
        btnReplaceRedFragment = (Button) findViewById(R.id.button4MainRePlace);

        btnPop = (Button) findViewById(R.id.button2MainPop);
        btnRemove = (Button) findViewById(R.id.button3MainRemove);
        btnAddRedFragment.setOnClickListener(this);
        btnReplaceRedFragment.setOnClickListener(this);
        btnPop.setOnClickListener(this);
        btnRemove.setOnClickListener(this);
    }
}
```

Example 5. MainActivity

2 of 4

```
// Callback (receiving messages coming from Fragments)
@Override
public void onMsgFromFragToMain(String sender, String strValue) {
    // show message arriving to MainActivity
    txtMsg.setText( sender + "=>" + strValue );
}
// -----
public void onClick(View v) {

    if(v.getId() == btnAddRedFragment.getId() ){
        addRedFragment(++serialCounter);
    }

    if(v.getId() == btnReplaceRedFragment.getId() ){
        replaceRedFragment(++serialCounter);
    }

    if(v.getId() == btnPop.getId() ){
        FragmentManager fragmentManager = getFragmentManager();
        int counter = fragmentManager.getBackStackEntryCount();
        txtMsg.setText("BACKSTACK old size=" + counter);

        if(counter>0) {
            // VERSION 1 [ popBackStack could be used as opposite of addBackStack() ]
            // pop takes a Transaction from the BackStack and a view is also deleted
            fragmentManager.popBackStackImmediate();
            txtMsg.append("\nBACKSTACK new size=" + fragmentManager.getBackStackEntryCount() );
        }
    }
}
} //Pop
```



```
if(v.getId() == btnRemove.getId() ){
    FragmentManager fragmentManager = getFragmentManager();
    int counter = fragmentManager.getBackStackEntryCount();
    txtMsg.setText("BACKSTACK old size=" + counter);

    // VERSION 2 -----
    // Removes an existing fragment from the fragmentTransaction.
    // If it was added to a container, its view is also removed from that
    // container. The BackStack may remain the same!
    Fragment f1 = fragmentManager.findFragmentByTag("RED-TAG");
    fragmentManager.beginTransaction().remove(f1).commit();
    txtMsg.append("\nBACKSTACK new size=" + fragmentManager.getBackStackEntryCount() );

    // VERSION 3 -----
    // Fragment f1 = fragmentManager.findFragmentById(R.id.main_holder);
    // fragmentManager.beginTransaction().remove(f1).commit();
    // txtMsg.append("\nBACKSTACK new size=" + fragmentManager.getBackStackEntryCount() );

} //Remove

} //onClick

@Override
public void onBackPressed() {
    super.onBackPressed();
    int counter = getFragmentManager().getBackStackEntryCount();
    txtMsg.setText("BACKSTACK size=" + counter);
}
```

Example 5. MainActivity

4 of 4

```
public void addRedFragment(int intValue) {
    // create a new RED fragment, add fragment to the transaction
    FragmentTransaction ft = getFragmentManager().beginTransaction();
    redFragment = FragmentRed.newInstance(intValue);
    ft.add(R.id.main_holder, redFragment, "RED-TAG");
    ft.addToBackStack("MYSTACK1");
    ft.commit();

    // complete any pending insertions in the BackStack, then report its size
    getFragmentManager().executePendingTransactions();
    txtMsg.setText("BACKSTACK size =" + getFragmentManager().getBackStackEntryCount() );
}

public void replaceRedFragment(int intValue) {
    // create a new RED fragment, replace fragments in the transaction
    FragmentTransaction ft = getFragmentManager().beginTransaction();
    redFragment = FragmentRed.newInstance(intValue);
    ft.replace(R.id.main_holder, redFragment, "RED-TAG");
    ft.addToBackStack("MYSTACK1");
    ft.commit();

    // complete any pending insertions in the BackStack, then report its size
    getFragmentManager().executePendingTransactions();
    txtMsg.setText("BACKSTACK size =" + getFragmentManager().getBackStackEntryCount() );
}
}
```

Example 5. FragmentRed

1 of 2

```
public class FragmentRed extends Fragment {
    MainActivity main;
    TextView txtRed;
    Button btnRedClock;
    int fragmentId;
    String selectedRedText = "";

    public static FragmentRed newInstance(int fragmentId) {
        FragmentRed fragment = new FragmentRed();
        Bundle bundle = new Bundle();
        bundle.putInt("fragmentId", fragmentId);
        fragment.setArguments(bundle);
        return fragment;
    } // newInstance

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

        // Activities containing this fragment must implement MainCallbacks
        if (!(getActivity() instanceof MainCallbacks)) {
            throw new IllegalStateException(
                ">>> Activity must implement MainCallbacks");
        }
        main = (MainActivity) getActivity();
        fragmentId = getArguments().getInt("fragmentId", -1);
    }
}
```

Example 5. FragmentRed

2 of 2

```
@Override
public View onCreateView( LayoutInflater inflater, ViewGroup container,
                          Bundle savedInstanceState) {

    LinearLayout view_layout_red = (LinearLayout) inflater.inflate(
        R.layout.layout_red, null);

    txtRed = (TextView) view_layout_red.findViewById(R.id.textView1Red);
    txtRed.setText( "Fragment " + fragmentId );

    btnRedClock = (Button) view_layout_red.findViewById(R.id.button1Red);
    btnRedClock.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
            selectedRedText = "\nRed Clock:\n" + new Date().toString();
            txtRed.append(selectedRedText);
            // main.onMsgFromFragToMain("RED-FRAG", selectedRedText );
        }
    });
    return view_layout_red;
}

} // FragmentRed
```



Fragments

Questions ?

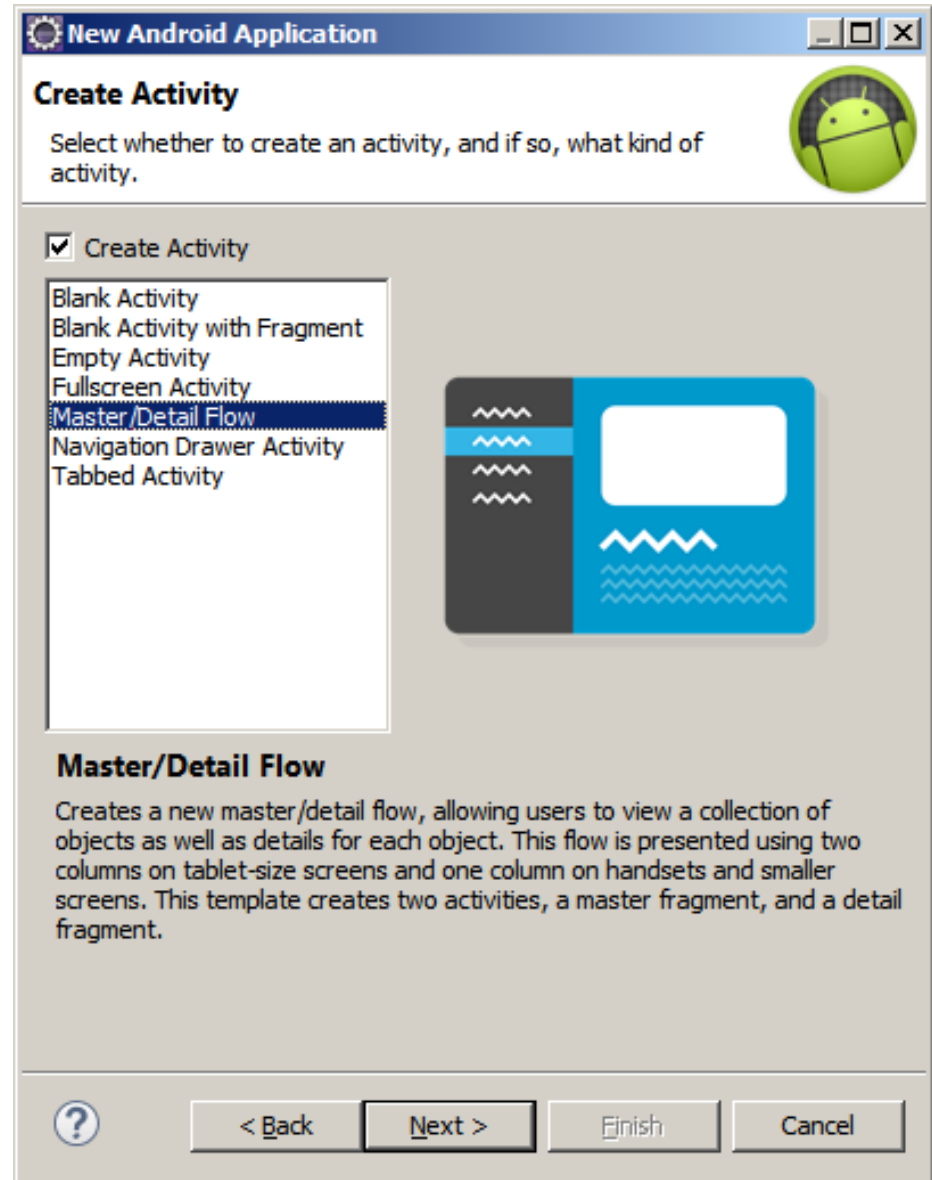
Appendix1. Multi-Pane Apps

The **Master/DetailFlow** design pattern is particularly useful for apps targeting large screen devices.

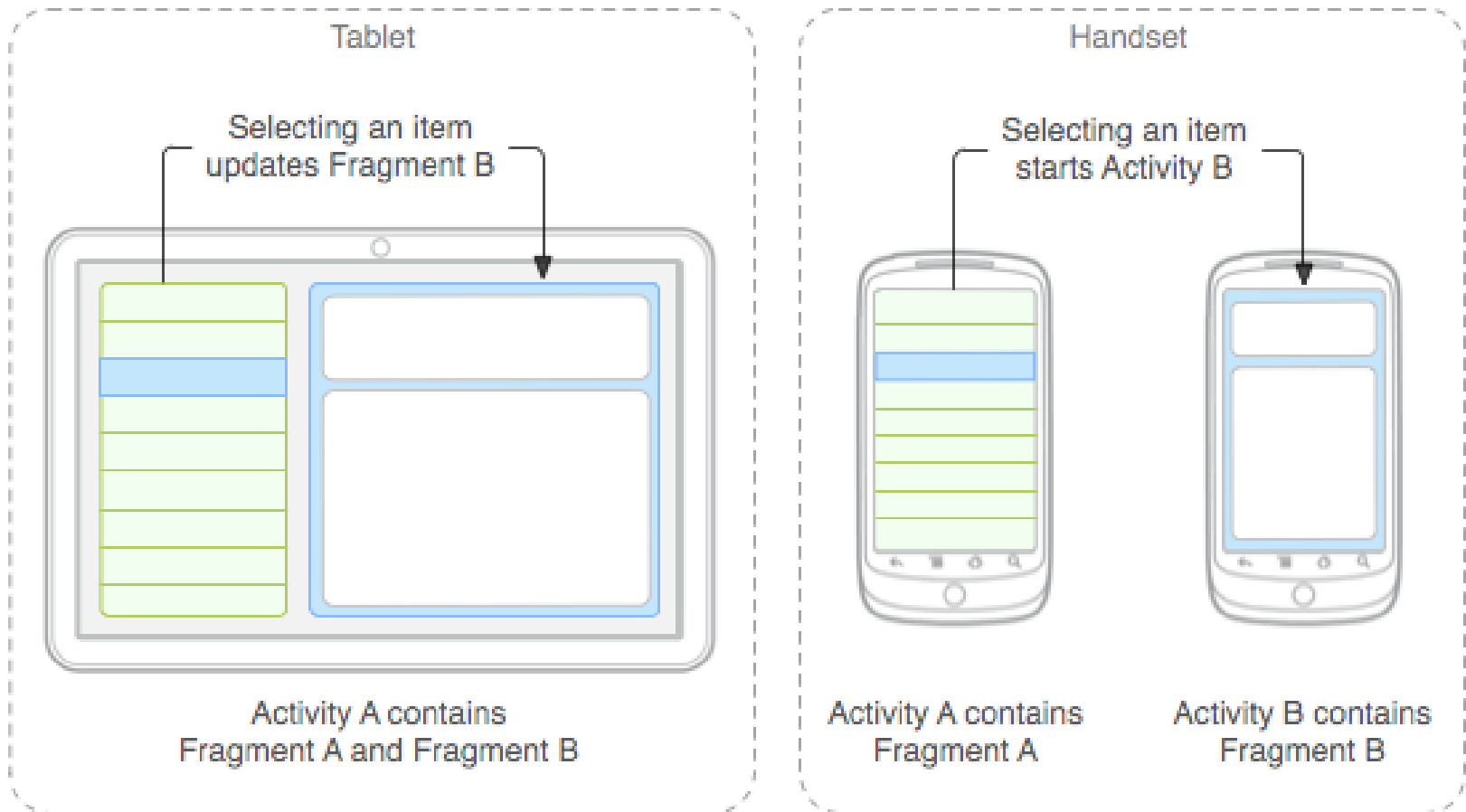
On a large unit, this model shows *-at the same time-* two interdependent columns called **Master** and **Detail**. Clicking a row on the small *master* panel displays its associated information in the *detail* panel.

On small screen units, it shows two separate screens. The first to appear is the master panel. After a selection is made it switches to a second screen displaying the related details.

The next example details the making of a Master/Detail app generated from the ADT wizard. As you will see, there is an extensive use of Activities and Fragments



Appendix1. Multi-Pane Apps



The image above, illustrates how the Master/Detail pattern is shown on a large screen unit (left), and a small handset device (right).

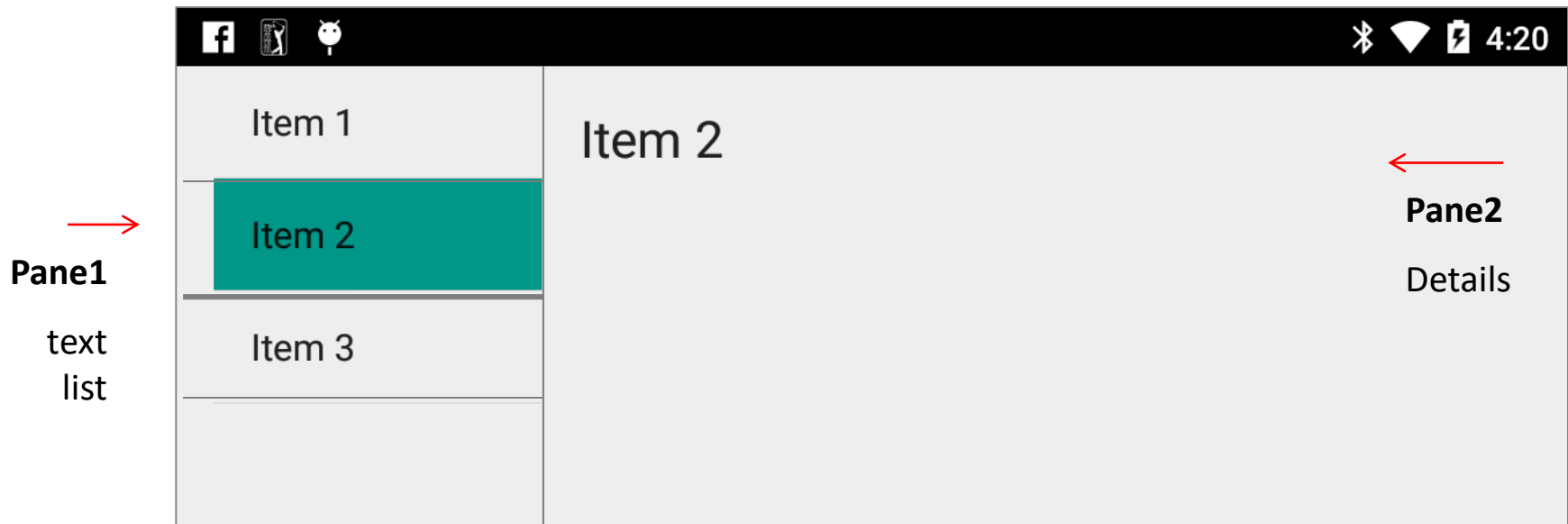
Reference: <http://developer.android.com/guide/components/fragments.html>

Appendix1. Multi-Pane Apps

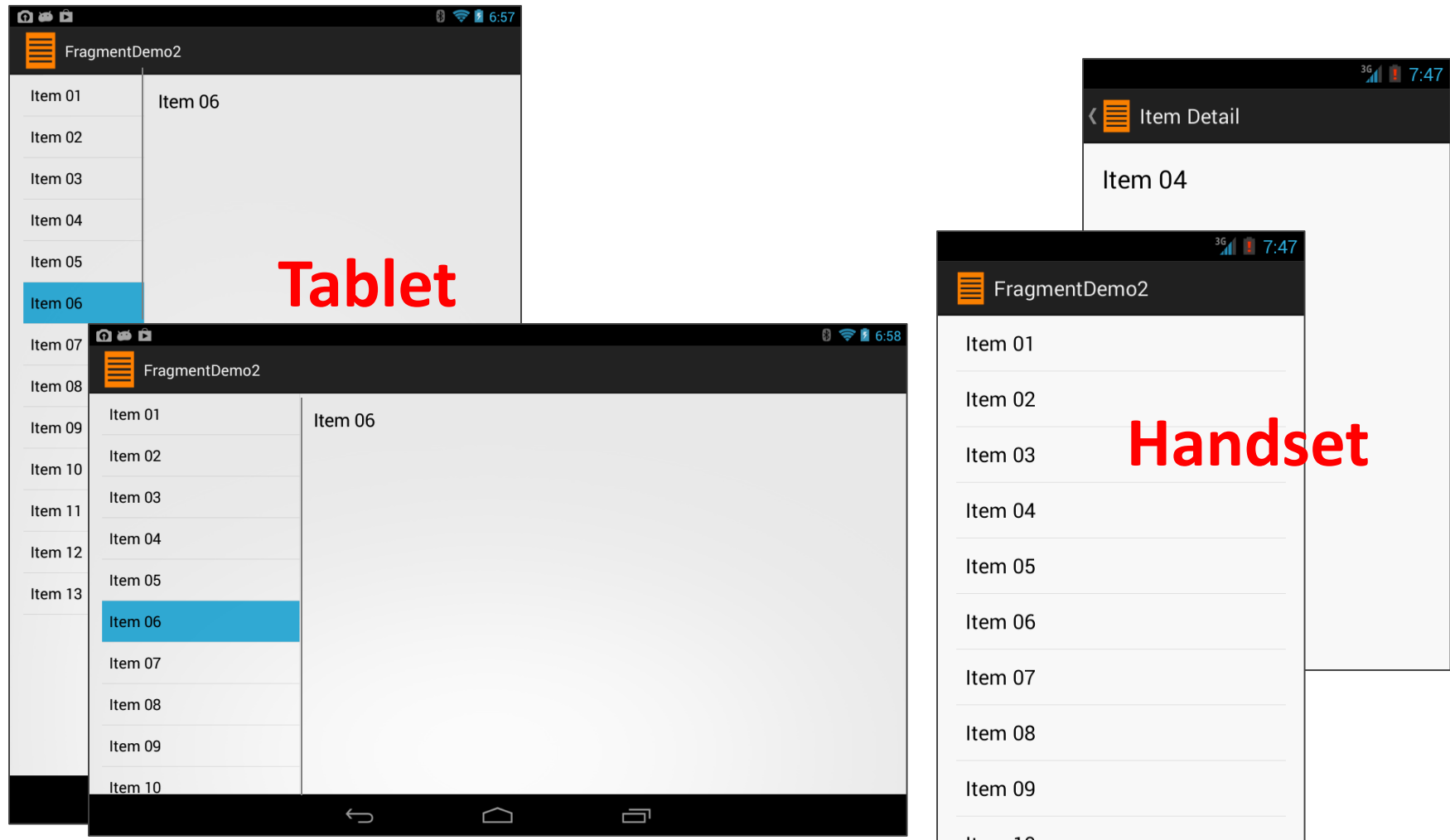


The skeleton **Master/Detail** app, consists of a **master** panel presenting to the user a simple text-based ListView and the **detail** panel is a large view holding a TextView.

Clicking on a master-row echoes the selection on the **Detail** panel. The skeleton-app adjust its presentation to the size of the device. It also preserves state data (row id) and the app behaves well on the case of rotation.



Appendix1. Multi-Pane Apps



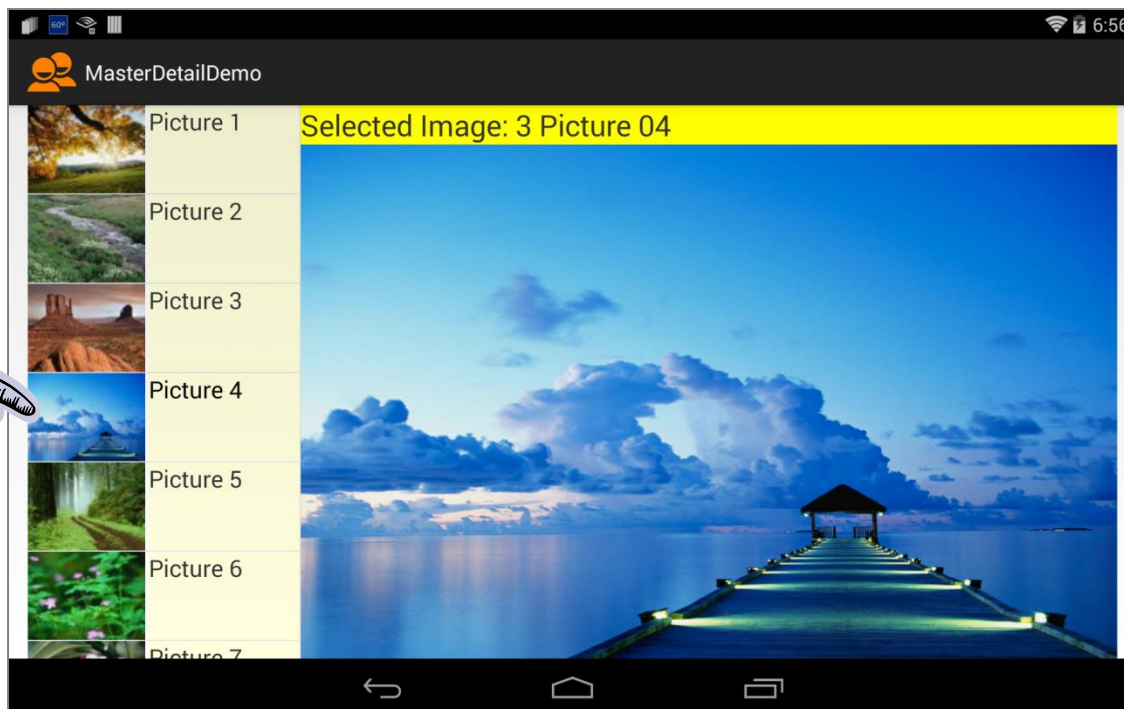
Images from the two-pane app's execution are shown when running on a tablet and a handset.

Appendix1. Multi-Pane Apps



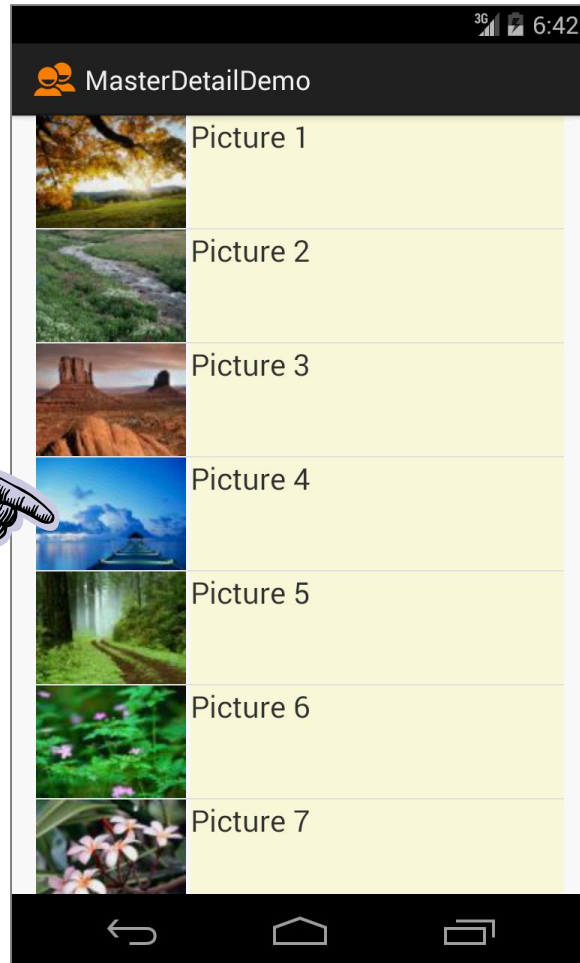
In this example we modify the skeleton app produced by the ADT **Master/Detail** wizard to behave in a way similar to the image below.

The Master panel has been changed to a custom [icon-text] pair. When a selection on the master panel is made, the Detail view shows a high-resolution version of the thumbnail selected from the list.



A single two-panes UI displayed on a 'large-screen' tablet

Appendix1. Multi-Pane Apps



On handsets the previous two-panes single UI is displayed as two separated screens.

Appendix1. Multi-Pane Apps



Step 1. Use the Master/Detail Flow wizard (Eclipse-ADT)

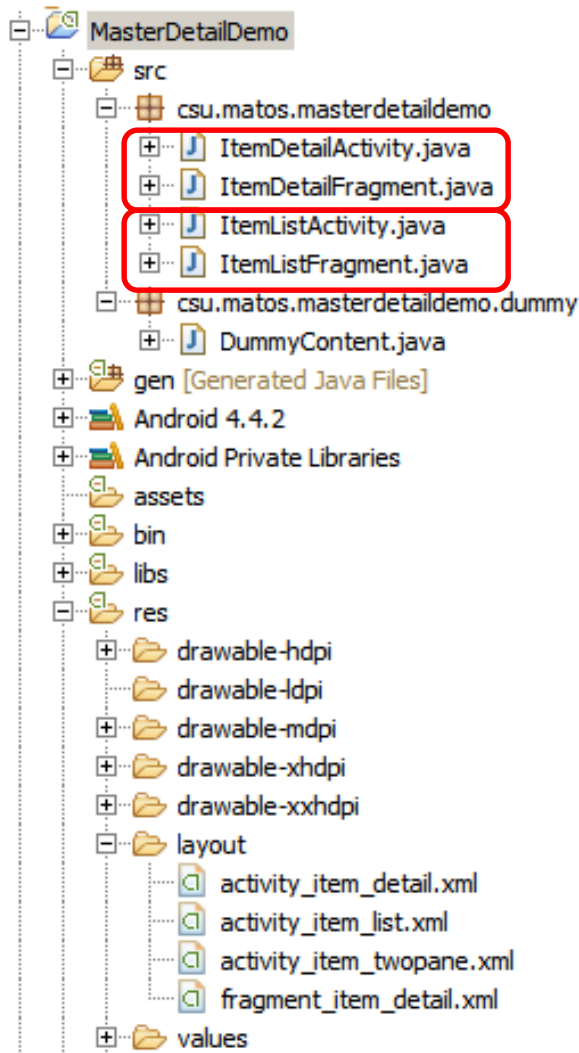
The image shows two screenshots of the Eclipse-ADT wizard. The left screenshot is the 'New Android Application' dialog, which has a title bar 'New Android Application' and a subtitle 'Creates a new Android Application'. It contains several input fields: 'Application Name' (set to 'MasterDetailDemo'), 'Project Name' (set to 'MasterDetailDemo'), 'Package Name' (set to 'csu.matos.masterdetaildemo'), 'Minimum Required SDK' (set to 'API 15: Android 4.0.3 (IceCreamSandwich)'), 'Target SDK' (set to 'API 16: Android 4.1 (Jelly Bean)'), 'Compile With' (set to 'API 19: Android 4.4 (KitKat)'), and 'Theme' (set to 'Holo Light with Dark Action Bar'). A red rectangle highlights the 'Application Name' field. The right screenshot is the 'Create Activity' dialog, which has a title bar 'New Android Application' and a subtitle 'Select whether to create an activity, and if so, what kind of activity.' It contains a list of activity types: 'Blank Activity', 'Blank Activity with Fragment', 'Empty Activity', 'Fullscreen Activity', 'Master/Detail Flow' (highlighted with a red rectangle), 'Navigation Drawer Activity', and 'Tabbed Activity'. Below the list is a section titled 'Master/Detail Flow' with a description: 'Creates a new master/detail flow, allowing users to view a collection of objects as well as details for each object. This flow is presented using two columns on tablet-size screens and one column on handsets and smaller screens. This template creates two activities, a master fragment, and a detail fragment.' Both screenshots have a small Android robot icon in the top right corner.

Warning: Minimum SDK is API-11

Appendix1. Multi-Pane Apps



Step3. Action Plan – What to keep & What to change?



The wizard creates two **xxxActivity.java** classes and two **yyyFragment.java** classes.

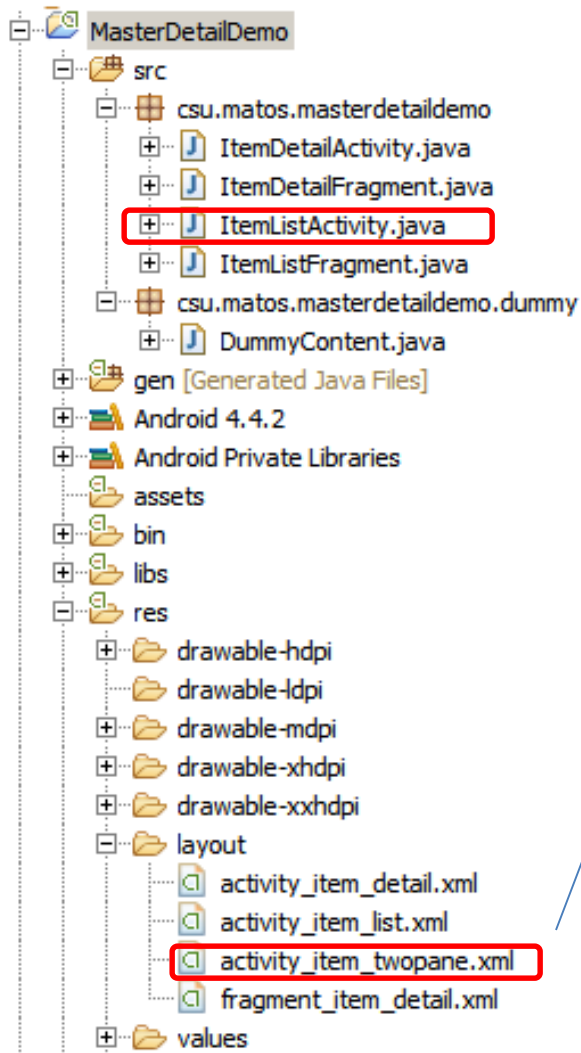
The **xxxActivity** classes are responsible for housekeeping operations such as providing a place on the current device for the fragments to be attached. We will *use them without making any changes* to the logic provided by the ADT wizard.

The **yyyFragments** are responsible for implementing all the ‘intelligent’ operations required by the app. Here is where we will write our business logic, such as populating the master list and later drawing a detailed picture of the selection.

Appendix1. Multi-Pane Apps



Step2. Examine the skeleton app made by the **Master/Detail Flow** wizard



ItemListActivity is the main activity. This activity has different presentations for handset and tablet-size devices.

- **On handsets**, the activity first presents a list of items (**ItemListFragment**), when one row is touched it switches to a second screen **ItemDetailActivity** detailing the chosen item.
- **On tablets**, the activity presents two vertical side-by-side panes; one shows a list and the other details for the chosen row.

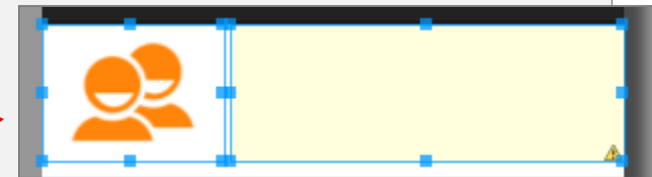
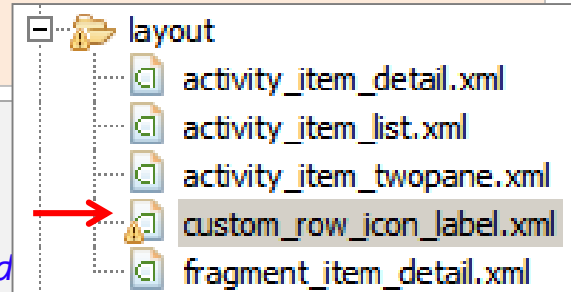
LAYOUT: custom_row_icon_label.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >

    <ImageView
        android:id="@+id/icon"
        android:layout_width="100dp"
        android:layout_height="75dp"
        android:layout_marginRight="3dp"
        android:src="@drawable/ic_launcher"
        android:contentDescription="@string/app_name" />

    <TextView
        android:id="@+id/label"
        android:layout_width="match_parent"
        android:layout_height="75dp"
        android:background="#22ffff00"
        android:textSize="20sp" />

</LinearLayout>
```

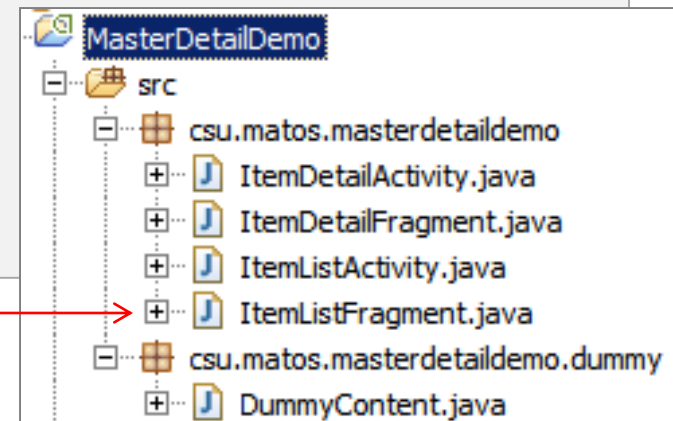
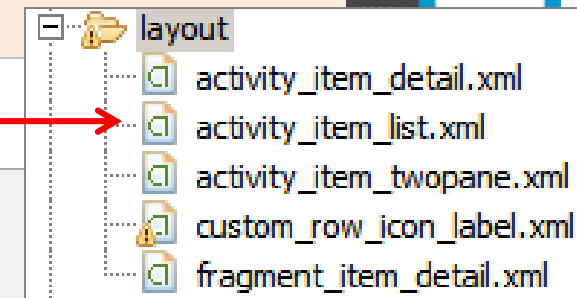


This format was already discussed in Lesson5–Example 4. It defines a row hosting [thumbnail, text] containers. These rows will be used to populate the master list. 6 - 63

LAYOUT: activity_item_list.xml

<!-- activity_item_list.xml / Single Pane Master -->

```
<fragment
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/item_list"
    android:name="csu.matos.masterdetaildemo.ItemListFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    tools:context=".ItemListActivity"
    tools:layout="@android:layout/list_content" />
```



LAYOUT: activity_item_detail.xml

```
<!-- activity_item_detail.xml -->
```

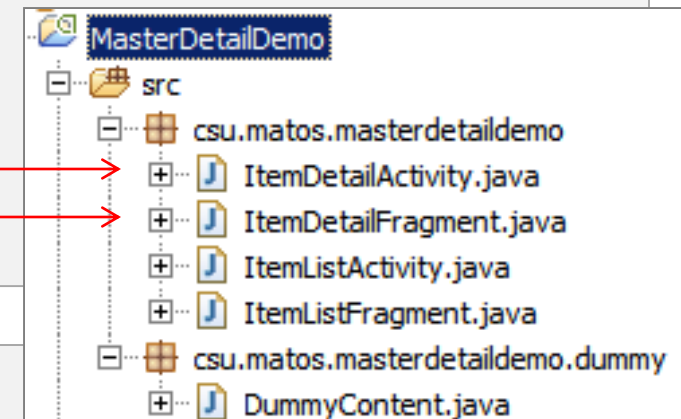
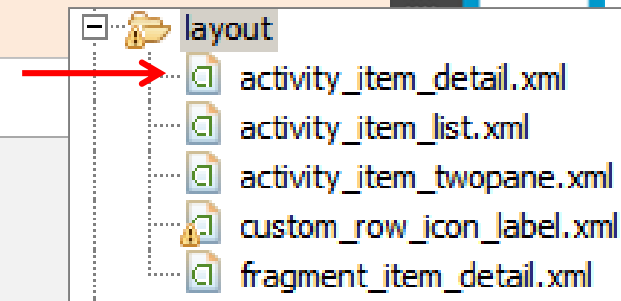
```
<FrameLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/item_detail_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ItemDetailActivity"
    tools:ignore="MergeRootFrame" />
```

```
<!-- fragment_item_detail.xml -->
```

```
<TextView
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/item_detail"
    style="?android:attr/textAppearanceLarge"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:textIsSelectable="true"
    tools:context=".ItemDetailFragment" />
```



LAYOUT: activity_item_twopane.xml

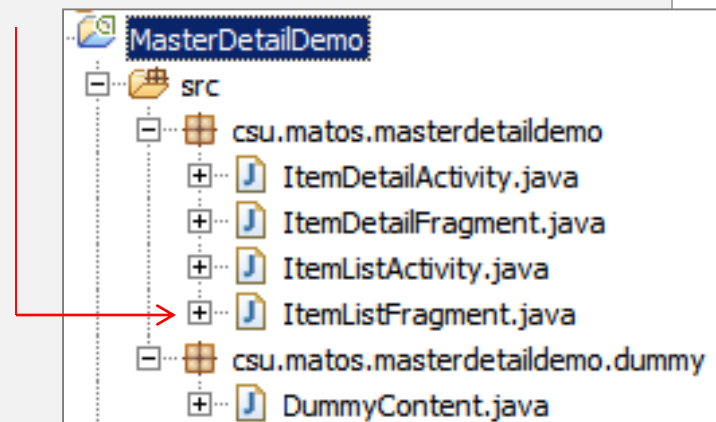
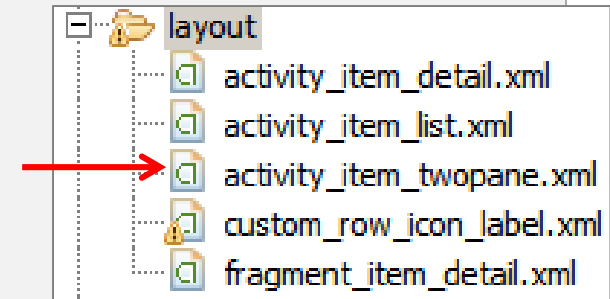


```
<!-- activity_item_twopane.xml / Double Pane: Master & Detail -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    android:baselineAligned="false"
    android:divider="?android:attr/dividerHorizontal"
    android:orientation="horizontal"
    android:showDividers="middle"
    tools:context=".ItemListActivity" >

    <fragment
        android:id="@+id/item_list"
        android:name="csu.matos.masterdetaildemo.ItemListFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        tools:layout="@android:layout/list_content" />

    <FrameLayout
        android:id="@+id/item_detail_container"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3" />

</LinearLayout>
```





```
/**
 * An activity representing a list of Items...
 * This is the MAIN activity. Please notice that
 * No changes were made to the original logic, however
 * GENERATED COMMENTS HAVE BEEN DELETED
 */

public class ItemListActivity extends Activity
    implements ItemListFragment.Callbacks {

    private boolean mTwoPane;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_item_list);

        if (findViewById(R.id.item_detail_container) != null) {
            mTwoPane = true;
            ((ItemListFragment) getFragmentManager().findFragmentById(
                R.id.item_list)).setActivateOnItemClick(true);
        }
    }
}
```



```
@Override
public void onItemSelected(String id) {
    if (mTwoPane) {
        Bundle arguments = new Bundle();
        arguments.putString(ItemDetailFragment.ARG_ITEM_ID, id);

        ItemDetailFragment fragment = new ItemDetailFragment();
        fragment.setArguments(arguments);

        getFragmentManager()
            .beginTransaction()
            .replace(R.id.item_detail_container, fragment)
            .commit();

    } else {
        Intent detailIntent = new Intent(this, ItemDetailActivity.class);
        detailIntent.putExtra(ItemDetailFragment.ARG_ITEM_ID, id);
        startActivity(detailIntent);
    }
}
```



```
/**
 * An activity representing a single Item detail screen...
 * NO CHANGES MADE TO ORIGINAL LOGIC - GENERATED COMMENTS DELETED
 */
public class ItemDetailActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_item_detail);

        getActionBar().setDisplayHomeAsUpEnabled(true);

        if (savedInstanceState == null) {
            Bundle arguments = new Bundle();
            arguments.putString( ItemDetailFragment.ARG_ITEM_ID,
                getIntent().getStringExtra(ItemDetailFragment.ARG_ITEM_ID));

            ItemDetailFragment fragment = new ItemDetailFragment();
            fragment.setArguments(arguments);
            getFragmentManager()
                .beginTransaction()
                .add(R.id.item_detail_container, fragment)
                .commit();
        }
    }
}
```



```
@Override
public boolean onOptionsItemSelected(MenuItem item) {

    int id = item.getItemId();

    if (id == android.R.id.home) {
        NavUtils.navigateUpTo(this,
                               new Intent(this, ItemListActivity.class) );
        return true;
    }
    return super.onOptionsItemSelected(item);
}
}
```



```
/**
 * A list fragment representing a list of ...
 * OUR CODE LOGIC GOES HERE - ORIGINAL COMMENTS REMOVED...
 */
public class ItemListFragment extends ListFragment {

    private static final String STATE_ACTIVATED_POSITION = "activated_position";
    private Callbacks mCallbacks = sDummyCallbacks;
    private int mActivatedPosition = ListView.INVALID_POSITION;

    public interface Callbacks {
        public void onItemSelected(String id);
    }

    private static Callbacks sDummyCallbacks = new Callbacks() {
        @Override
        public void onItemSelected(String id) {
        }
    };

    public ItemListFragment() {
    }
```



NEW



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

    // Using our CUSTOM ADAPTER. Each line has a thumbnail & text.
    // Adapter's args are: contex, layout-to-be-inflated, row-text, icons
    CustomIconLabelAdapter adapter = new CustomIconLabelAdapter(
        getActivity(),
        R.layout.custom_row_icon_label,
        DummyContent.items,
        DummyContent.thumbnails);

    // bind intrinsic ListView to custom adapter
    setListAdapter(adapter);
}
```

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
    if (savedInstanceState != null
        && savedInstanceState.containsKey(STATE_ACTIVATED_POSITION)) {
        setActivatedPosition(savedInstanceState
            .getInt(STATE_ACTIVATED_POSITION));
    }
}
```




```
@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    if (!(activity instanceof Callbacks)) {
        throw new IllegalStateException(
            "Activity must implement fragment's callbacks.");
    }
    mCallbacks = (Callbacks) activity;
}

@Override
public void onDetach() {
    super.onDetach();
    mCallbacks = sDummyCallbacks;
}

@Override
public void onListItemClick(ListView listView, View view, int position,
    long id) {
    super.onListItemClick(listView, view, position, id);
    // Notify (via callback) that an item has been selected,
    // indicate the row-position of the selected item (0,1,2...)
    mCallbacks.onItemSelected("" + position);
}
```

NEW



```
@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    if (mActivatedPosition != ListView.INVALID_POSITION) {
        outState.putInt(STATE_ACTIVATED_POSITION, mActivatedPosition);
    }
}

public void setActivateOnItemClick(boolean activateOnItemClick) {
    getListView().setChoiceMode(
        activateOnItemClick ? ListView.CHOICE_MODE_SINGLE
        : ListView.CHOICE_MODE_NONE);
}

private void setActivatedPosition(int position) {
    if (position == ListView.INVALID_POSITION) {
        getListView().setItemChecked(mActivatedPosition, false);
    } else {
        getListView().setItemChecked(position, true);
    }

    mActivatedPosition = position;
}
}
```



```
/**
 * A fragment representing a single Picture detail screen...
 * OUR CODE LOGIC GOES HERE - GENERATED COMMENTS REMOVED.
 */
public class ItemDetailFragment extends Fragment {

    public static final String ARG_ITEM_ID = "item_id";
    int intSelectedRow = 0;

    public ItemDetailFragment() {

    }

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

        if (getArguments().containsKey(ARG_ITEM_ID)) {
            // obtain position of selected row (it comes from the ListFragment
            // bundled as string argument ARG_ITEM_ID)

            String strSelectedRowId = getArguments().getString(ARG_ITEM_ID);
            intSelectedRow = Integer.valueOf(strSelectedRowId);

        }
    }
}
```

NEW



NEW

```
@Override
public View onCreateView(LayoutInflater inflater,
                        ViewGroup container, Bundle savedInstanceState) {

    // create an UI showing a caption on top and a large image
    LinearLayout linearlayout = new LinearLayout(getActivity());
    linearlayout.setOrientation(LinearLayout.VERTICAL);

    TextView txtMsg2 = new TextView(getActivity());
    txtMsg2.setBackgroundColor(0xffffff00); //yellow back
    txtMsg2.setTextSize(25);
    txtMsg2.setText("Selected Image: " + intSelectedRow + " "
                    + DummyContent.items[intSelectedRow]);

    ImageView image = new ImageView(getActivity());
    image.setLayoutParams(new RelativeLayout.LayoutParams(
        LayoutParams.MATCH_PARENT,
        LayoutParams.MATCH_PARENT));
    image.setBackgroundResource( DummyContent.LargeImages[intSelectedRow] );

    linearlayout.addView(txtMsg2);
    linearlayout.addView(image);

    return linearlayout;
}
```



```
// this custom Adapter inflates rows each holding a  
// thumbnail & text side-by-side [image | text]  
// the ViewHolder pattern is used to enhance performance
```

```
class CustomIconLabelAdapter extends ArrayAdapter<String> {  
    Context context;  
    Integer[] thumbnails;  
    String[] items;  
  
    public CustomIconLabelAdapter(Context context, int layoutToBeInflated,  
        String[] items, Integer[] thumbnails) {  
        super(context, layoutToBeInflated, items);  
        this.context = context;  
        this.thumbnails = thumbnails;  
        this.items = items;  
    }  
  
    public View getView(int position, View convertView, ViewGroup parent) {  
        ViewHolder viewholder;  
        // Use ViewHolder to keeps references to children views already made  
        // this pattern reduces the amount of unneeded calls to findViewById()  
        if (convertView == null) {  
            LayoutInflater inflater = ((Activity) context).getLayoutInflater();  
            convertView = inflater.inflate(R.layout.custom_row_icon_label, null);  
  
            viewholder = new ViewHolder();  
            viewholder.label = (TextView) convertView.findViewById(R.id.label);  
            viewholder.icon = (ImageView) convertView.findViewById(R.id.icon);  
            convertView.setTag(viewholder);  
        }  
    }  
}
```

NEW



NEW



```
    } else {
        // reuse existing viewholder
        viewholder = (ViewHolder) convertView.getTag();
    }

    // move data to the viewholder.
    viewholder.label.setText(DummyContent.items[position]);
    viewholder.icon.setImageResource(DummyContent.thumbnails[position]);

    return convertView;
}

// -----
static class ViewHolder {
    TextView label;
    ImageView icon;
}

} // IconTextAdapter
```

This custom `DataAdapter` is similar to the version introduced in Example4 (Lesson5), however the **ViewHolder** pattern is here used to enhance the inflating of custom rows. Each row holds an icon and a line of text (See Appendix B for an example of a custom `BaseAdapter` using **ViewHolder** design strategy)

DATA: DummyContent.java



```
/* *****  
 * Helper class acting as 'fake' Data Provider - COMMENTS REMOVED...  
 * ***** */  
  
public class DummyContent {  
    // text to be shown on each list-line  
    public static String[] items = { "Picture 1", "Picture 2", "Picture 3", "Picture 4",  
        "Picture 5", "Picture 6", "Picture 7", "Picture 8", "Picture 9", "Picture 10",  
        "Picture 11", "Picture 12", "Picture 13", "Picture 14", "Picture 15" };  
    // small image to accompany text on each list-line  
    public static Integer[] thumbnails = {  
        R.drawable.pic01_small, R.drawable.pic02_small,  
        R.drawable.pic03_small, R.drawable.pic04_small,  
        R.drawable.pic05_small, R.drawable.pic06_small,  
        R.drawable.pic07_small, R.drawable.pic08_small,  
        R.drawable.pic09_small, R.drawable.pic10_small,  
        R.drawable.pic11_small, R.drawable.pic12_small,  
        R.drawable.pic13_small, R.drawable.pic14_small,  
        R.drawable.pic15_small };  
  
    // high-resolution version of thumbnail, to be shown as detail  
    public static Integer[] largeImages = {  
        R.drawable.pic01_large, R.drawable.pic02_large, R.drawable.pic03_large,  
        R.drawable.pic04_large, R.drawable.pic05_large, R.drawable.pic06_large,  
        R.drawable.pic07_large, R.drawable.pic08_large, R.drawable.pic09_large,  
        R.drawable.pic10_large, R.drawable.pic11_large, R.drawable.pic12_large,  
        R.drawable.pic13_large, R.drawable.pic14_large, R.drawable.pic15_large };  
  
    static { //nothing to do, for now... }  
}
```

NEW