SESSION #4

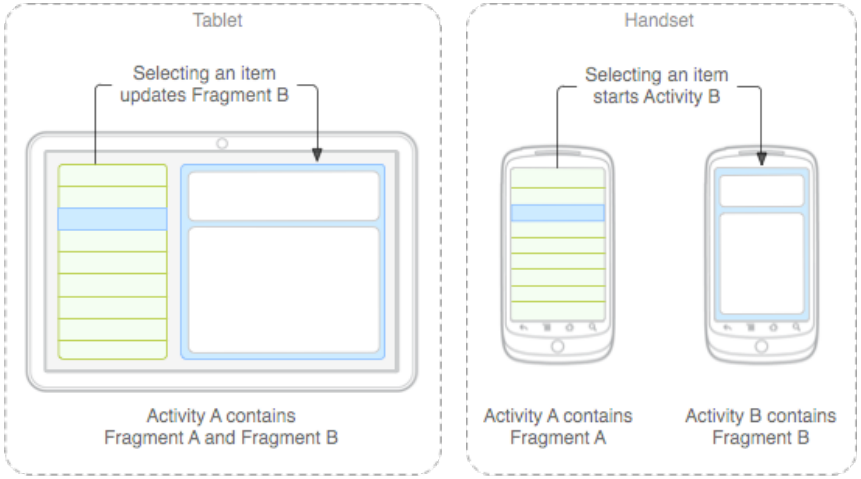
GUI (III)

OUTLINE

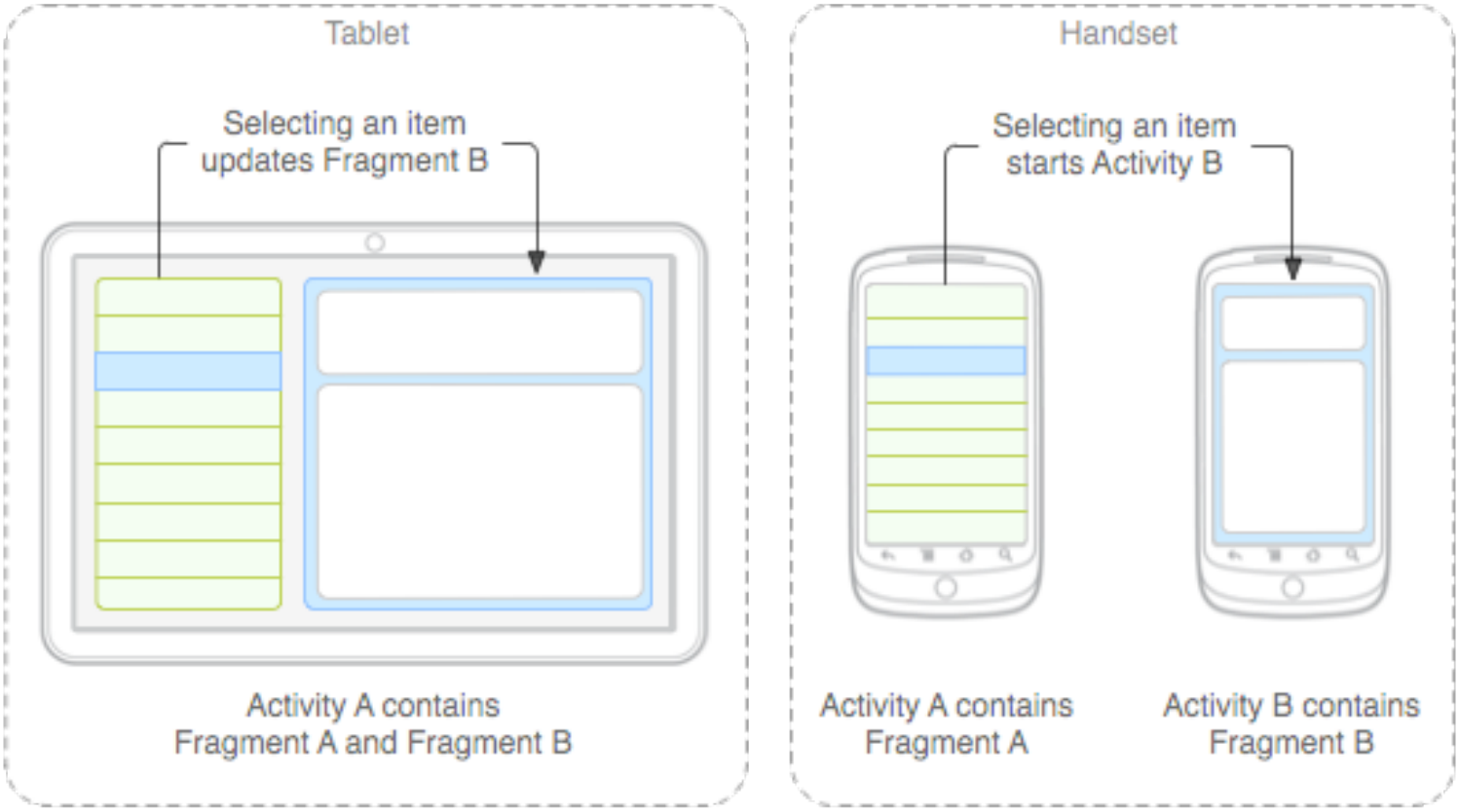
- Fragments
- Support library
- The App Bar
- Settings
- Dialogs
- Styles and themes

FRAGMENTS

- **Fragments** are an optional layer you can put between your activities and your widgets, designed to help you reconfigure your activities to support both large (e.g., tablets) and small (e.g., phones) screens.
- A **Fragment** represents a behavior or a portion of user interface in an **Activity**. You can combine multiple fragments in a single activity to build a **multi-pane UI** and **reuse** a fragment in multiple activities.
  - A fragment can be viewed as a sub-activity, that has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running.
  - A fragment must always be embedded in an activity and the fragment's lifecycle is directly affected by the host activity's lifecycle.



FRAGMENTS


Tablet
Selecting an item
updates Fragment B
Activity A contains
Fragment A and Fragment B
Handset
Selecting an item
starts Activity B
Activity A contains
Fragment A
Activity B contains
Fragment B

FRAGMENTS

ADVANTAGES

- **Capability to add fragments dynamically based on user interaction.**
- **Capability to animate dynamic fragments as they move on and off the screen.**
- **Automatic *Back button* management.**
  - None managed by developers, simply adding the dynamic fragment via a **FragmentTransaction** allows Android to automatically handle the Back button, including reversing all animations.
- **Capability to add options to the options menu, and therefore to the action bar:**
  - Call *setHasOptionsMenu*() in *onCreate*() of your fragment to register an interest in this, and then override *onCreateOptionsMenu*() and *onOptionsItemSelected*() in the fragment.
- **Capability to add tabs to the action bar:**
  - The action bar can have tabs, replacing a TabHost, where each tab's content is a fragment.

CREATION

- To create a fragment, you must create a subclass of **Fragment** (or an existing subclass of it).
  - The **Fragment** class has code that looks a lot like an Activity.
- You need to provide the layout of the fragment to be attached to the activity view hierarchy
  - To provide a layout for a fragment, you must implement the *onCreateView*() callback method, which the Android system calls when it's time for the fragment to draw its layout.
  - Your implementation of this method must return a View that is the root of your fragment's layout.
- To create the fragment layout you can inflate it from a layout resource defined in XML, using the *inflate*() method from the LayoutInflater object provide by *onCreateView*() callback.

FRAGMENTS

CREATION

The LayoutInflater object that can be used to inflate any views in the fragment

If non-null, it provides data about the previous instance of the fragment, if the fragment is being resumed

If non-null, this is the parent view that the fragment's UI should be attached to

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

The inflate(int resource, ViewGroup root, boolean attachToRoot) method takes 3 arguments:

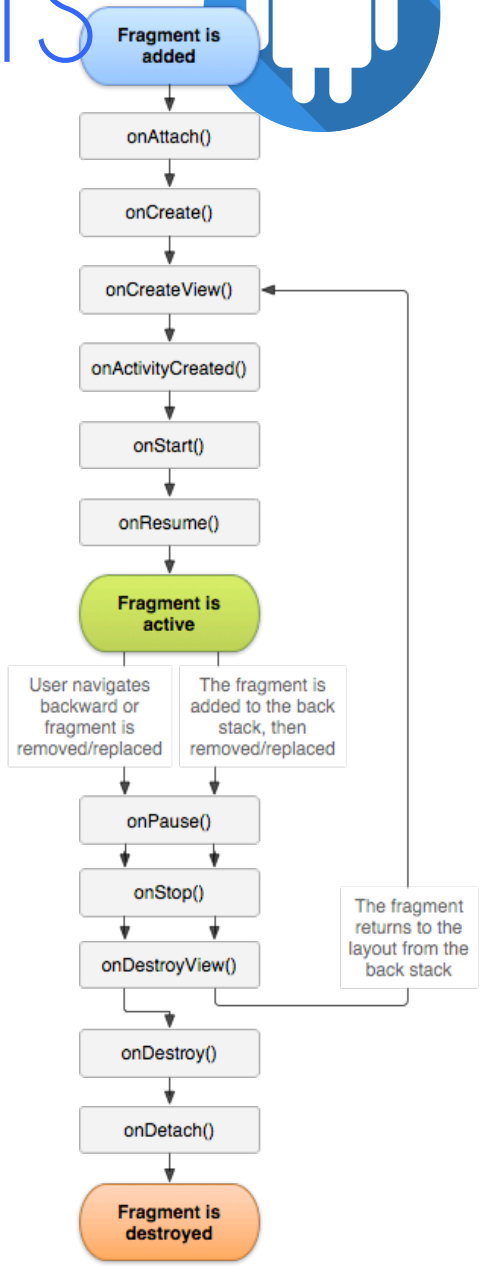
- The resource ID of the layout you want to inflate (res/layout/example_fragment.xml).
- The ViewGroup to be the parent of the inflated layout.
- A boolean indicating whether the inflated layout should be attached to the ViewGroup during inflation. For fragments, this is false because the system is already inserting the inflated layout into the container.

FRAGMENTS

LIFECYCLE

The Fragment contains callback methods similar to an activity.

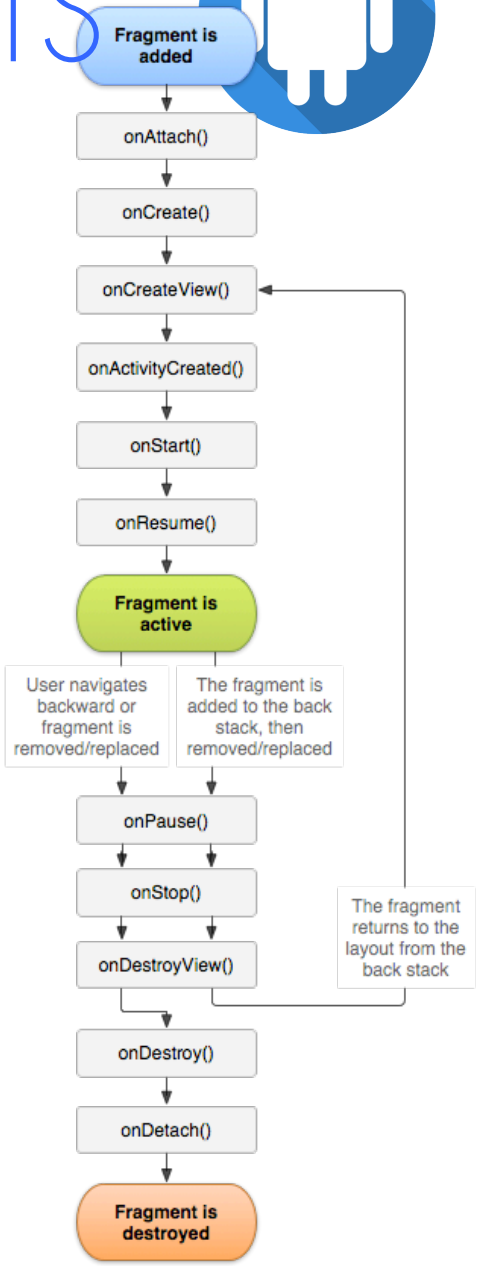
- **onAttach(Activity** activity**)**: is invoked after your fragment is associated with its activity. The activity reference is passed to you if you want to use it.
- **onCreate(Bundle** savedInstanceState**)**: is invoked when creating the fragment. You should initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.
- **onCreateView(LayoutInflater** inflater, **ViewGroup** container, **Bundle** savedInstanceState**)**: is called when it's time for the fragment to draw its user interface for the first time. You must return a View from this method that is the root of your fragment's layout.



FRAGMENTS

LIFECYCLE

- **onActivityCreated (Bundle** savedInstanceState**)**: is called when the fragment's activity has been created and this fragment's view hierarchy instantiated. It can be used to do final initialization once these pieces are in place, such as retrieving views or restoring from a saved state.
- **onStart, onResume, onPause and onStop** call-backs are equivalent to the activity ones.
- **onDestroyView():** is called when the view previously created by onCreateView has been detached from the fragment. The next time the fragment needs to be displayed, a new view will be created.
- **onDestroy():** is invoked when the fragment is no longer in use.
- **onDetach():** is called when the fragment is no longer attached to its activity.



FRAGMENT CLASSES

- **ListFragment**: Wraps a **ListView** in a **Fragment**
  - Designed to simplify setting up lists of things.
  - Similar to a **ListActivity**, all you need to do is call *setListAdapter*() with your chosen and configured **ListAdapter**, plus override *onListItemClick*() to respond to when the user clicks on a row in the list.
- **DialogFragment**: Displays a floating dialog.
  - This class allows to create a dialog fragment that can be incorporated into the back stack of fragments managed by the activity, allowing the user to return to a dismissed fragment.
- **PreferenceFragment**: Displays a hierarchy of **Preference** objects as a list, similar to **PreferenceActivity**.
  - This is useful when creating a "settings" activity for your application.

LIST FRAGMENT

```
public class MainActivity extends Activity {

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

        FragmentManager fm = getFragmentManager();

        if (fm.findFragmentById(android.R.id.content) == null) {
            SimpleListFragment list = new SimpleListFragment();
            fm.beginTransaction().add(android.R.id.content, list).commit();
        }
    }

    public static class SimpleListFragment extends ListFragment
    {
        String[] numbers_text = new String[] { "one", "two", "three", "four", "five", "six", "seven”};
        String[] numbers_digits = new String[] { "1", "2", "3", "4", "5", "6", "7”};

        @Override
        public void onListItemClick(ListView l, View v, int position, long id) {
            new CustomToast(getActivity(), numbers_digits[(int) id]);
        }

        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup container,
                                 Bundle savedInstanceState) {
            ArrayAdapter<String> adapter = new ArrayAdapter<String>(inflater.getContext(),
                                 android.R.layout.simple_list_item_1, numbers_text);
            setListAdapter(adapter);
            return super.onCreateView(inflater, container, savedInstanceState);
        }
    }
}
```

ADDING TO AN ACTIVITY

- Usually, a fragment contributes a portion of UI to the host activity, which is embedded as a part of the activity's overall view hierarchy.
- There are two ways you can add a fragment to the activity layout:
  - **Declare the fragment inside the activity's layout file.**
    - In this case, you can specify layout properties for the fragment as if it were a view.
  - **Programmatically add the fragment to an existing ViewGroup.**
    - At any time while your activity is running, you can add fragments to your activity layout. You simply need to specify a **ViewGroup** in which to place the fragment.
    - To make fragment transactions in your activity (such as add, remove, or replace a fragment), you must use APIs from **FragmentTransaction**.

FRAGMENTS

LAYOUT

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

- The **android:name** attribute in the **<fragment>** specifies the Fragment class to instantiate
- When the system creates this activity layout, it instantiates each fragment specified in the layout and calls the ***onCreateView()*** method for each one, to retrieve each fragment's layout.
- The system inserts the View returned by the fragment directly in place of the **<fragment>** element.

ADDING TO AN ACTIVITY

- At any time while your activity is running, you can add fragments to your activity layout.
  - You simply need to specify a ViewGroup in which to place the fragment.
  - To perform fragment transactions in your activity (such as add, remove, or replace a fragment), you must use APIs from **FragmentTransaction**.

```
FragmentManager fragmentManager =  getFragmentManager()
FragmentTransaction fragmentTransaction = fragmentManager. beginTransaction() ;
```

- You can then add a fragment using the add() method, specifying the fragment to add and the view in which to insert it.

```
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

  - The first argument to ***add()*** is the **ViewGroup** in which the fragment should be placed (resource ID), and the second parameter is the fragment to add.
  - Once made the changes, you must call ***commit()*** for the changes to take effect.

MANAGING (I)

- To manage the fragments in your activity, you need to use **FragmentManager**
  - To get it, call ***getFragmentManager()*** from your activity.
- With **FragmentManager** you can do:
  - Get fragments that exist in the activity, with ***findFragmentById()*** (for fragments that provide a UI in the activity layout) or ***findFragmentByTag()*** (for fragments that do or don't provide a UI).
  - Open a **FragmentTransaction**, which allows to perform transactions using methods such as ***add()***, ***remove()***, and ***replace()***.
    - Inside a transaction call to ***addToBackStack()*** to add the transaction to a back stack of fragment transactions.
      - This back stack allows the user to return to the previous fragment state, by the Back button.
      - For each fragment transaction, you can apply a transition animation, by calling ***setTransition()*** before you commit.
  - Pop fragments off the back stack, with ***popBackStack()*** (simulating a Back command by the user).
  - Register a listener for changes to the back stack, with ***addOnBackStackChangedListener()***.

MANAGING (II)

```
if (findViewById(R.id.fragPage) != null) {
    WebViewFragment wvf = (WebViewFragment) getFragmentManager().findFragmentById(R.id.fragPage);
    if (wvf == null) {
        System.out.println("Dual fragment new");
        wvf = new WebViewFragment();
        // We are in dual fragment (Tablet and so on)
        FragmentManager fm = getFragmentManager();
        FragmentTransaction ft = fm.beginTransaction();
        ft.add(R.id.fragPage, wvf);
        ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
        // Add to backstack
        ft.addToBackStack(linkData.getName());
        ft.commit();
    }
    else {
        Log.d("SwA", "Dual Fragment update");
        wvf = new WebViewFragment();
        FragmentManager fm = getFragmentManager();
        FragmentTransaction ft = fm.beginTransaction();
    ft.replace(R.id.fragPage, wvf);
    ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
        // Add to backstack
        ft.addToBackStack(linkData.getName());
        ft.commit();
    }
```

COMMUNICATING FRAGMENTS

- All fragments attached to a activity or any fragment in that activity can ask for any other fragment using the ***getActivity()***, ***findFragmentById()*** or ***findFragmentByTag()*** methods.

- Once the fragment reference has been obtained, the activity or fragment could cast the reference appropriately and then call methods directly on that activity or fragment.

```
// Check to see if we have a frame which embed DetailViewFragment directly
View detailsFrame = getActivity().findViewById(R.id.FLDetailView);
mDualPanel = (detailsFrame != null && detailsFrame.getVisibility() ==
View.VISIBLE);

ExampleFragment fragment = (ExampleFragment)
    getFragmentManager().findFragmentById(R.id.example_fragment);
fragment.callCustomMethod(arg1,arg2);
```

ACTIVITY CALLBACKS (I)

- A fragment might need to share events with the activity. To support that, it is necessary to define a callback interface inside the fragment and require that the host activity implement it.
  - When the activity receives a callback through the interface, it can share the information with other fragments in the layout as necessary.
- Steps:
  1. Define an interface in the Fragment which has all of the methods that you would like the Activity to implement as callbacks.
  2. Retrieve each of the interface methods defined in the Activity and store them in private fields in Fragment's onAttach event .
  3. Use of the interface methods anywhere within the Fragment by calling them using the private fields.
  4. In the Activity, implement the interface that the Fragment defines.
  5. Add an onDetach method in the Fragment to set the private fields to null.

ACTIVITY CALLBACKS (II)

```
public class MainActivity extends Activity  implements OnArticleSelectedListener {
    @Override
    public void onArticleSelected(Uri articleUri) {
    }
}

public static class FragmentA extends ListFragment {

    OnArticleSelectedListener mListener;
     ...
    // Container Activity must implement this interface
    public interface OnArticleSelectedListener {
        public void onArticleSelected(Uri articleUri);
    }

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        try {
            mListener = (OnArticleSelectedListener) activity;
        } catch (ClassCastException e) { throw new ClassCastException(activity.toString()
                                        + " must implement OnArticleSelectedListener");
        }
    }
```

4

4

1

2

ACTIVITY CALLBACKS (III)

```
    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        // Append the clicked item's row ID with the content provider Uri
        Uri noteUri = ContentUris.withAppendedId(ArticleColumns.CONTENT_URI,id);
        // Send the event and Uri to the host activity
        mListener.onArticleSelected(noteUri);
    }

    @Override
    public void onDetach() {
        super.onDetach();
        mListener = null;
    }

    ...
}
```

3

5

SUPPORT LIBRARY

- The Android Support Library package is a set of code libraries that provide backward-compatible versions of Android framework APIs as well as features that are only available through the library APIs.
- Each Support Library is backward-compatible to a specific Android API level:
  - **v4 Support Library**: This library is designed to be used with Android 1.6 (API level 4) and higher. It includes the largest set of APIs compared to the other libraries, including Fragments, NotificationCompat, ViewPager, Loader, FileProvider,...
  - **v7 Support Libraries**: Several libraries (v7 appcompat library, v7 cardview library, v7 gridlayout library, v7 mediarouter library,…) designed to be used with Android 2.1 (API level 7) and higher. These libraries provide specific feature sets and can be included in your application independently from each other.
  - **v13 Support Library**: This library is designed to be used for Android 3.2 (API level 13) and higher. It adds support for the Fragment user interface pattern with the FragmentCompat class and additional fragment support classes.
  - **v17 Leanback Library**: Provides APIs to support building user interfaces on TV devices.
- Support Library classes that provide support for existing framework APIs typically have the same name as framework class but are located in the android.support class packages:

```
import android.support.v4.app.*;
import android.support.v7.app.*;
```

SUPPORT LIBRARY

FRAGMENTS COMPATIBILITY

Main differences between standard and compatibility fragment versions:

- The activity rather that extend **Fragment**, should instead extend **FragmentActivity** (v4 appcompat library) or extend **ActionBarActivity** (v7 appcompat library).
- Use *onCreate* callback and ***setContentView(***R.layout.news_articles**)** to create your fragment (instead of *OnCreateView* and inflate() method)
- Using the Support Library APIs you have to use the ***getSupportFragmentManager()*** method to get a **FragmentManager** (instead of ***getFramentManager***).

THE APP BAR

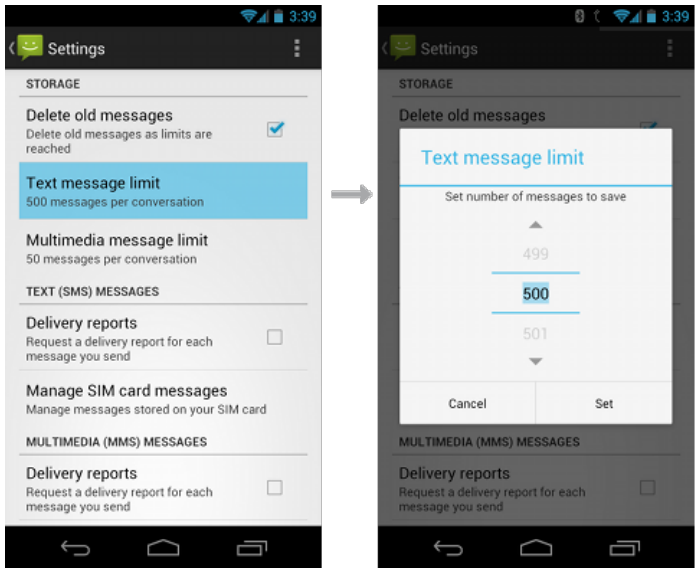
- The app bar, also known as the action bar, is one of the most important design elements in your app's activities, because it provides a visual structure and interactive elements that are familiar to users. Using the app bar makes your app consistent with other Android apps, allowing users to quickly understand how to operate your app and have a great experience. The key functions of the app bar are as follows:
  - A dedicated space for giving your app an identity and indicating the user's location in the app.
  - Access to important actions in a predictable way, such as search.
  - Support for navigation and view switching (with tabs or drop-down lists).

THE APP BAR

- In its most basic form, the action bar displays the title for the activity on one side and an overflow menu on the other. Even in this simple form, the app bar provides useful information to the users, and helps to give Android apps a consistent look and feel.
- Beginning with Android 3.0 (API level 11), all activities that use the default theme have an ActionBar as an app bar. However, app bar features have gradually been added to the native ActionBar over various Android releases. As a result, the native ActionBar behaves differently depending on what version of the Android system a device may be using. By contrast, the most recent features are added to the support library's version of Toolbar, and they are available on any device that can use the support library.
- For this reason, you should use the support library's Toolbar class to implement your activities' app bars.

SETTINGS

- Applications often include settings that allow users to modify app features and behaviors. For example, some apps allow users to specify whether notifications are enabled or specify how often the application syncs data with the cloud.
- If you want to provide settings for your app, you should use Android's Preference APIs to build an interface that's consistent with the user experience in other Android apps (including the system settings). This document describes how to build your app settings using Preference APIs.
- Instead of using View objects to build the user interface, settings are built using various subclasses of the Preference class that you declare in an XML file.



https://developer.android.com/guide/topics/ui/settings.html

SETTINGS

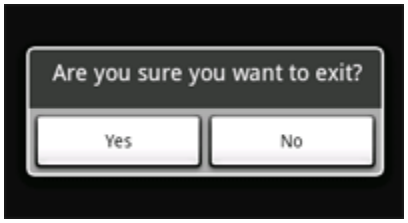
- The value saved in SharedPreferences for each setting can be one of the following data types:
  - Boolean
  - Float
  - Int
  - Long
  - String
  - String Set
- Because your app's settings UI is built using Preference objects instead of View objects, you need to use a specialized Activity or Fragment subclass to display the list settings:
  - If your app supports versions of Android older than 3.0 (API level 10 and lower), you must build the activity as an extension of the PreferenceActivity class.
  - On Android 3.0 and later, you should instead use a traditional Activity that hosts a PreferenceFragment that displays your app settings. However, you can also use PreferenceActivity to create a two-pane layout for large screens when you have multiple groups of settings.--

SETTINGS

Although you can instantiate new Preference objects at runtime, you should define your list of settings in XML with a hierarchy of Preference objects. Using an XML file to define your collection of settings is preferred because the file provides an easy-to-read structure that's simple to update.

```
<PreferenceScreen  xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- opens a subscreen of settings -->
    <PreferenceScreen
        android:key="button_voicemail_category_key"
        android:title="@string/voicemail"
        android:persistent="false">
        <ListPreference
            android:key="button_voicemail_provider_key"
            android:title="@string/voicemail_provider" ... />
        <!-- opens another nested subscreen -->
        <PreferenceScreen
            android:key="button_voicemail_setting_key"
            android:title="@string/voicemail_settings"
            android:persistent="false">
            ...
        </PreferenceScreen>
        <RingtonePreference
            android:key="button_voicemail_ringtone_key"
            android:title="@string/voicemail_ringtone_title"
            android:ringtoneType="notification" ... />
        ...
    </PreferenceScreen>
    ...
</PreferenceScreen>
```

DIALOGS



- A dialog is usually a small window that appears in front of the current Activity.
  - The underlying Activity loses focus and the dialog accepts all user interaction.
  - Dialogs are used for notifications that should interrupt the user and to perform short tasks (progress bar, login prompt).
- Dialogs fragment subclasses:
  - **AlertDialog:** A dialog that can manage zero, one, two, or three buttons, and/or a list of selectable items that can include checkboxes or radio buttons.
  - **DatePickerDialog:** A dialog that allows the user to select a date.
  - **TimePickerDialog:** A dialog that allows the user to select a time.

  These classes define the style and structure for your dialog, but you should use a DialogFragment as a container for your dialog.

ALERT DIALOG

- An **AlertDialog** is capable of constructing most dialog user interfaces. You should use it for dialogs that use any of the following features:
  - A title
  - A text message
  - One, two, or three buttons
  - A list of selectable items (with optional checkboxes or radio buttons)
- To create an AlertDialog, use the AlertDialog.Builder subclass:
  - Get a Builder with *AlertDialog.Builder(Context)* and then use the class's public methods to define all of the AlertDialog properties.
  - After you're done with the Builder, retrieve the AlertDialog object with *create()*.

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage("Are you sure you want to exit?")
 ............
 ............

AlertDialog alert = builder.create();
```

DIALOG FRAGMENT

- You can accomplish a wide variety of dialog designs by extending **DialogFragment** and creating a **AlertDialog** in the *onCreateDialog()* callback method.

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

SHOWING A DIALOG

- When you want to show your dialog, create an instance of your DialogFragment and call *show()*, passing the FragmentManager and a tag name for the dialog fragment.
  - You can get the FragmentManager by calling *getSupportFragmentManager()* from the FragmentActivity or *getFragmentManager()* from a Fragment.
  - The second argument, ***"missiles"***, is a unique tag name that the system uses to save and restore the fragment state when necessary. The tag also allows you to get a handle to the fragment by calling *findFragmentByTag()*.

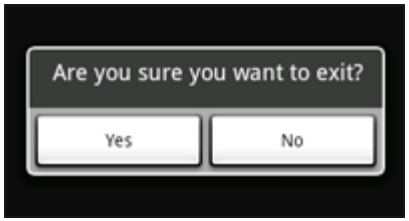
```
public void confirmFireMissiles() {
    DialogFragment newFragment = new FireMissilesDialogFragment();
    newFragment.show(getSupportFragmentManager(), "missiles");
}
```

DIALOGS

OPTIONS

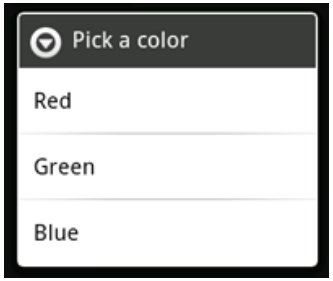
- **Adding Buttons**: You can create an AlertDialog with side-by-side buttons:

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage("Are you sure you want to exit?")
       .setCancelable(false)
       .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id) {
                MyActivity.this.finish();
           }
       })
       .setNegativeButton("No", new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id) {
                dialog.cancel();
           }
       });
AlertDialog alert = builder.create();
```



- **Adding a list**: To create an AlertDialog with a list of selectable items. A list of multiple-choice items (checkboxes) or single-choice items (radio buttons) inside the dialog, use the *setMultiChoiceItems()* and *setSingleChoiceItems()* methods.

```
final CharSequence[] items = {"Red", "Green", "Blue"};

AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Pick a color");
builder.setItems(items, new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int item) {
        Toast.makeText(getApplicationContext(), items[item], Toast.LENGTH_SHORT).show();
    }
});
AlertDialog alert = builder.create();
```



EXAMPLE

```
public static class MyDialogFragment extends DialogFragment {
    int mNum;

    // Create a new instance of MyDialogFragment.
    static MyDialogFragment newInstance(int num) {
        MyDialogFragment f = new MyDialogFragment();

        // Supply num input as an argument.
        Bundle args = new Bundle();
        args.putInt("num", num);
        f.setArguments(args);

        return f;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mNum = getArguments().getInt("num");

        // Pick a style based on the num.
        int style = DialogFragment.STYLE_NORMAL, theme = 0;
        ...
        setStyle(style, theme);   // Defines the dialog style.
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragment_dialog, container, false);
```

EXAMPLE

```
        // Watch for button clicks.
        Button button = (Button)v.findViewById(R.id.show);
        button.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                // When button is clicked, call up to owning activity.
                ((FragmentDialog)getActivity()).showDialog();
            }
        });

        return v;
    }
}

//  Activity showDialog() method:
void showDialog() {
    mStackLevel++;

    // Adds the fragment in a transaction, and removes any currently showing dialog.
    FragmentTransaction ft = getFragmentManager().beginTransaction();
    Fragment prev = getFragmentManager().findFragmentByTag("dialog");
    if (prev != null) {
        ft.remove(prev);
    }
    ft.addToBackStack(null);  // New state on the back stack

    // Create and show the dialog.
    DialogFragment newFragment = MyDialogFragment.newInstance(mStackLevel);
    newFragment.show(ft, "dialog");   // Add and show the fragment in a transaction.
}
```

CUSTOM LAYOUT

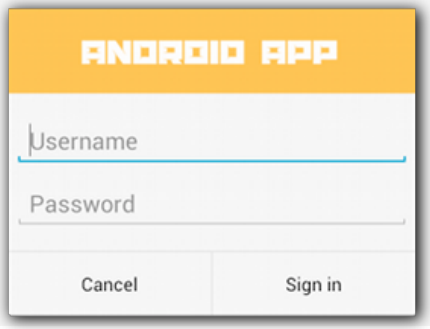
- You can use a custom layout in a dialog, creating a layout and adding it to an AlertDialog by calling *setView()* on your AlertDialog.Builder object.
  - By default, the custom layout fills the dialog window, but you can still use AlertDialog.Builder methods to add buttons and a title.
- To inflate the layout in your DialogFragment, get a LayoutInflater with *getLayoutInflater()* and call inflate():
  - The first parameter is the layout resource ID and the second parameter is a parent view for the layout.
- If you want a custom dialog, you can instead display an Activity as a dialog instead of using the Dialog APIs. Simply create an activity and set its theme to Theme.Holo.Dialog in the <activity> manifest element:

```
<activity android:theme="@android:style/Theme.Holo.Dialog”>
```

DIALOGS

CUSTOM LAYOUT

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

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




```
<LinearLayout xmlns:android="http://
schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <ImageView
        android:src="@drawable/header_logo"
        android:layout_width="match_parent"
        android:layout_height="64dp"
        android:scaleType="center"
        android:background="#FFFFBB33"
        android:contentDescription="@string/
app_name" />
    <EditText
        android:id="@+id/username"
        android:inputType="textEmailAddress"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:layout_marginLeft="4dp"
        android:layout_marginRight="4dp"
        android:layout_marginBottom="4dp"
        android:hint="@string/username" />
    <EditText
        android:id="@+id/password"
        android:inputType="textPassword"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="4dp"
        android:layout_marginLeft="4dp"
        android:layout_marginRight="4dp"
        android:layout_marginBottom="16dp"
        android:fontFamily="sans-serif"
        android:hint="@string/password"/>
</LinearLayout>
```

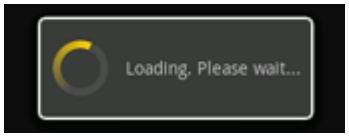
DISMISSING A DIALOG

- When the user touches any of the action buttons created with an AlertDialog.Builder, the system dismisses the dialog for you.
  - The system also dismisses the dialog when the user touches an item in a dialog list, except when the list uses radio buttons or checkboxes
  - You can manually dismiss your dialog by calling *dismiss()* on your DialogFragment.
- To perform certain actions when the dialog goes away, you can implement the *onDismiss()* method in your DialogFragment.
- You can also cancel a dialog, without completing the task. This occurs if the user presses the ***Back*** button, touches the screen outside the dialog area, or if you explicitly call *cancel()* on the Dialog.
  - It is possible to respond to the cancel event by implementing *onCancel()* in your DialogFragment class.

PROGRESS DIALOG

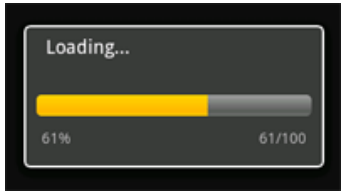
- A **ProgressDialog** can display a progress animation in the form of a spinning wheel, for a task with progress that's undefined, or a progress bar, for a task that has a defined progression.
  - The dialog can also provide buttons, such as one to cancel a download.
- To open a progress dialog call to the *ProgressDialog.show()* method:

```
ProgressDialog dialog = ProgressDialog.show(MyActivity.this, "",
                        "Loading. Please wait...", true);
```



- To show the progression with an animated bar:

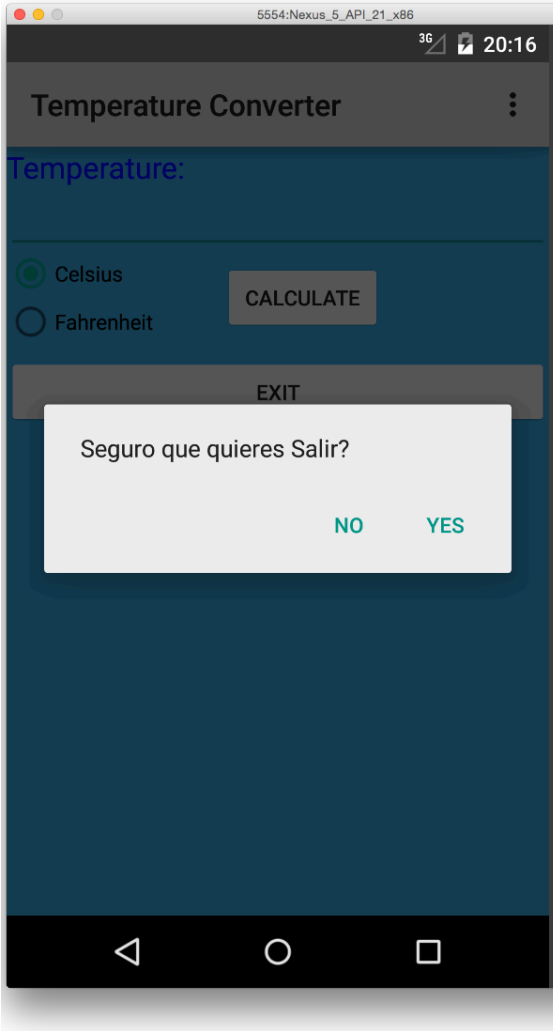
```
ProgressDialog progressDialog;
progressDialog = new ProgressDialog(mContext);
progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
progressDialog.setMessage("Loading...");
progressDialog.setCancelable(false);
```



  - You can increment the amount of progress displayed in the bar by calling either:
    - *setProgress(int)* with a value for the total percentage completed.
    - *incrementProgressBy(int)* with an incremental value to add to the total percentage completed.

ALERT DIALOG EXAMPLE

- ✧ Modify UI for the conversion of Celsius temperature to Fahrenheit Temperature.
- ✧ Add the following:
  - An additional Button to close the application.
  - A AlertDialog to ask for the finalization:
    - Two Buttons.
    - Tittle.



ALERT DIALOG EXAMPLE

DIALOGS

ExitDialogFragment.java

```
public class ExitDialogFragment extends DialogFragment
{
    ExitDialogListener mListener;    // Use this instance of the interface to deliver action events

    /* The activity that creates an instance of this dialog fragment must implement this interface in
     * order to receive event callbacks. */
    public interface ExitDialogListener {
        public void onExitDialogPositiveClick(DialogFragment dialog);
    }

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setMessage("Seguro que quieres Salir?")
                .setCancelable(false)
                .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int id) {
                        // Send the positive button event back to the host activity
                        mListener.onExitDialogPositiveClick(ExitDialogFragment.this);
                    }
                })
                .setNegativeButton("No", new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int id) {
                        dialog.cancel();
                    }
                });
        return builder.create();
    }

    // Override the Fragment.onAttach() method to instantiate the ExitDialogListener
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        try { // Instantiate the ExitDialogListener so we can send events to the host
              mListener = (ExitDialogListener) activity;
        } catch (ClassCastException e) {
              // The activity doesn't implement the interface, throw exception
              throw new ClassCastException(activity.toString() + " must implement ExitDialogListener");
        }
    }
}
```

ALERT DIALOG EXAMPLE

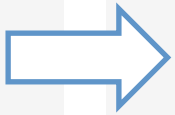
TemperatureConverterActivity.java

```
public class TemperatureConverterActivity extends Activity implements View.OnKeyListener,
ExitDialogFragment.ExitDialogListener
{
    private EditText text;

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

        text = (EditText) findViewById(R.id.editText1);
        text.setOnKeyListener(this);

        final Button buttonExit = (Button) findViewById(R.id.buttonexit);
        buttonExit.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                // Perform action on clicks
                showExitDialog();
            }
        });

    }

    public void showExitDialog() {
        // Create an instance of the dialog fragment and show it
        DialogFragment dialog = new ExitDialogFragment();
        dialog.show(getFragmentManager(), "ExitDialogFragment");
    }

    public void onExitDialogPositiveClick(DialogFragment dialog){
        this.finish();
    }
...
```

STYLES & THEMES

- A **style** is a collection of properties that specify the look and format for a View or window. A style can specify properties such as height, padding, font color, font size, background color, and much more. A style is defined in an XML resource that is separate from the XML that specifies the layout.

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="#00FF00"
    android:typeface="monospace"
    android:text="@string/hello" />
```



```
<TextView
    style="@style/CodeFont"
    android:text="@string/hello" />
```

  - All of the attributes related to style have been removed from the layout XML and put into a style definition called CodeFont, which is then applied with the style attribute.

- A **theme** is a style applied to an entire Activity or application, rather than an individual View (as in the example above). When a style is applied as a theme, every View in the Activity or application will apply each style property that it supports. For example, you can apply the same CodeFont style as a theme for an Activity and then all text inside that Activity will have green monospace font.

STYLE DEFINITION

- To create a set of styles, save an XML file in the res/values/ directory of your project. The name of the XML file is arbitrary, but it must use the .xml extension and be saved in the res/values/ folder.
- For each style you want to create, add a <style> element to the file with a name that uniquely identifies the style (this attribute is required). Then add an <item> element for each property of that style, with a name that declares the style property and a value to go with it (this attribute is required).
- Inheritance. To reference a built-in style, such as TextAppearance, you must use the parent attribute.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="CodeFont" parent="@android:style/
TextAppearance.Medium">
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

THEME DEFINITION

- To set a theme for all the activities of your application, open the AndroidManifest.xml file and edit the <application> tag to include the android:theme attribute with the style name.

```
<application android:theme="@style/CustomTheme">
```

- If you want a theme applied to just one Activity in your application, then add the android:theme attribute to the <activity> tag instead.

- Just as Android provides other built-in resources, there are many pre-defined themes that you can use, to avoid writing them yourself. For example, you can use the Dialog theme and make your Activity appear like a dialog

```
<activity android:theme="@android:style/Theme.Dialog">
```