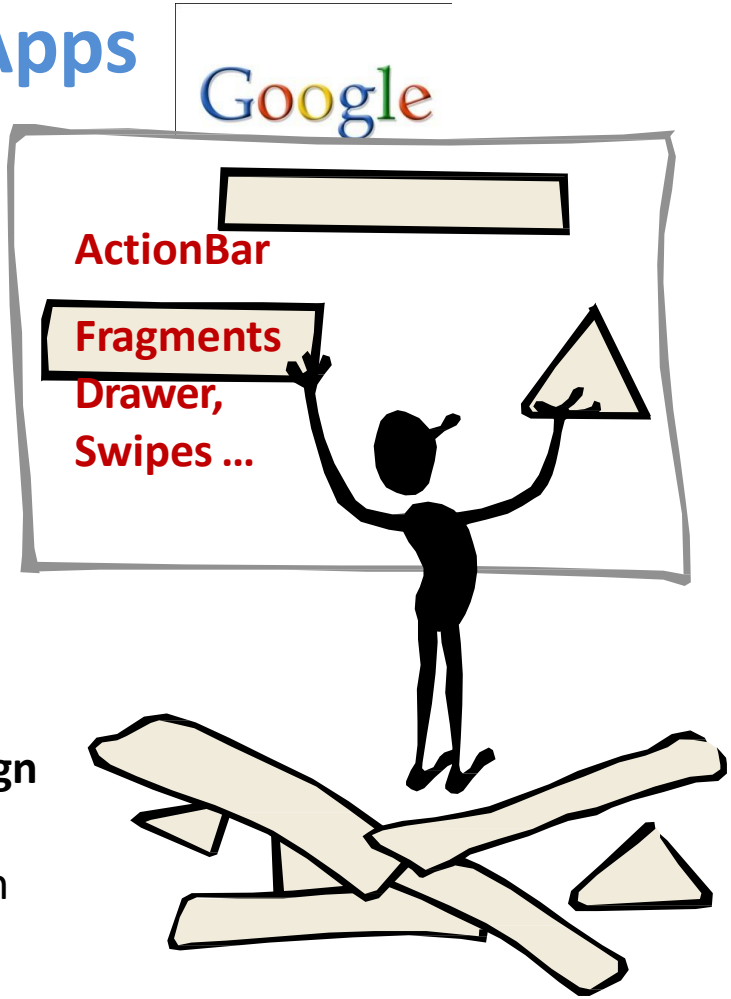
Programming mobile devices

Usage of fragments

Adaptation of materials: dr Tomasz Xięski.
Based on presentations made available by Victor Matos, Cleveland State University.

Portions of this page are reproduced from work created and shared by Google and used according to terms

Designing Complex Android Apps



- A major goal of the Android platform is to consistently offer a pleasant, rich, intuitive, and homogeneous user-experience.
- This is a challenge for the designers. Your apps need to provide a sense of sameness and familiarity as well as a touch of their 'unique personality'.
- A set of recommendations called **Material Design** is suggested to adopters of SDK5+ as a way to create apps within the boundaries of a common modeling framework.
- By adopting those suggestions the "look-and-feel" of all new apps is expected to become more uniform and its navigation more predictable.
- In this lesson we will explore some of the building blocks used to implement this design vision

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

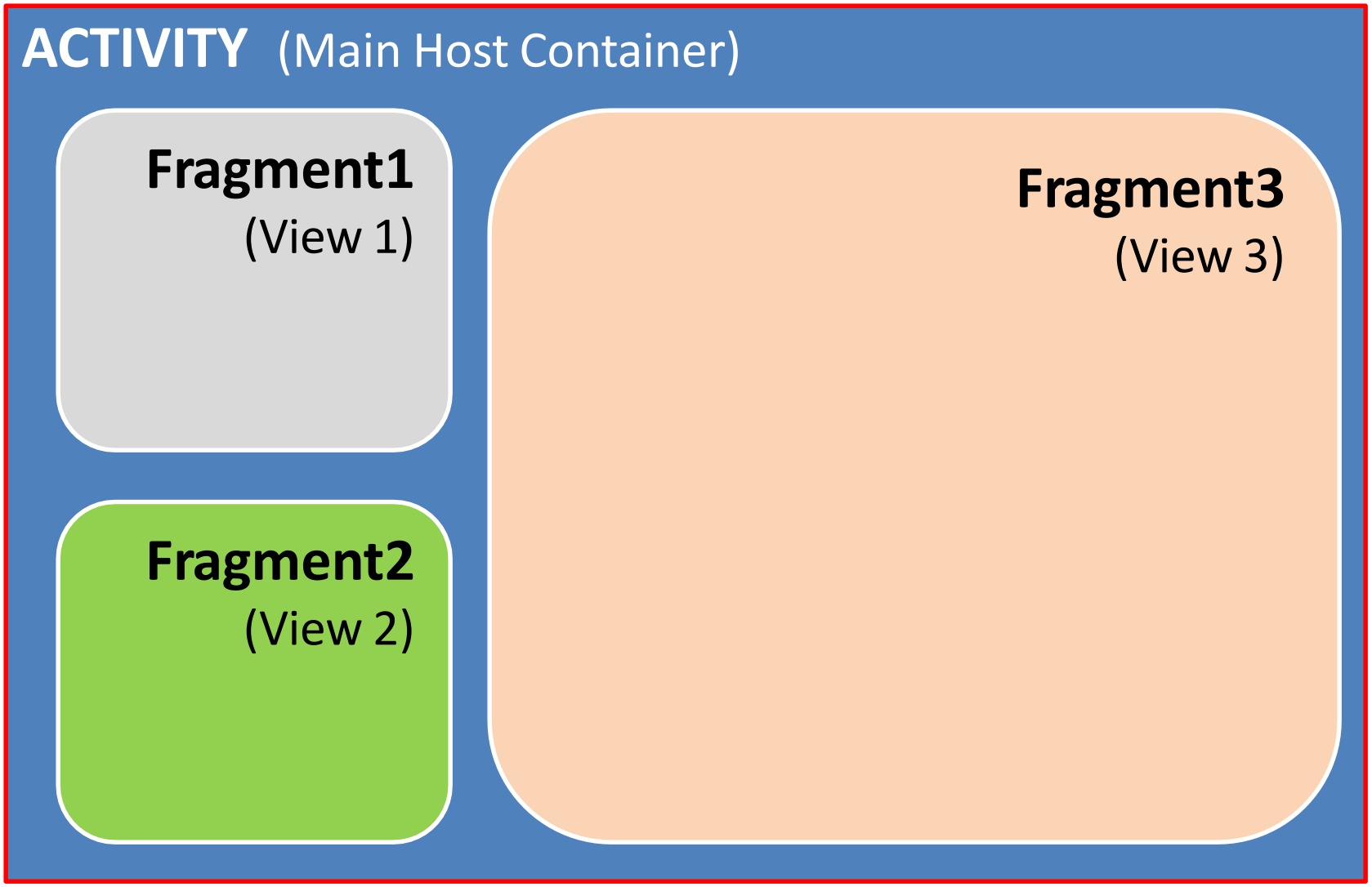
- Fragments behave as independent threads, usually they cooperate in achieving a common goal; however each can run its own I/O, events and business logic.

- 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).ch

Fragments



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

Fragment's Lifecycle

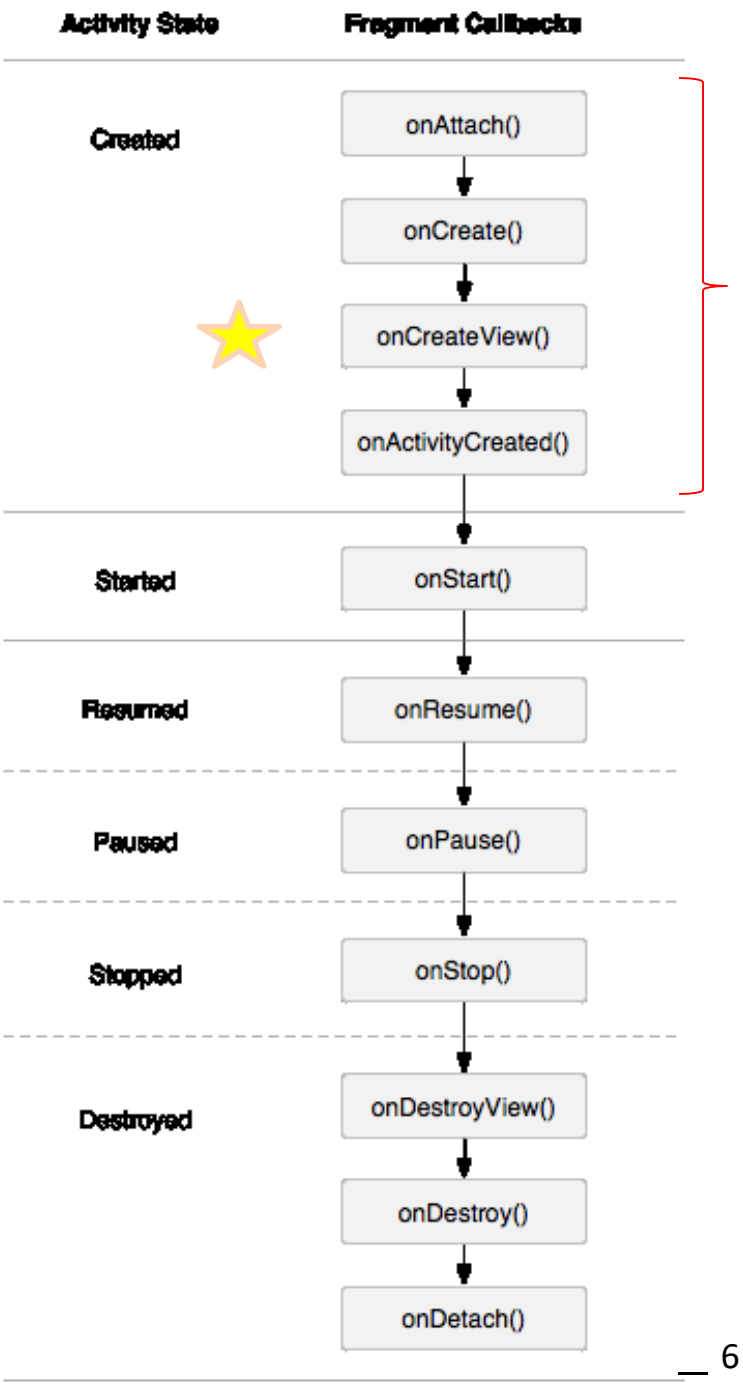
onAttach() Invoked when the fragment has been connected to the host activity.

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.

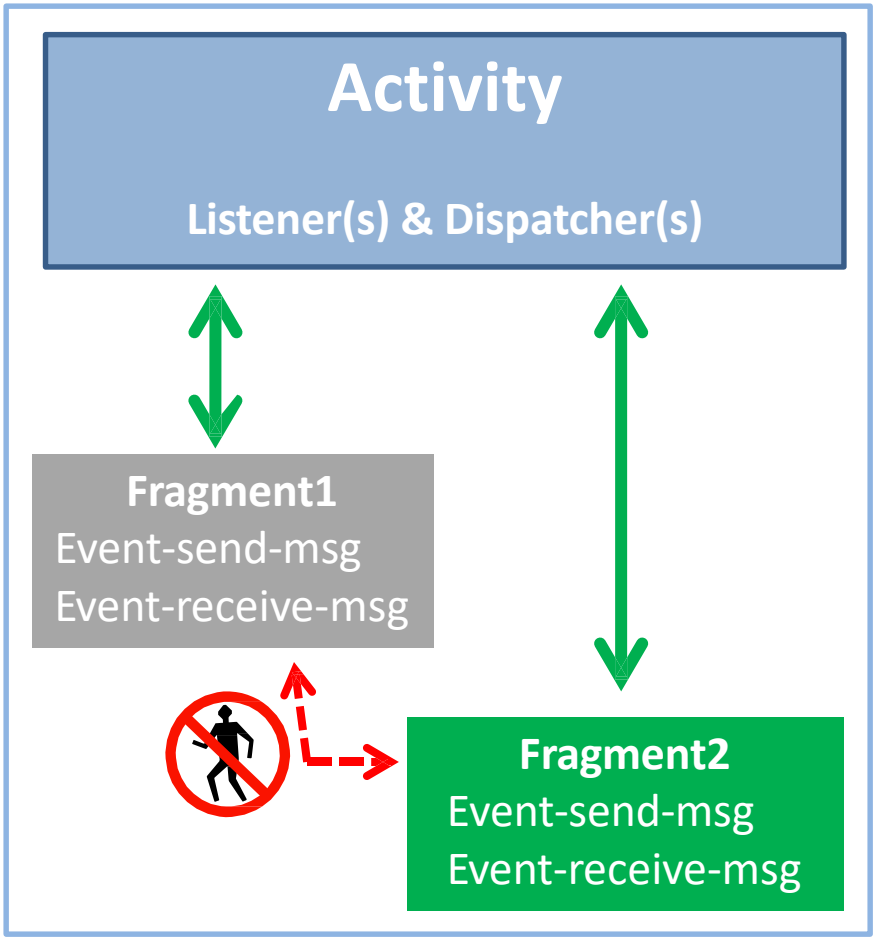
onDetach() Called when the inactive fragment is disconnected from the activity.



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.

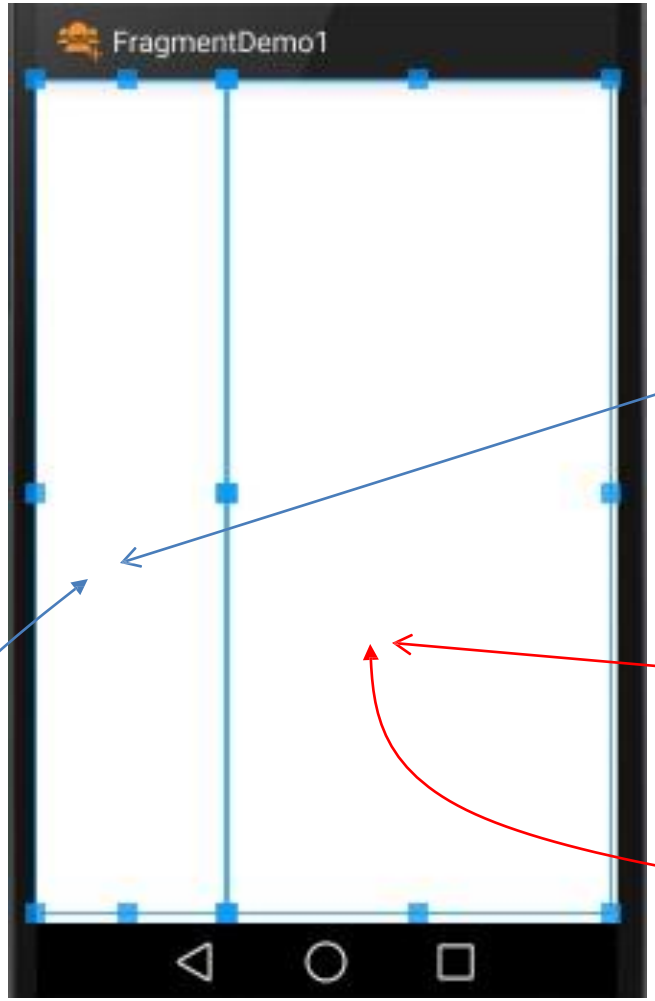
```
ft.replace(R.id.main_holder_blue, blueFragment);
```

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.

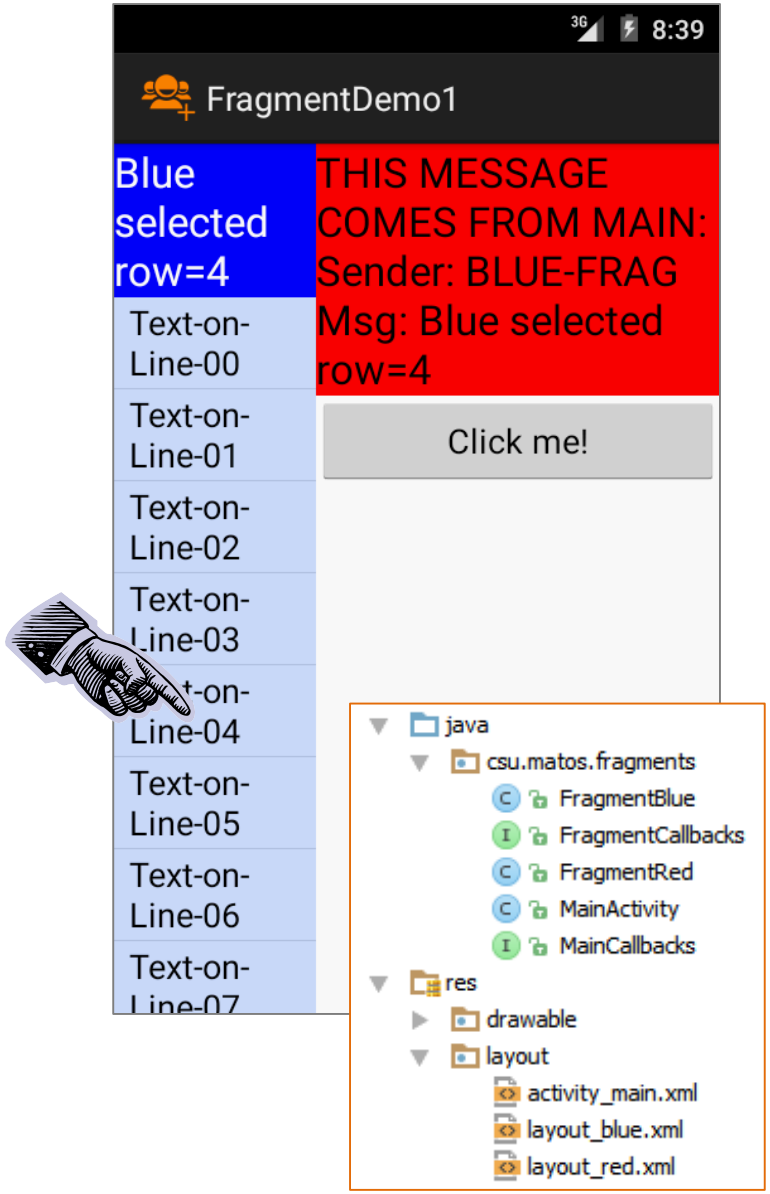
```
// create a new BLUE fragment - show it
ft = getFragmentManager().beginTransaction();
blueFragment = FragmentBlue.newInstance("new-blue");
ft.replace(R.id.main_holder_blue, blueFragment);
ft.commit();
```

```
// 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),
- **FragmentRed** (master) and
- **FragmentBlue** (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).

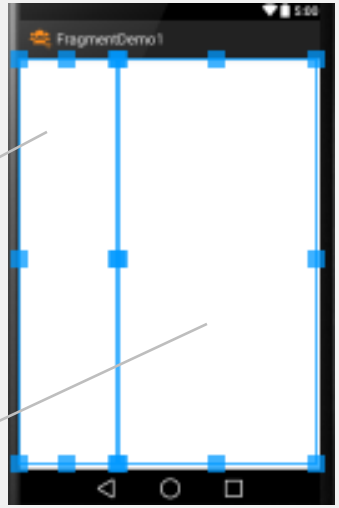
The **detail** fragment echoes the value of the selected row.

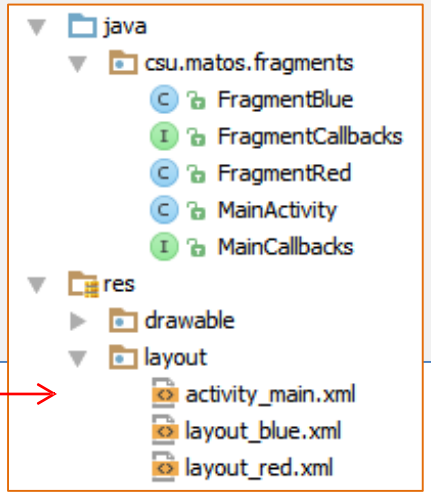
The interfaces **FragmentCallbacks** and **MainCallbacks** define the methods used to pass messages from the MainActivity to fragments and from fragments to MainActivity respectively.

Example 1 – Dynamic Activity-Fragment Binding

XML LAYOUT: activity_main.xml

```xml
<?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>
```

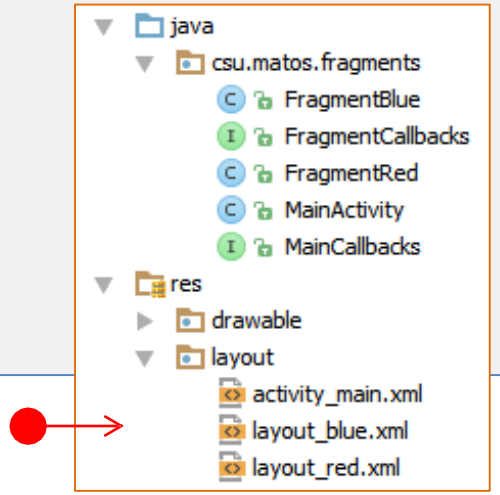




Example 1 – Dynamic Activity-Fragment Binding

XML LAYOUT: layout_blue.xml

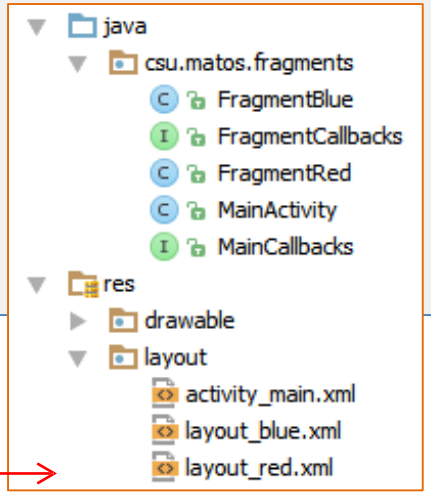
```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout_blue"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

      <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
<?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>
```



Example 1 – Dynamic Binding - MainActivity


```java
//-----------------------------------------------------------------------
// 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();

   }
```

Example 1 – Dynamic Binding - MainActivity


```
    // MainCallback implementation (receiving messages coming from Fragments)
    @Override
    public void onMsgFromFragToMain(String sender, String strValue) {
        // show message arriving to MainActivity
        Toast.makeText(getApplication(),
                " 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**.

Example 1 – Dynamic Binding - FragmentBlue


```java
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");
        }
    }
```

Example 1 – Dynamic Binding - FragmentBlue


```
    @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 trough 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`**.

Example 1 – Dynamic Binding - FragmentRed


```
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);
```

Example 1 – Dynamic Binding - FragmentRed


```
        // 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 snipped 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 mehods 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.

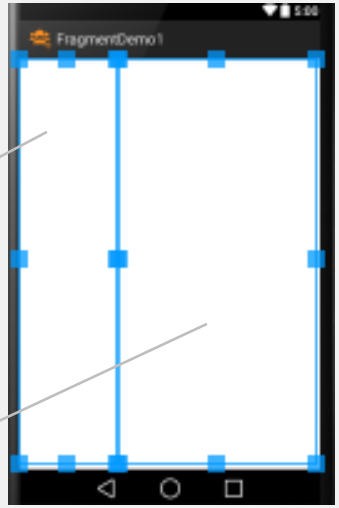
- The following attributes can be used to identify the fragment in case it needs to be restarted (if none is provided the fragment is identified by the system's id of the fragment's container id)

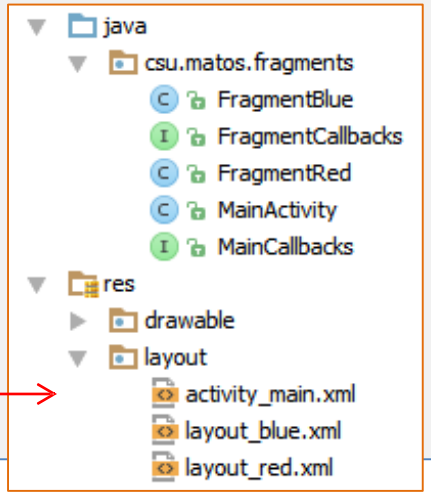
```
1.  android:name="AppPackageName.FragmentClassName"
2.  android:id="@id+/uniqueName" or
    android:tag="string"
```

Example 2 – Static Activity-Fragment Binding

XML LAYOUT: activity_main.xml

```xml
<?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>
```





Example 2 – Dynamic Binding - MainActivity


```
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;

      }

   }
```

Example 2 – Dynamic Binding - MainActivity


```
   @Override
   public void onMsgFromFragToMain(String sender, String strValue) {
      Toast.makeText(getApplication(), " 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

}
```

Example 2 – Static Binding - MainActivity

COMMENTS

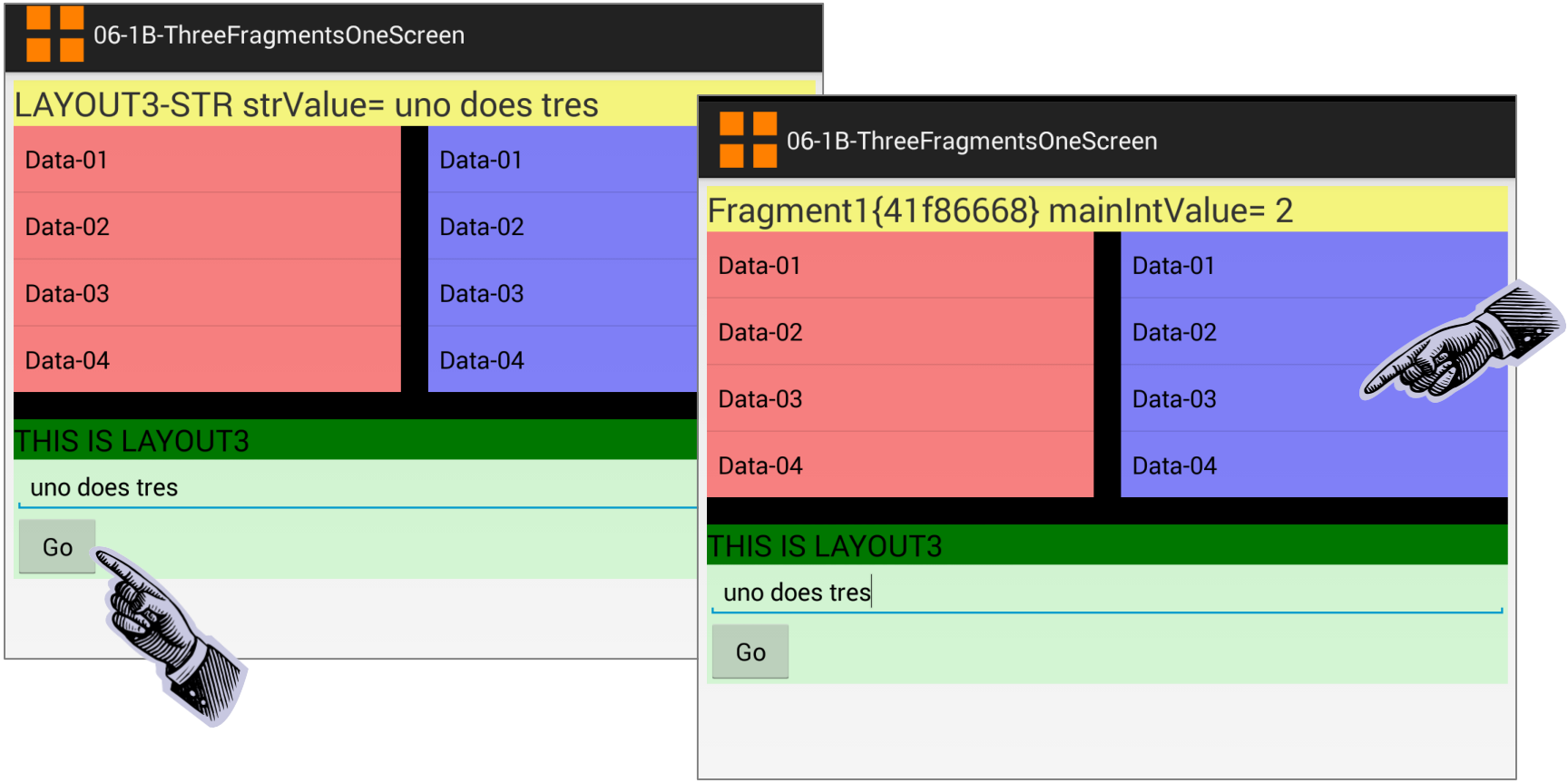
1. In this example the **onCreate** method has nothing to do. Moreover, **onCreateView** is not even called, observe that the XML-layout clause
   `android:name="csu.matos.fragments.FragmentXYZ"`
   defines the specific fragment to be plugged in the activity's screen.

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.



Example 3 – Multiple-Fragments-One-Screen

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.

Example 3 – Multiple-Fragments-One-Screen

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 3 – Multiple-Fragments-One-Screen

```
public class MainActivity extends Activity implements MainCallbacks {

    FragmentTransaction ft;
    FragmentRed redFragment;
    FragmentBlue blueFragment;
    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("blue");
        ft.replace(R.id.home1, blueFragment);  ft.commit();

        // create a new BLUE fragment - show it
        ft = getFragmentManager().beginTransaction();
        blueFragment2 = FragmentBlue.newInstance("red");
        ft.replace(R.id.home2, blueFragment2);  ft.commit();

        // create a new RED fragment - show it
        ft = getFragmentManager().beginTransaction();
        redFragment = FragmentRed.newInstance("green");
        ft.replace(R.id.home3, redFragment);  ft.commit();

    }
```

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);
```

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 **FragmentTransaction** 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 replace 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

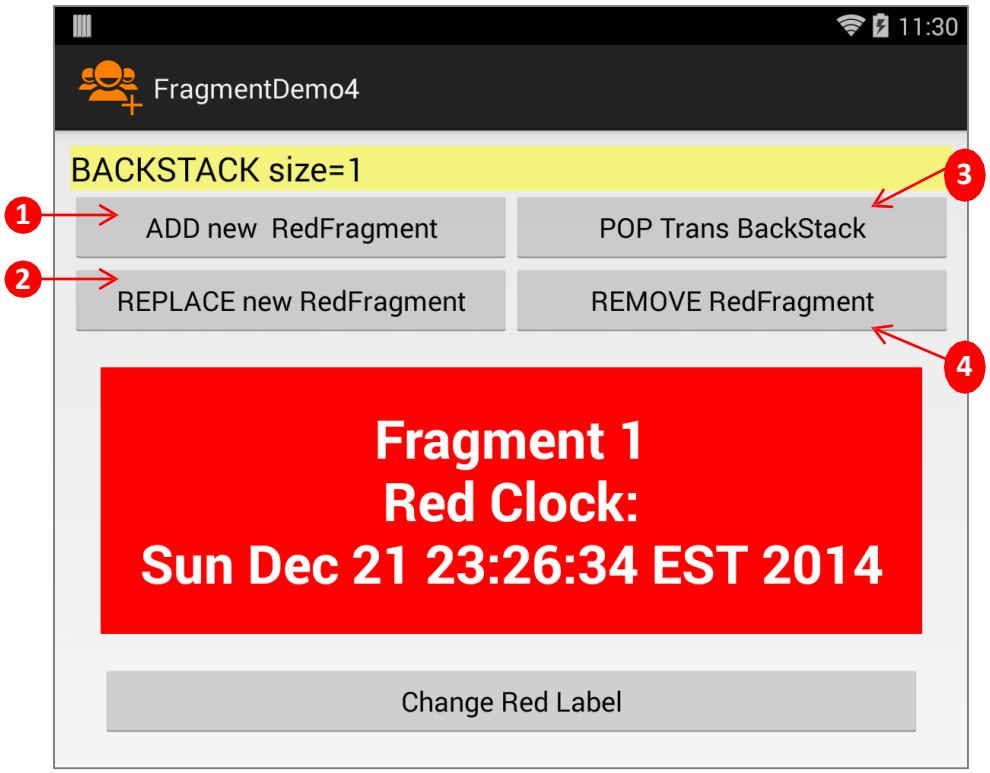
Navigating Through the BackStack

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. 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.

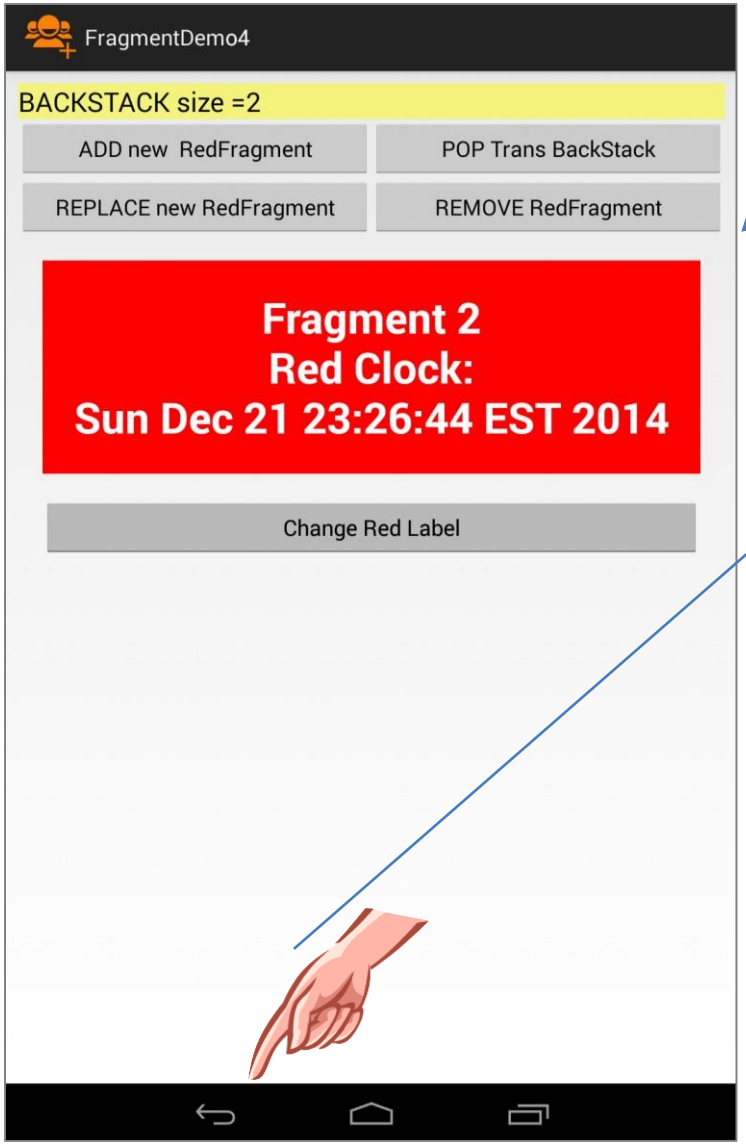
2. As above, however; the fragment's view is attached to the activity's UI using the **replace()** method (old view is destroyed). The current transaction is also added to the BackStack.

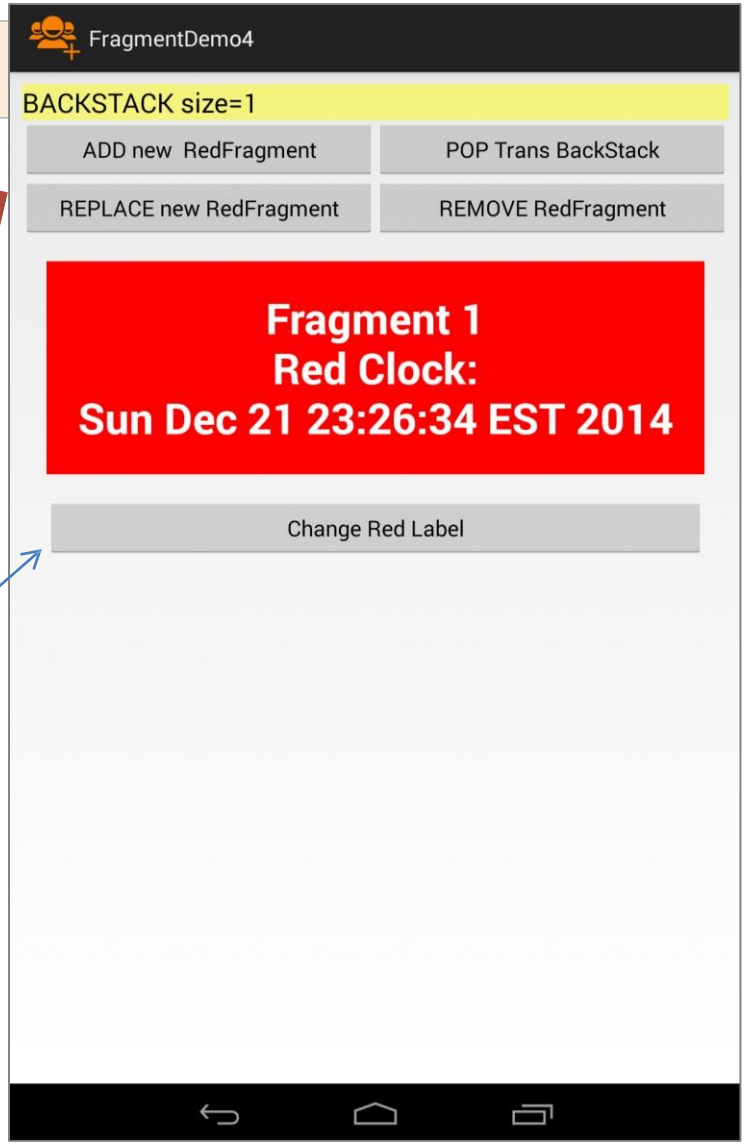
3. Popping an entry from the BackStack removes the current app's UI and navigates back to the previously stored fragment's view. State data (if any) is shown as it was before leaving the view. The size of BackStack is reduced by one.

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.

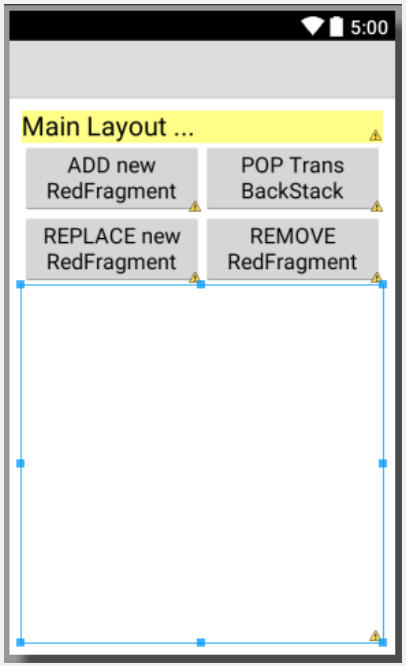




2. Pressing the **Back** button removes the current fragment from the UI and Back-Navigates to the previous fragment. Its state is preserved, so you do not need to refill its widgets.

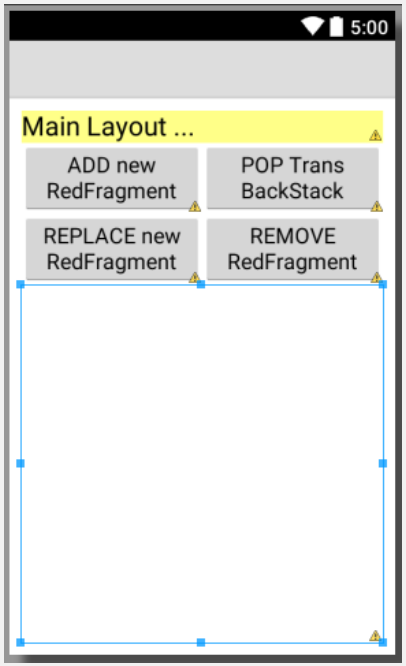
Example 5. LAYOUT: activity_main.xml


```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="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" />
```



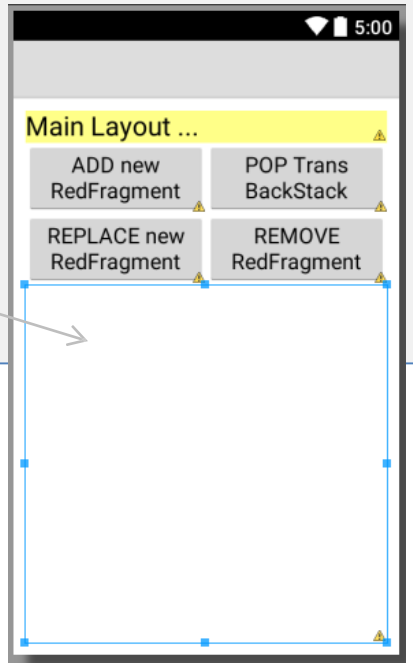
Example 5. LAYOUT: activity_main.xml

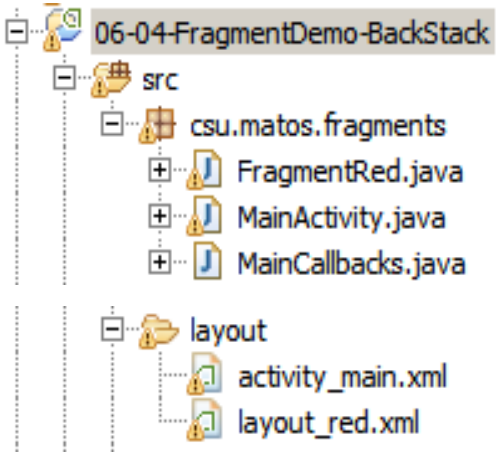

```
    <Button
          android:id="@+id/button2MainPop"
          android:layout_width="150dp"
          android:layout_height="wrap_content"
          android:layout_weight="1"
          android:text="POP Trans BackStack" />
</LinearLayout>

<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


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

</LinearLayout>
```



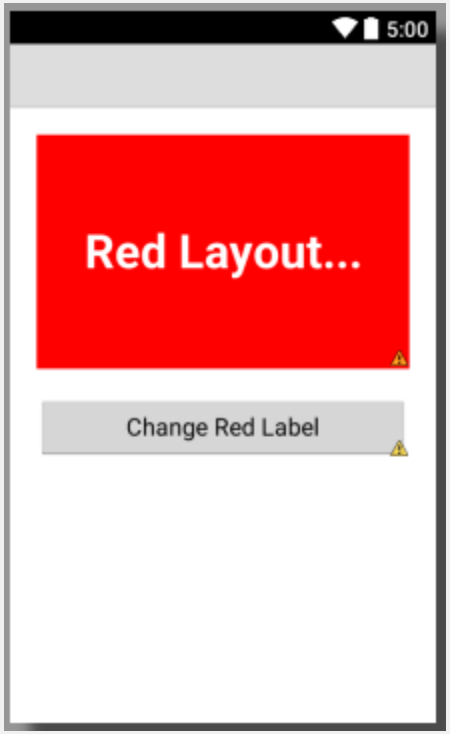


MainCallbacks

```
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


```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="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


```
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


```java
// 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
```

Example 5. MainActivity

```
    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


```
  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


```java
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


```
    @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 ?