# Intro to Navigational Patterns

Includes parts of Chapter 10

## Overview

- Learn about typical navigation patterns
- Discover various widgets and views for implementing app’s navigation
- Explore the different types of dialogs
- Learn how to incorporate dialogs and dialog fragments

# Fragment Lifecycle Callbacks Review

**`onCreate()`**

The `onCreate()` method is called before the Fragment's `onCreateView()`.

Typically, in this method you would get any values sent to the fragment from the hosting Activity (by accessing the `Intent`'s extras) or initialize any other variables.

You cannot gain access to or modify the fragment's View hierarchy (layout) since it may not exist yet. This would likely cause an exception!

## Fragment Lifecycle Callbacks Review

**`onCreateView()`**

After the fragment's `onCreate()` call is finished, the fragment's `onCreateView()` is called.

Typically, it is used to create the fragment's View hierarchy (its layout).

Must return a View – the main UI view of the fragment.  May be inflated (from an XML layout) or programmatically created.

## Fragment Lifecycle Callbacks Review

**`onViewCreated()`**

This method is called after the fragment's view has been created, i.e., `onCreateView()` call has finished.

It is possible to gain access the layout's view objects now.

It is is mainly used for setting up control listeners and other final initializations.

## Fragment Lifecycle Callbacks Review

**To recap**:

The `onCreate()` is called first, for performing any non-graphical initializations.

Next, `onCreateView()` is called and it must create the fragment's view; then you can initialize any View variables you want to use in the fragment.

Finally, `onViewCreated()` is called and you can perform any final initializations.

## Fragment Class Instance

`newInstance` method for fragment’s controller object creation

```
public static SomeFragment newInstance( int param ) {
     SomeFragment fragment = new SomeFragment ();
     Bundle args = new Bundle();
     args.putInt( ”param", param );
     fragment.setArguments( args );
     return fragment;
}
```

Technically it would be possible to create a Fragment controller class with an overloaded constructor (which would accept some parameters).

## Fragment Class Instance

However, Android re/creates a fragment only using a default (parameter-less) constructor.

Consequently, an established way to pass parameters is to use a `Bundle` and send the data this way, using `setArguments` and `getArguments` methods.

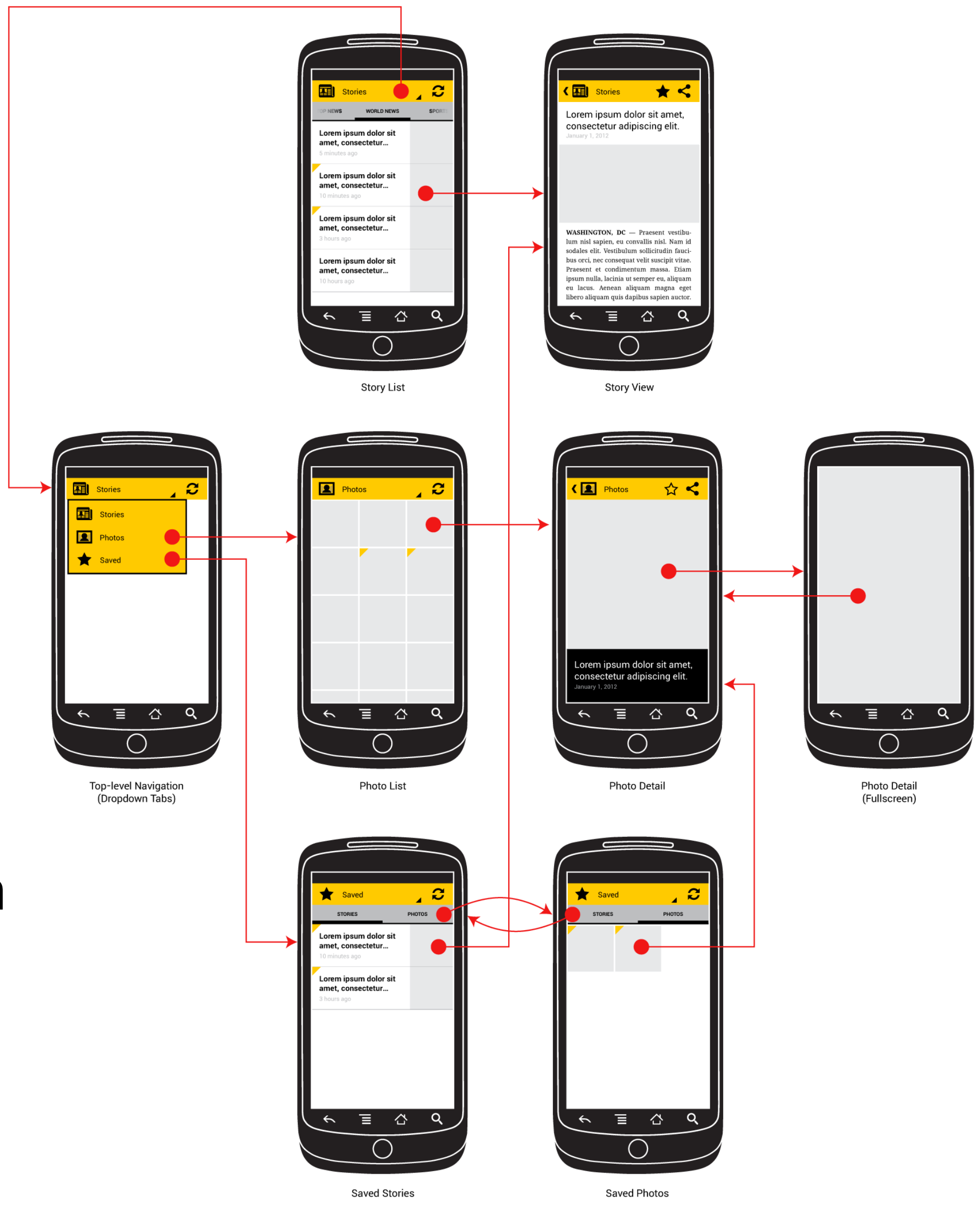
Also, the `setArguments` call must happen BEFORE the fragment is attached to an activity.

So, having a `newInstance` (or similar) method is an effective way to create an instance using the default constructor, but also pass some parameters to it.

# Architecting Application's Navigation

An application with 8 screens

From: developer.android.com


Stories
TOP NEWS
WORLD NEWS
SPORTS
Lorem ipsum dolor sit amet, consectetur...
Story List
Story View
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
January 1, 2012
WASHINGTON, DC — Praesent vestibulum nisl sapien, eu convallis nisl. Nam id sodales elit. Vestibulum sollicitudin faucibus orci, nec consequat velit suscipit vitae. Praesent et condimentum massa. Etiam ipsum nulla, lacinia ut semper eu, aliquam eu lacus. Aenean aliquam magna eget libero aliquam quis dapibus sapien auctor.
Photos
Saved
Top-level Navigation (Dropdown Tabs)
Photo List
Photo Detail
Photo Detail (Fullscreen)
STORIES
PHOTOS
Saved Stories
Saved Photos

# Architecting Application’s Navigation

- Initial set of Android application navigation scenarios:
  - Entry navigation
  - Lateral navigation
  - Descendant navigation
  - Back navigation
  - Ancestral navigation
  - External navigation

## Entry Navigation

- Entry navigation is how a user navigates into an application.
- There are many different ways this can occur, such as:
  - from a *Home screen* widget,
  - from the *All Apps* screen,
  - from a notification listed within the status bar, or
  - from another application altogether.

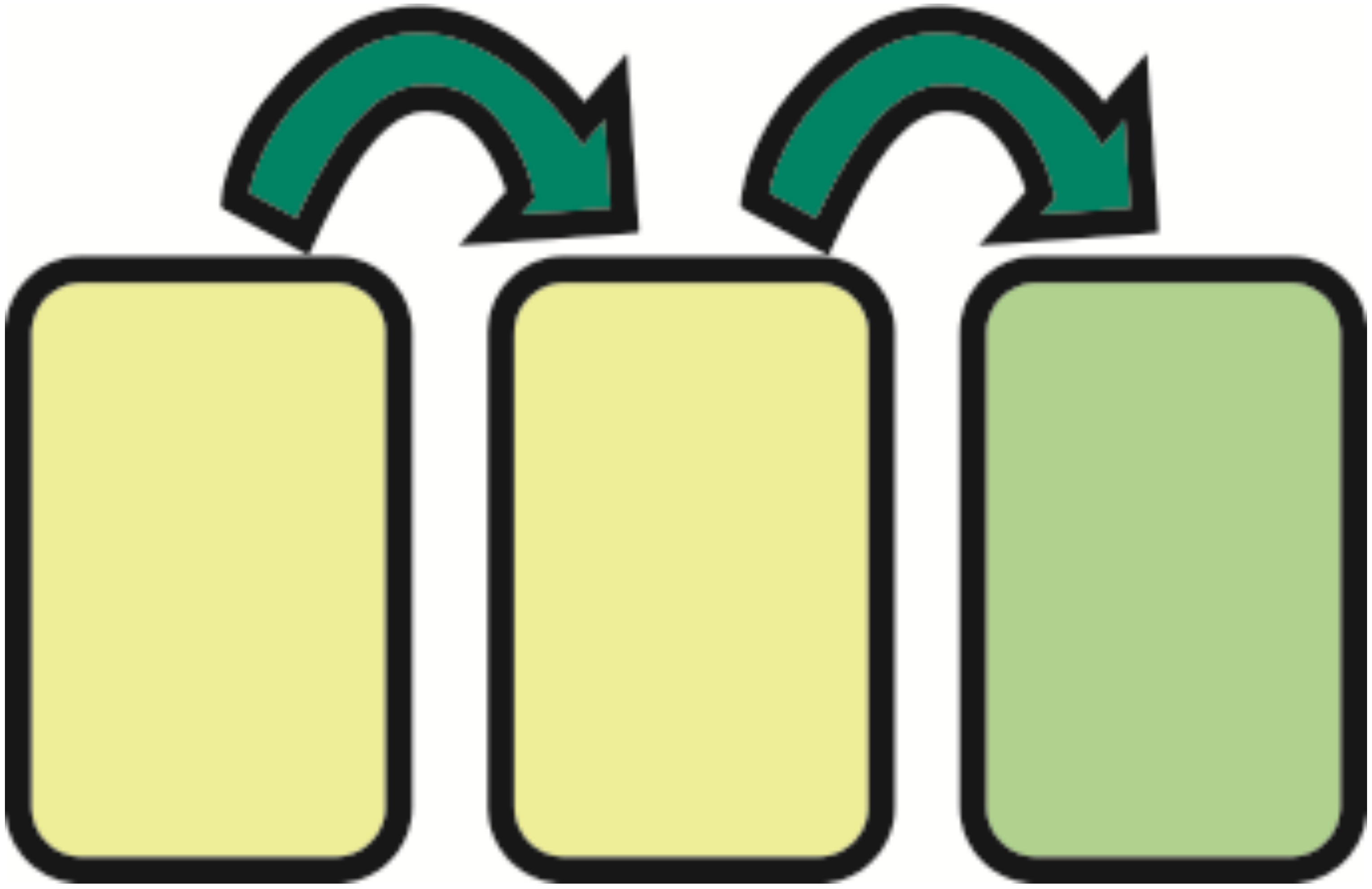
## Lateral Navigation

- Lateral navigation is mainly for applications that have screens residing on the same hierarchy level.
- If your application has more than one screen residing on the same level, you may want to provide users the ability to navigate laterally across that hierarchy level by implementing swipe navigation, tabs, or both, or another method.

## Lateral Navigation

- To implement lateral navigation in your application for activities, all you need to do is make a call to `startActivity()` using an `Intent` with the lateral `Activity` and ensure that each of the activities is on the same hierarchy level within your application.
- If your activities are on the same hierarchy level but not the top level, you can define the `parentActivityName` attribute in your manifest and set the parent `Activity` to be the same for each of the activities that should reside on the same hierarchy level.

## Lateral Navigation (Cont'd)



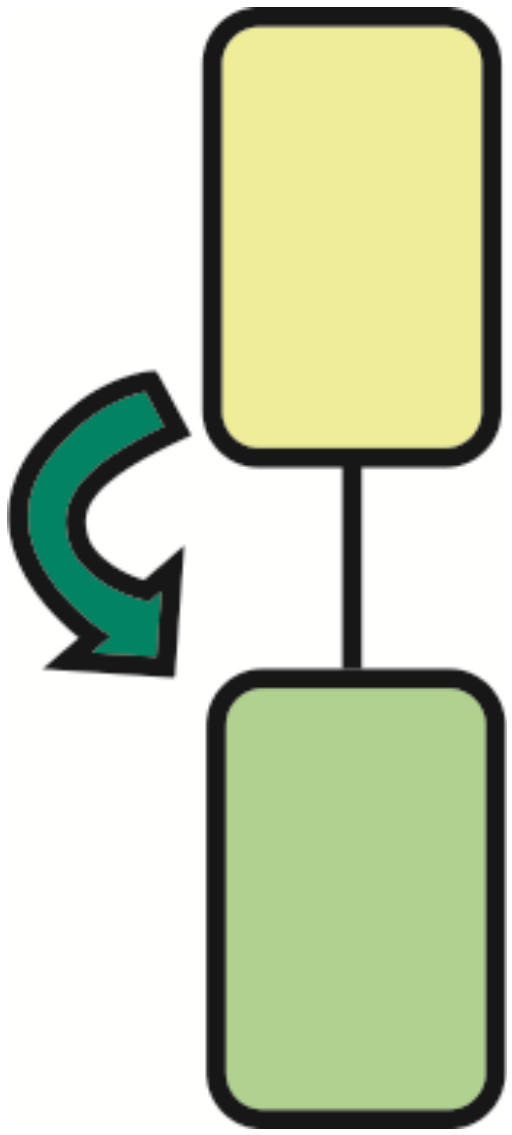
# Descendant Navigation

- Descendant navigation is used when an application has more than one hierarchy level.
- This means that users can navigate deeper into the application's hierarchy levels.
- Usually this is done by creating a new `Activity` with `startActivity()`.

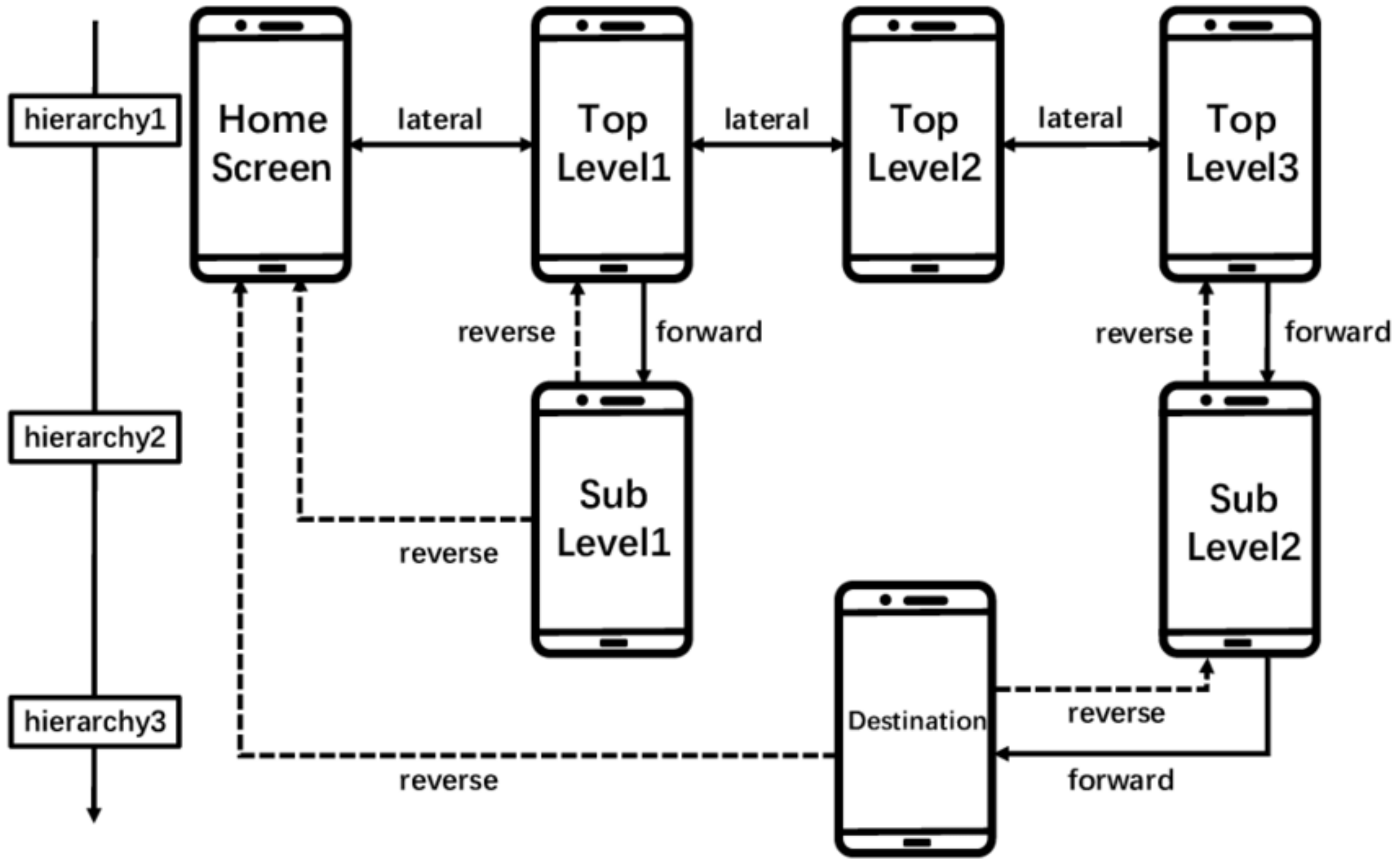
## Descendant Navigation

- To implement descendant navigation, make sure that the descendant `Activity` declares `parentActivityName` in your application manifest and set the `Activity` to be the ancestor.
- Then, from the ancestor `Activity`, create an `Intent` with the descendant `Activity`, and simply call `startActivity()`.

# Descendant Navigation



# Architecting Application's Navigation



An app with a hierarchy of screens

From: "How Should I Improve the UI of My App?: A Study of User Reviews of Popular Apps in the Google Play", ACM Transactions on Software Engineering and Methodology 30(3):1-38

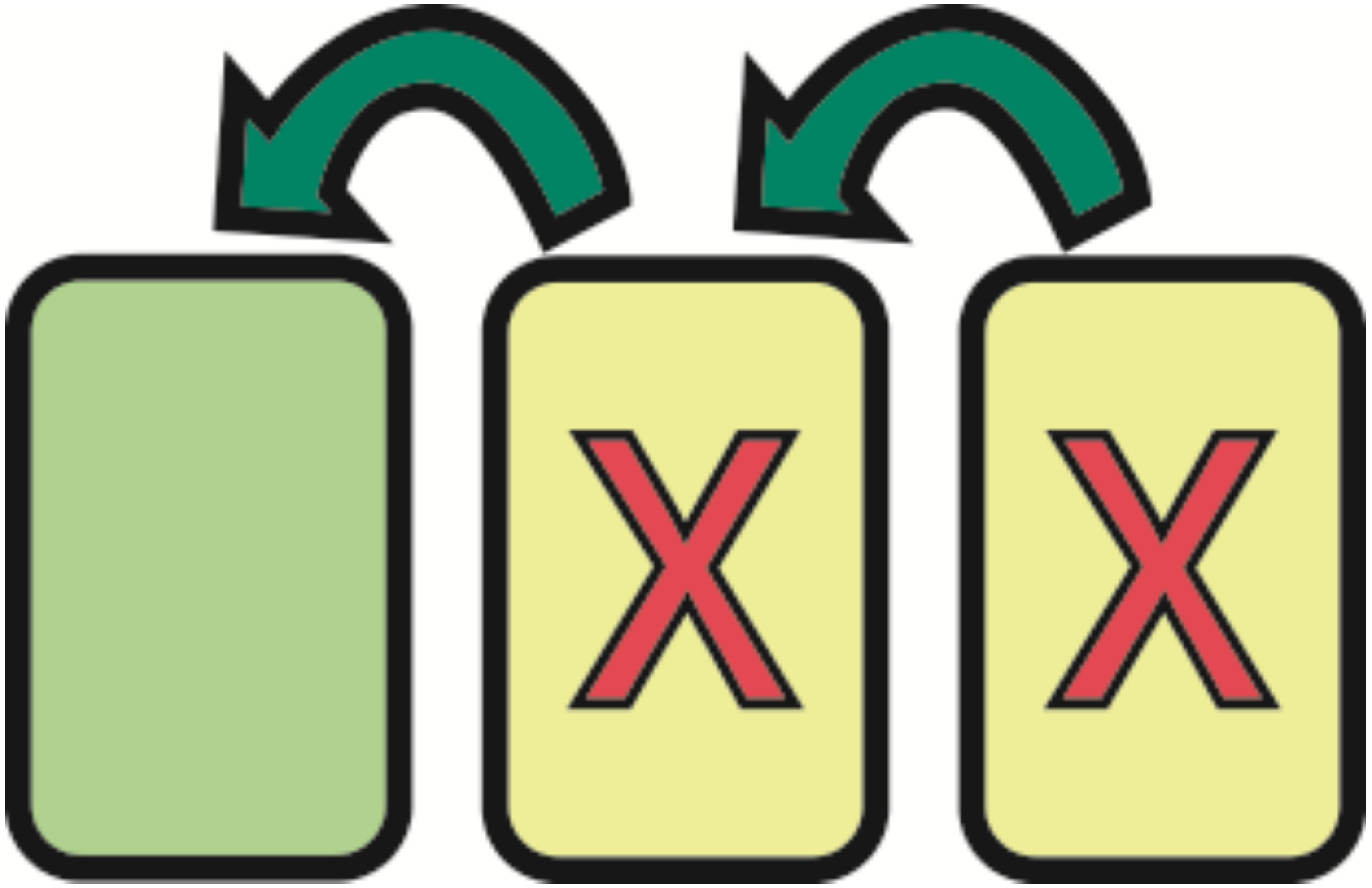
## Back Navigation

- Back navigation is used when a user clicks the Back button located on the Android navigation bar or presses a hardware Back button.
- The default behavior navigates the user to the last `Activity` or `Fragment` that was placed on the back stack.

## Back Navigation

- To override this behavior, call `onBackPressed()` from within your `Activity`.
- You do not have to do anything special to implement back navigation in your application unless you are working with fragments.
- With fragments, if you want to navigate back, you need to make sure that you add your Fragment to the back stack with a call to `addToBackStack()`.

## Back Navigation



## Ancestral Navigation

- Ancestral navigation, or up navigation, is used when your application has more than one hierarchy level and you must provide a means for navigating up.
- To implement ancestral application within your application, you must do two things.

## Ancestral Navigation

- The first is to make sure that your descendant `Activity` defines the correct `parentActivityName` attribute within your application manifest; then, in your `onCreate()` method of your `Activity`, simply call the following:

```
myActionBar.setDisplayHomeAsUpEnabled(true);
```

## External Navigation

- External navigation is when the user navigates from one application to another.
- The current application may want to transition to a different one altogether.
- It is possible to retrieve a result from another application (e.g., a contact info). It is done with

  `ActivityResultLauncher`

  and an overridden callback

  `onActivityResult()`

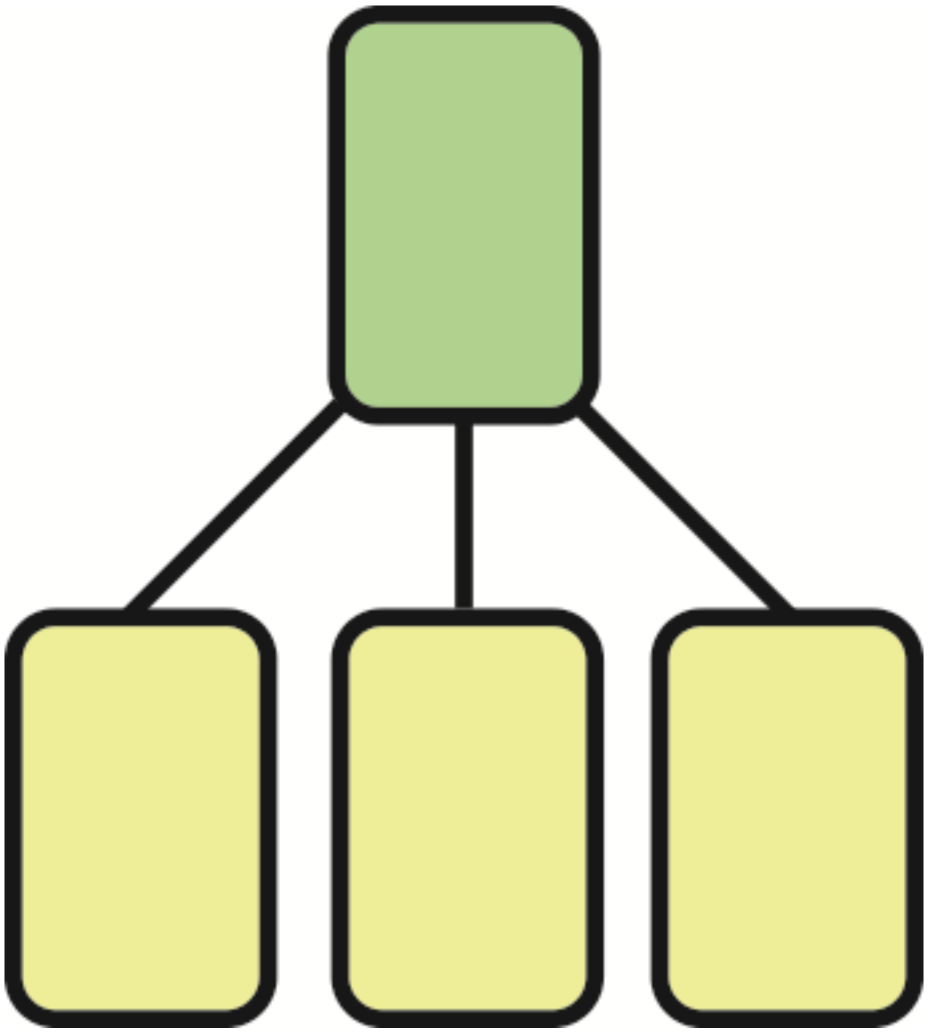
## Launching Tasks and Navigating the Back Stack

- A task is one or more activities that are used for accomplishing a specific goal for the user.
- The back stack is where Android manages these activities with a last-in-first-out (stack) ordering.
- As the activities of a task are created, they are added to the back stack in order.

## Launching Tasks and Navigating the Back Stack

- If the default behavior applies to an `Activity`, and the user presses the Back button, Android removes the last `Activity` added to the stack.
- In addition, if the user instead were to press the Home button, the task would then move itself and the activities to the background.
- The task may be resumed by the user at a later point in time, as moving it to the background does not destroy the `Activity` or task.

## Relationships between Screens



## Relationships between Screens

- For your application to understand the hierarchical relationship among your activities, all you need to do is add the `android:parentActivityName` to your `<activity>` tag of your Android manifest file.
- To ensure proper support on older versions of Android, you should also define a `<meta-data>` tag with the name of `android.support.PARENT_ACTIVITY`.

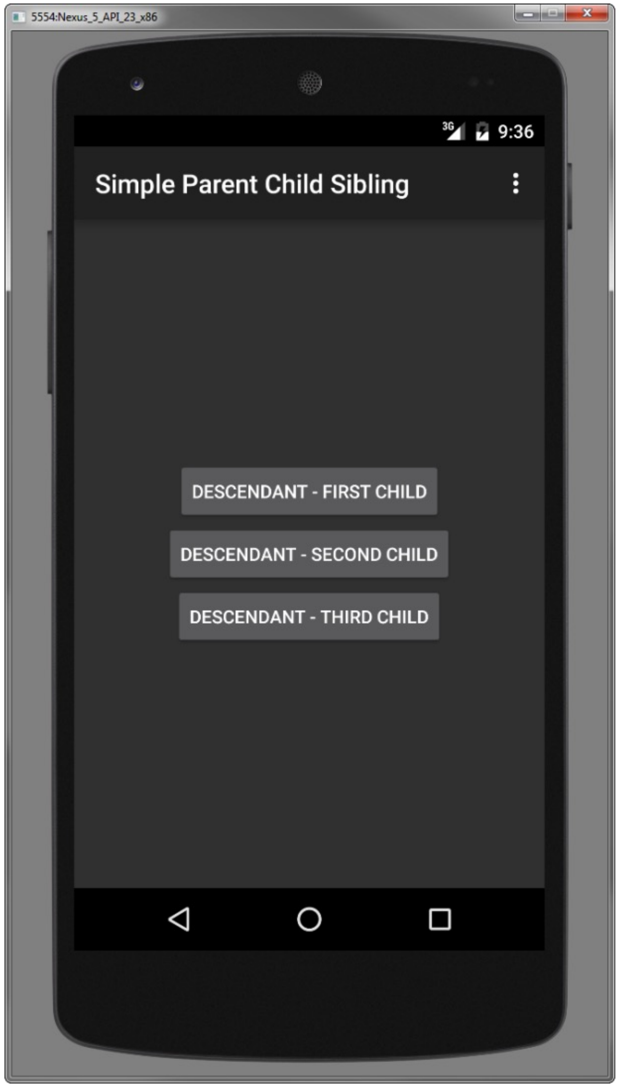
## Relationships between Screens

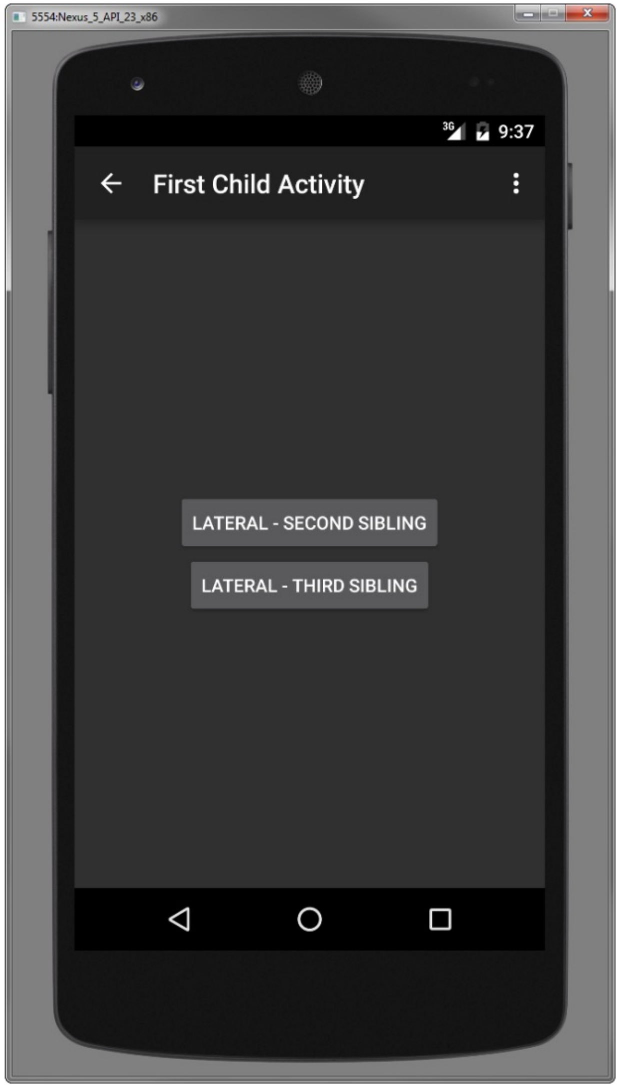
```
<activity
    android:name=".FirstChildActivity"
    android:label="@string/title_activity_first_child"
    android:parentActivityName=“.SimpleParentChildSiblingActivity" >
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value="com.introtoandroid
                .simpleparentchildsibling
                .SimpleParentChildSiblingActivity" />
</activity>
```

## Android Navigation Design Patterns

- There are many design patterns that are commonly found within Android applications.
- Many of these patterns are highlighted within the Android documentation due to their effectiveness.
- Here are a few of these common navigation design patterns:
  - Targets
  - Swipe views
  - Tabs
  - Navigation drawer
  - Master-detail flow
  - Action bar menu

## Targets

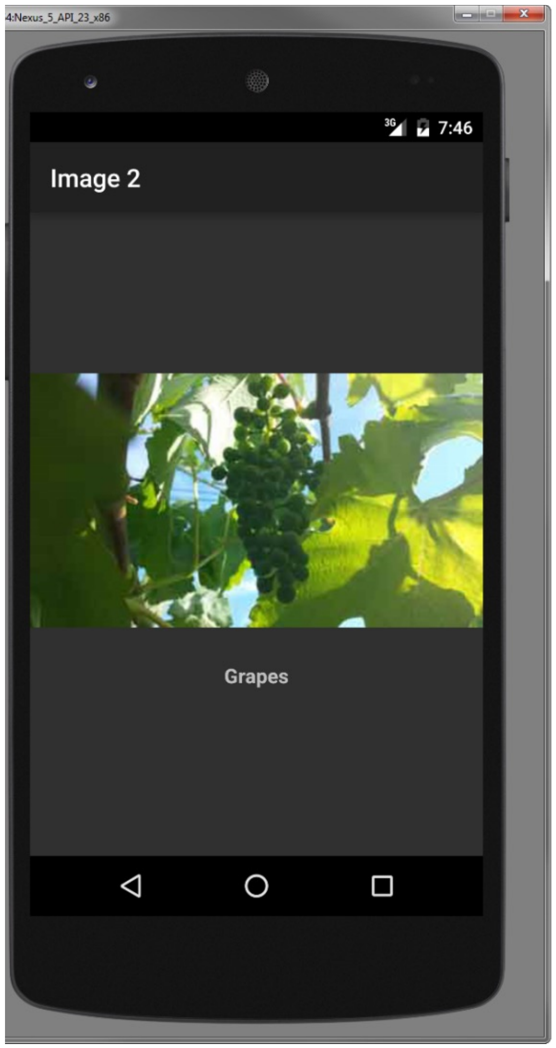

5554:Nexus_5_API_23_x86
9:36
Simple Parent Child Sibling
DESCENDANT - FIRST CHILD
DESCENDANT - SECOND CHILD
DESCENDANT - THIRD CHILD

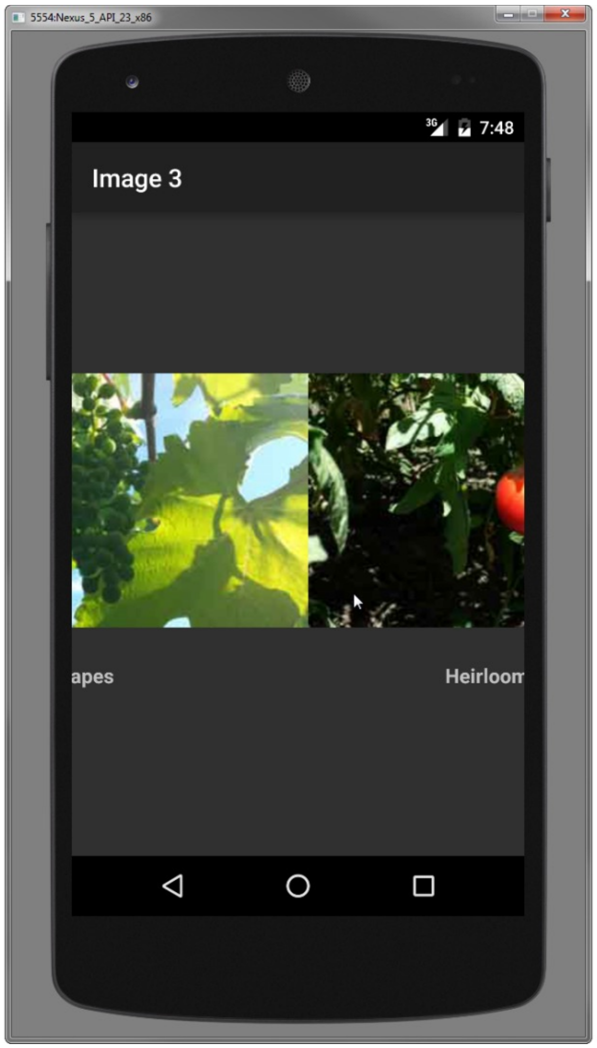

5554:Nexus_5_API_23_x86
9:37
First Child Activity
LATERAL - SECOND SIBLING
LATERAL - THIRD SIBLING

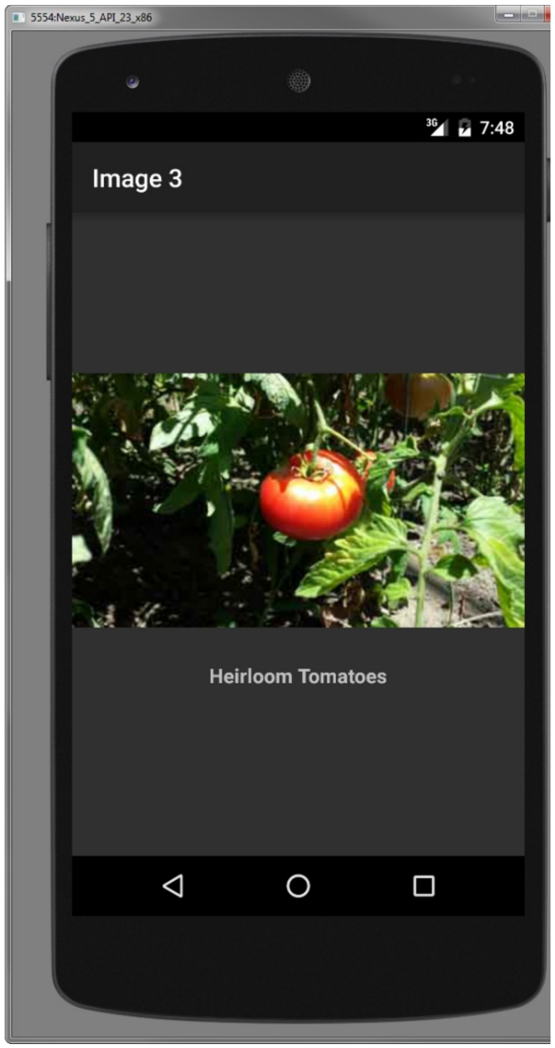
## Targets

```
Button firstChild = findViewById(R.id.firstChild);
firstChild.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(getApplicationContext(),
                                   FirstChildActivity.class);
        startActivity(intent);
    }
});
```

# Swipe Views


4:Nexus_5_API_23_x86
7:46
Image 2
Grapes


5554:Nexus_5_API_23_x86
7:48
Image 3
apes
Heirloom


5554:Nexus_5_API_23_x86
7:48
Image 3
Heirloom Tomatoes

## Swipe Views

```
<android.support.v4.view.ViewPager      <- deprecated
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

- Use ViewPager from the JetPack library (AndroidX)

  `androidx.viewpager.widget.ViewPager`

  https://developer.android.com/training/animation/screen-slide

- Also, ViewPager has been superseded by ViewPager2 (in AndroidX), which supports vertical paging.

## Swipe Views

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:gravity="center"
   android:orientation="vertical"
   tools:context=".SimpleViewPagerActivity$PlaceholderFragment">
   <ImageView  android:id="@+id/image_view"
      android:layout_width="match_parent"
      android:layout_height="wrap_content" />
   <TextView android:id="@+id/section_label"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:gravity="center_horizontal"
      android:textSize="16sp"
      android:textStyle="bold" />
</LinearLayout>
```

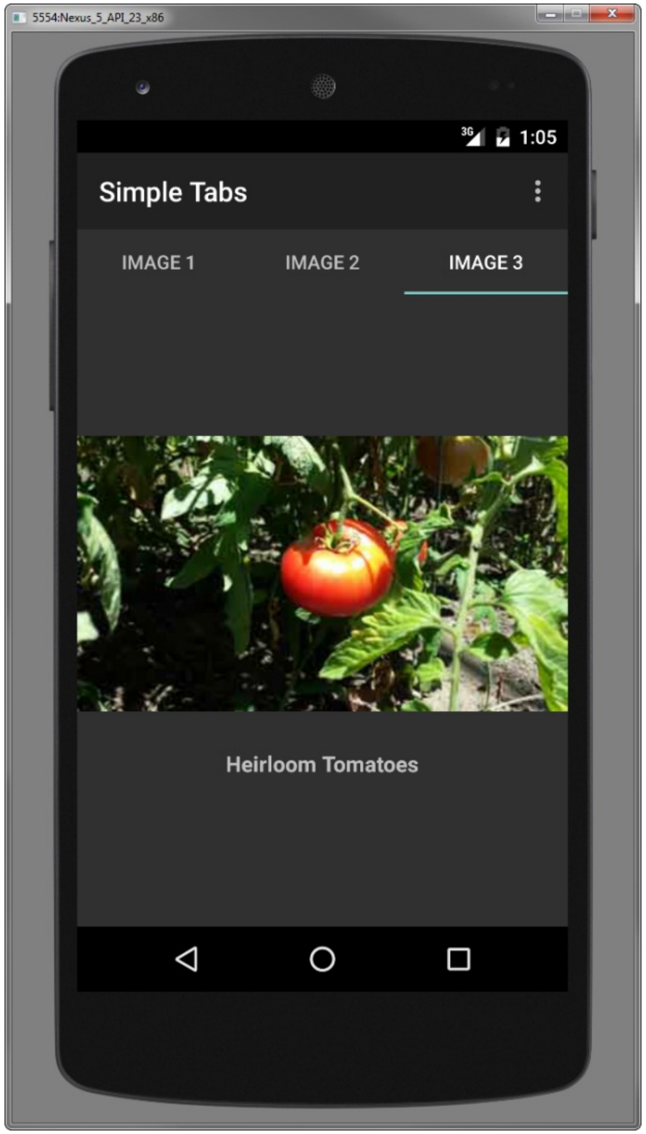
## Navigating with Fragments

- We have covered how the various navigation scenarios work when using activities.
- If you have implemented a descendant `ViewPager` with fragments, and there are dozens or potentially hundreds of fragments that a user could page through, it is probably not a good idea to add each of the fragments to the back stack just to allow the user the ability to be able to navigate back up to its ancestor `Activity`.

## Navigating with Fragments

- If so, and the user has a habit of using the Back button rather than the Up button, he or she may become extremely frustrated having to navigate back through dozens of fragments just to get to the ancestral `Activity`.
- It is clear that ancestral navigation should be used to handle this scenario, but not every user knows to use the Up button.

# Tabs


5554:Nexus_5_API_23_x86
1:05
Simple Tabs
IMAGE 1
IMAGE 2
IMAGE 3
Heirloom Tomatoes

## Tabs

```
<android.support.design.widget.TabLayout
    android:id="@id/tab_layout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

The above will add tabs in your layout.

Now, use:

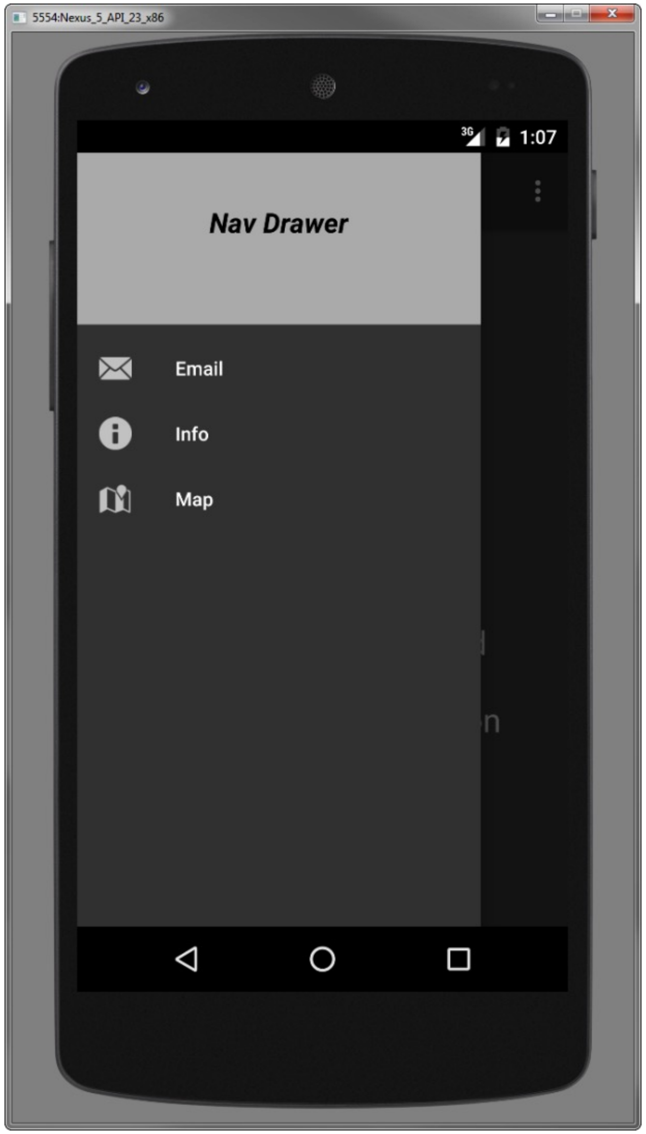
```
com.google.android.material.tabs.TabLayout
```

Tabs can be added using a FragmentPagerAdapter or FragmentStateAdapter when using ViewPager2.

## Tabs

```
tabLayout.setOnTabSelectedListener(
    new TabLayout.ViewPagerOnTabSelectedListener(mViewPager));
```

# Navigation Drawer



## Navigation Drawer

The example from the textbook uses an older version of DrawerLayout (from support library v4).

Now, use DrawerLayout from the JetPack library (AndroidX):

```
androidx.drawerlayout.widget.DrawerLayout
```

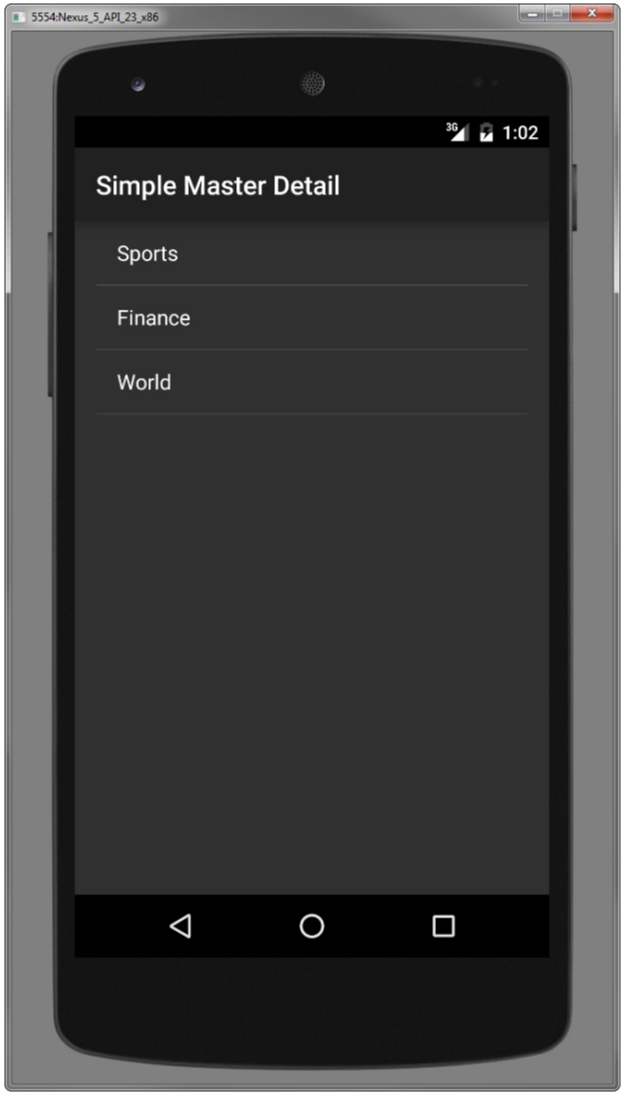
# Navigation Drawer

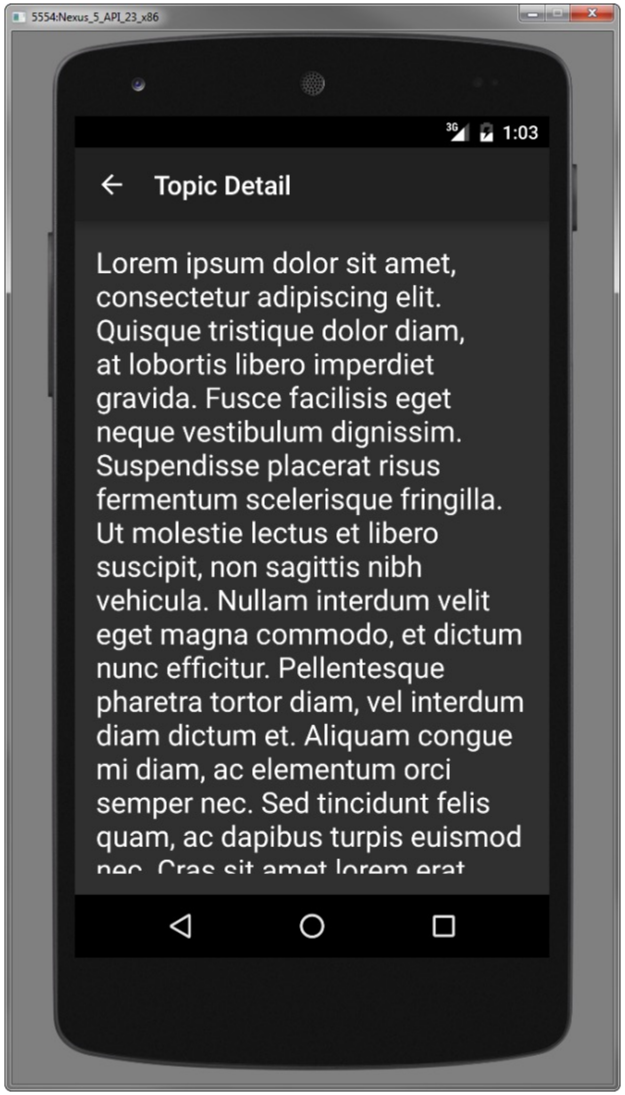
```
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:drawer="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout" android:layout_width="match_parent"
    android:layout_height="match_parent" android:fitsSystemWindows="true"
    tools:context=".SimpleNavDrawerAndViewActivity">
    <LinearLayout android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <TextView android:id="@+id/text_view"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:padding="16dp"
            android:text="@string/instructions"
            android:textSize="24sp"/>
    </LinearLayout>
    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view" android:layout_width="wrap_content"
        android:layout_height="match_parent" android:layout_gravity="start"
        drawer:headerLayout="@layout/drawer_headers"
        drawer:menu="@menu/menu_nav_drawer" />
</android.support.v4.widget.DrawerLayout>
```

## Master-Detail Flow

- Use the master-detail flow with fragments when using views such as lists or grids.
- One `Fragment` is for the list or grid and the other `Fragment` for the associated detail view.
- When selecting an item in the `Fragment` list or grid on a single-pane layout, launch a new `Activity` to display the `Fragment` detail view.
- On a double-pane layout, show both fragments side-by-side. This may involve changing the device orientation. This example has side-by-side fragments for devices with available width of 600dp (a 7” tablet).

## Master-Detail Flow

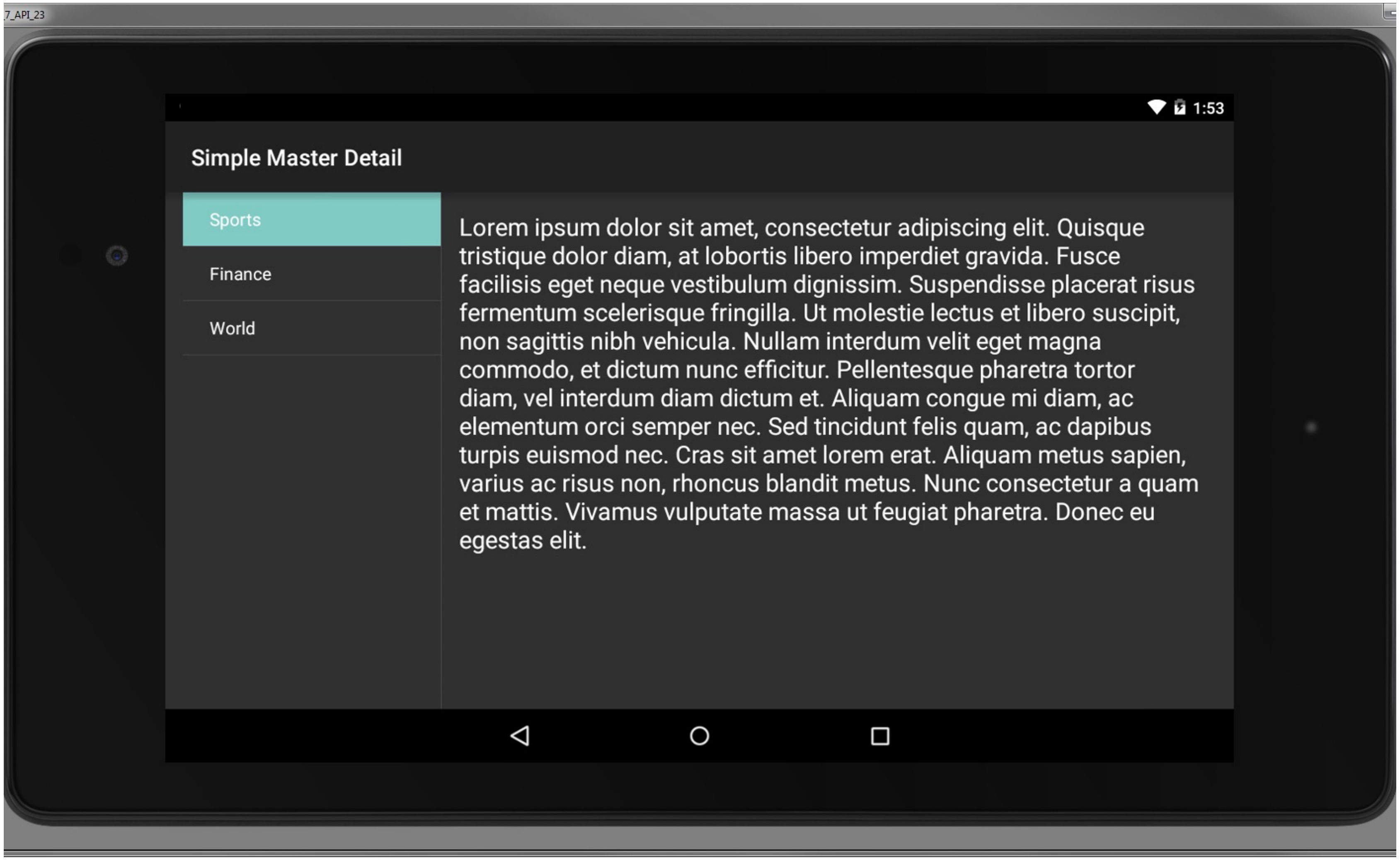




Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque tristique dolor diam, at lobortis libero imperdiet gravida. Fusce facilisis eget neque vestibulum dignissim. Suspendisse placerat risus fermentum scelerisque fringilla. Ut molestie lectus et libero suscipit, non sagittis nibh vehicula. Nullam interdum velit eget magna commodo, et dictum nunc efficitur. Pellentesque pharetra tortor diam, vel interdum diam dictum et. Aliquam congue mi diam, ac elementum orci semper nec. Sed tincidunt felis quam, ac dapibus turpis euismod nec. Cras sit amet lorem erat

# Master-Detail Flow

Only on a larger screen



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque tristique dolor diam, at lobortis libero imperdiet gravida. Fusce facilisis eget neque vestibulum dignissim. Suspendisse placerat risus fermentum scelerisque fringilla. Ut molestie lectus et libero suscipit, non sagittis nibh vehicula. Nullam interdum velit eget magna commodo, et dictum nunc efficitur. Pellentesque pharetra tortor diam, vel interdum diam dictum et. Aliquam congue mi diam, ac elementum orci semper nec. Sed tincidunt felis quam, ac dapibus turpis euismod nec. Cras sit amet lorem erat. Aliquam metus sapien, varius ac risus non, rhoncus blandit metus. Nunc consectetur a quam et mattis. Vivamus vulputate massa ut feugiat pharetra. Donec eu egestas elit.

# Single Activity Architecture

- Single Activity Architecture is a particular app architecture with a single activity and many fragments for different screens/parts of the app's functionality
- When the user launches an app, a base/manager Activity is launched
- When the user navigates to another screen, the same Activity is reused, but it hosts a different fragment

## Single Activity Architecture

- Single Activity Architecture offers several advantages:
  - Screen transitions are faster
  - Portions of the screen, e.g., ActionBar or Navigation Drawer may be retained
  - Navigation is managed in one place
  - Simpler communication across screens
  - Simpler AndroidManifest file

## Menus

- Presenting menus to users to provoke action has been around since Android API Level 1.
- With Android API Level 11, menus have been replaced with a newer design pattern known as the `ActionBar`.
- Documentation on menus:

  developer.android.com/guide/topics/ui/menus

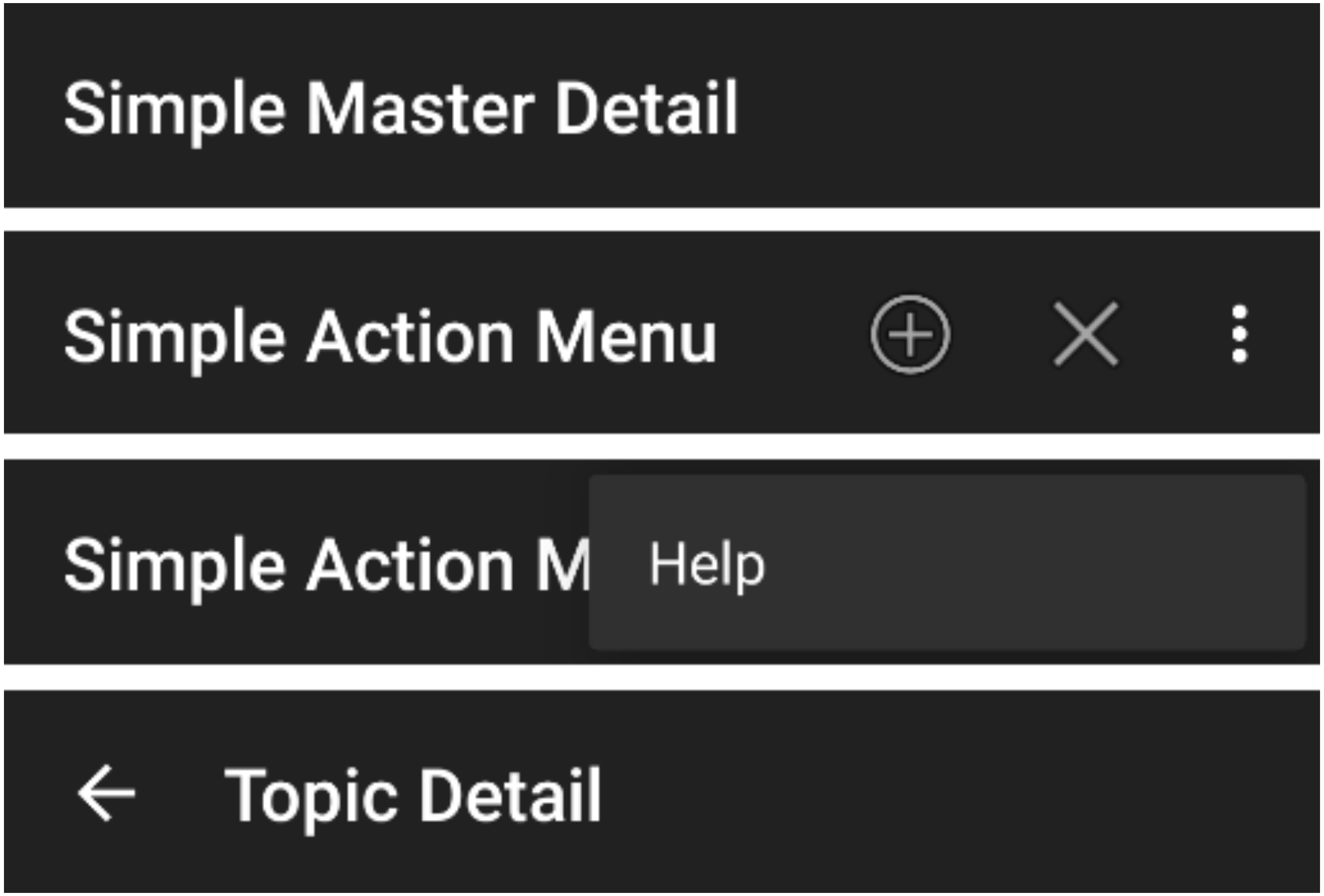
## Menus

- For supporting menus in applications prior to API Level 11, you may want to consider implementing menus. No longer used
- There are three different types of menus available for use:
  - Options menu
  - Context menu
  - Pop-up menu

## ActionBar

- The `ActionBar` has become the preferred way to present actions to users.
- The `ActionBar` is where to place actions that you want to make available to a particular `Activity`.
- You can add actions to and remove them from the `ActionBar` from within an `Activity` or `Fragment`.
- In JetPack, the ActionBar class is: `androidx.appcompat.app.ActionBar`

## ActionBar


Simple Master Detail
Simple Action Menu
Simple Action M
Help
Topic Detail

## Application Icon

- You may place your application’s icon within the `ActionBar`.
- If your application supports up navigation, your application icon would be where a user would press to navigate up a level.

## View Control

- A `View` control may be placed on the `ActionBar` to enable actions such as search or navigation using tabs or a drop-down menu.

- More information on ActionBar in AndroidX:

  https://developer.android.com/guide/fragments/appbar

  and

  https://developer.android.com/reference/androidx/appcompat/app/ActionBar

## Action Buttons

- Action buttons are usually icons, text, or both icons and text for displaying the actions you would like to make available to users from within your `Activity`.

- More information on Action buttons:

  https://developer.android.com/develop/ui/views/components/appbar/actions

## Action Buttons

**menu_simple_action_menu.xml**

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <item android:id="@+id/menu_add"
        android:icon="@android:drawable/ic_menu_add"
        android:orderInCategory="2"
        android:showAsAction="ifRoom|withText"
        android:title="@string/action_add"/>
    <item android:id="@+id/menu_close"
        android:icon="@android:drawable/ic_menu_close_clear_cancel"
        android:orderInCategory="4"
        android:showAsAction="ifRoom|withText"
        android:title="@string/action_close"/>
    <item android:id="@+id/menu_help"
        android:icon="@android:drawable/ic_menu_help"
        android:orderInCategory="5"
        android:showAsAction="never"
        android:title="@string/action_help"/>
</menu>
```

## Action Buttons

**onCreateOptionsMenu** callback is used to create a menu for an Activity or for a Fragment.

**getMenuInflater** returns an inflater to use for the menu

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.simple_action_menu, menu);
    return true;
}
```

## Action Buttons

- `android:showAsAction` can be either:
  - `ifRoom`
  - `withText` – include text defined as android:title
  - `never` – do not include the item on the action bar, but put it in the overflow menu
  - `always` – should be avoided, if possible

## Action Overflow

- Action items that you are not able to fit on the main `ActionBar` will be placed within the overflow section (overflow menu).
- Make sure to order your action items in order of their importance and frequency of use.
- Use `android:orderInCategory` to specify the ordering of items in the menu

## Action Overflow

- If your application supports both small screens and large tablets, you may want your `ActionBar` to display differently based on the type of device.
- Since there is more room on the `ActionBar` on large tablets, you are able to fit more actions.

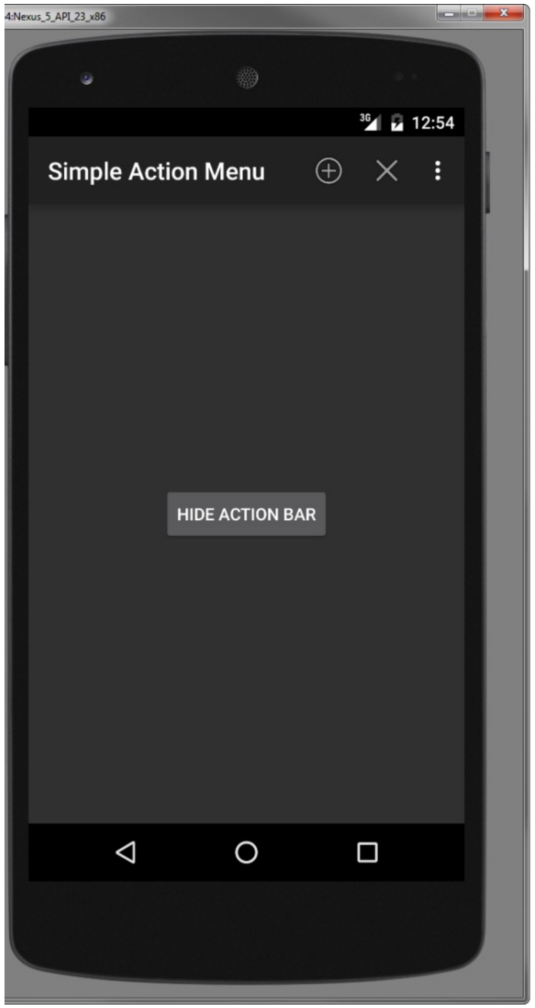
## Action Overflow

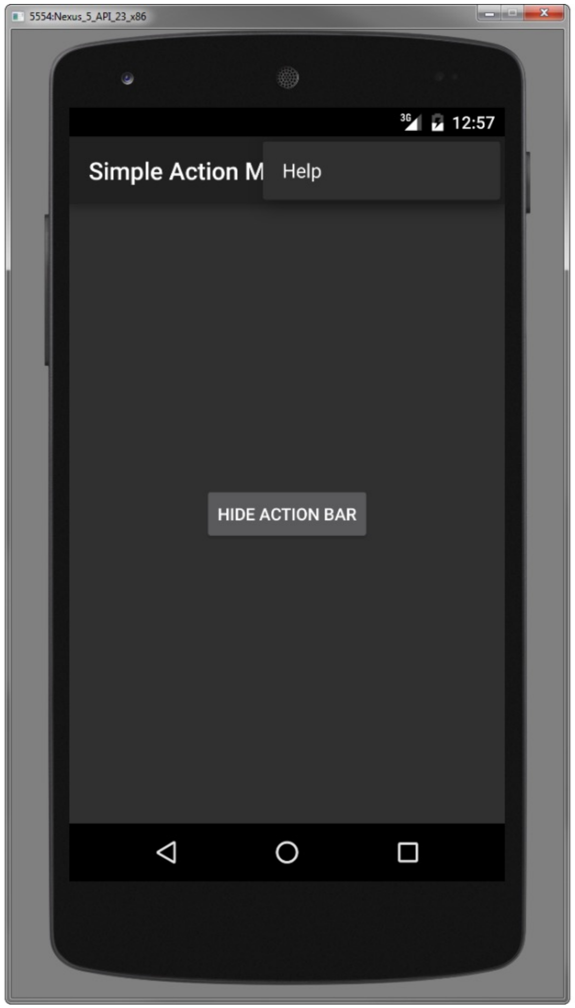
- With smaller screens, rather than having all of the actions displayed in the overflow, you should add a `Toolbar` to the bottom of your `View` and add the additional action items there.
- Before Android 5.0, support for a split `ActionBar` was available.
- If your application makes use of the `Theme.Material` default theme of Android 5.0 and above, this is no longer supported and inserting a `Toolbar` at the bottom of your `View` is the preferred way for achieving the same result.

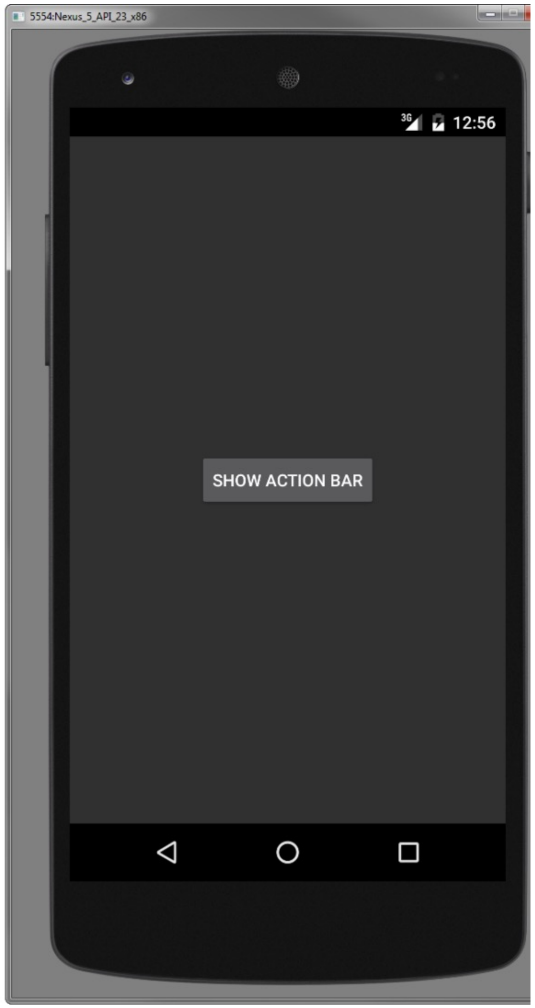
## Action Overflow

```
getActionBar().hide();
getActionBar().show();
```

## Action Overflow







## Action Overflow

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_add:
            Toast.makeText(this, "Add Clicked", Toast.LENGTH_SHORT).show();
            return true;
        case R.id.menu_close:
            finish();
            return true;
        case R.id.menu_help:
            Toast.makeText(this, "Help Clicked",
                Toast.LENGTH_SHORT).show();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

## Toolbar as ActionBar

```
<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary" />
```

Now, use Jetpack’s Toolbar:

```
androidx.appcompat.widget.Toolbar
```

## Toolbar as ActionBar

- Toolbar is a generalization of ActionBar
- It can be used in place of the Activity's ActionBar

```
Toolbar toolbar = findViewById( R.id.toolbar );
setSupportActionBar(toolbar);
```

- It can also be used anywhere else within a layout

## Floating Action Button

- The `FloatingActionButton` is a recent addition to the Android design support library.
- The `FloatingActionButton` is used to perform a primary action of a particular `Activity`.
- For example, a contacts application would use a `FloatingActionButton` as the primary action for initiating the addition of a new contact by launching an “add contact” `Activity`.

## Floating Action Button

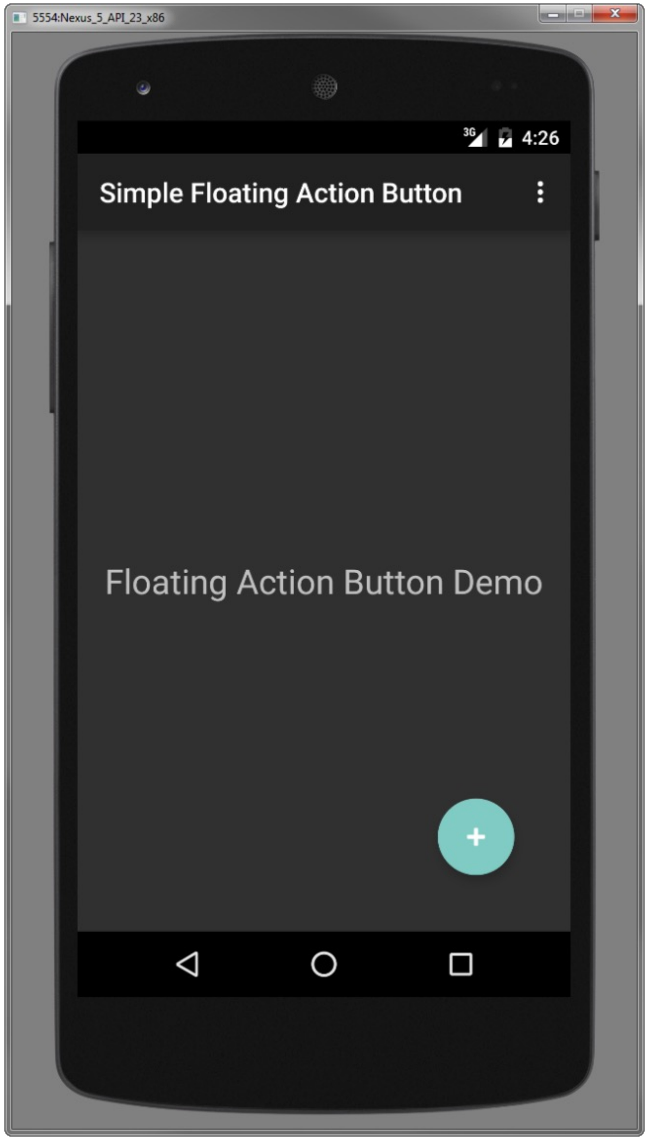
```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentEnd="true"
    android:layout_alignParentRight="true"
    android:layout_marginBottom="25dp"
    android:layout_marginEnd="25dp"
    android:layout_marginRight="25dp"
    android:clickable="true"
    android:contentDescription="@string/fab"
    android:elevation="6dp"
    android:src="@android:drawable/ic_input_add"
    android:tint="@android:color/white" />
```

## Floating Action Button

Now, use the Google library:

com.google.android.material.floatingactionbutton.FloatingActionButton

## Floating Action Button



# Actions Originating from Your Application’s Content

- You may need to make certain actions available from within your application’s content areas.
- If so, there are various UI elements that you can use for enabling actions, including:
  - Buttons
  - Check boxes
  - Radio buttons
  - Toggle buttons and switches
  - Spinners
  - Text fields
  - Seek bars
  - Pickers

## Single Activity Architecture

- Single Activity Architecture is a particular app architecture with a single activity and many fragments for different screens/parts of the app’s functionality
- When the user launches an app, a base/manager Activity is launched
- When the user navigates to another screen, the same Activity is reused, but it hosts a different fragment

## Single Activity Architecture

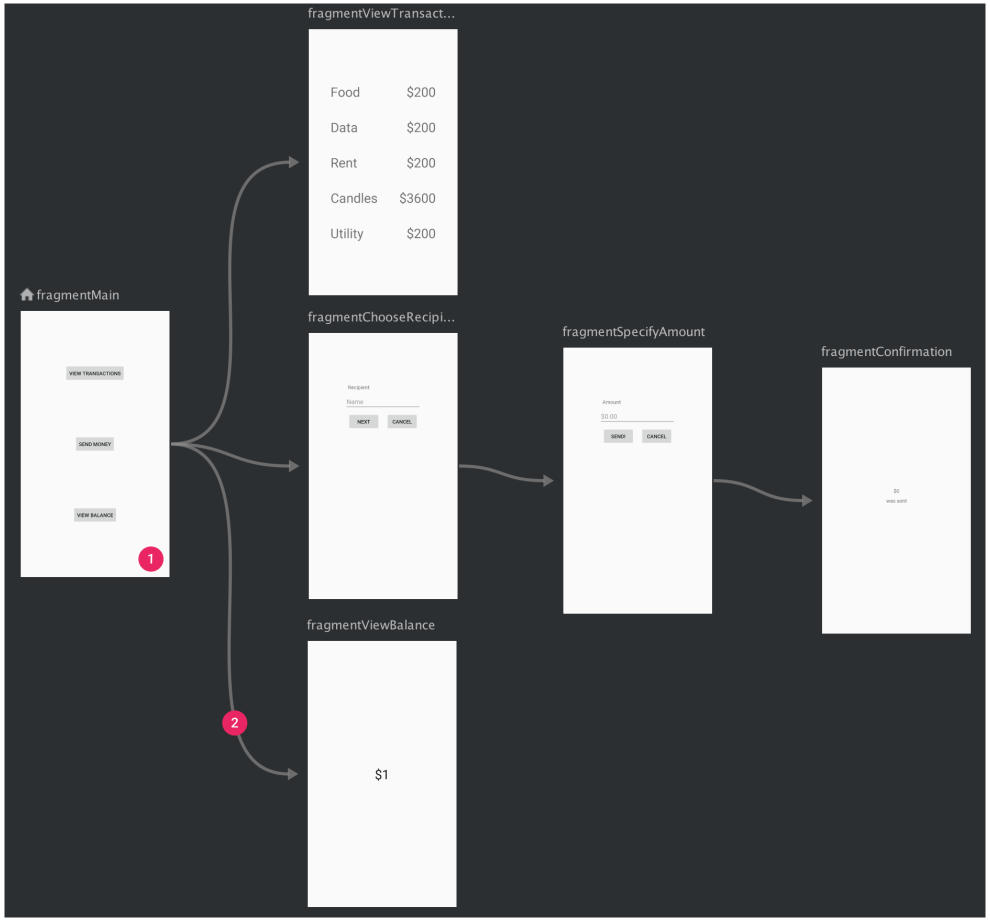
- Single Activity Architecture offers several advantages:
  - Screen transitions are faster
  - Portions of the screen, e.g., ActionBar or Navigation Drawer may be retained
  - Navigation is managed in one place
  - Simpler communication across screens
  - Simpler AndroidManifest file

## Navigation Component

- Android Jetpack library introduced a new Navigation component for implementing navigations to and from activities and fragments in an app
- It helps in providing a consistent navigation experience to users
- May include navigations triggered by buttons, activity bars, navigation drawers
- Provides a navigation graph for defining the overall navigation patterns in the app
- Learn more at:

https://developer.android.com/guide/navigation/migrate

# Navigation Component



An example of a navigation graph using the Graph Editor of Android Studio

From: developer.android.com

- Time permitting, we will talk about it later in this course

## Dialogs

- Dialogs are yet another way to present actions to your users.
- The best time to use a `Dialog` is when your application needs to confirm or acknowledge an action a user has taken that will permanently alter the user's data.
- If you allow users to edit their application data directly from within a `Dialog`, you should present actions within the `Dialog` to either confirm or deny the changes before actually committing them.

## Choosing Your Dialog Implementation

- Using the Fragment-based method, which was introduced in API Level 11 (Android 3.0), dialogs are managed using the `FragmentManager` class (`android.app.FragmentManager`).
- Dialogs become a special type of `Fragment` that must still be used within the scope of an `Activity` class, but its lifecycle is managed like that of any other `Fragment`.

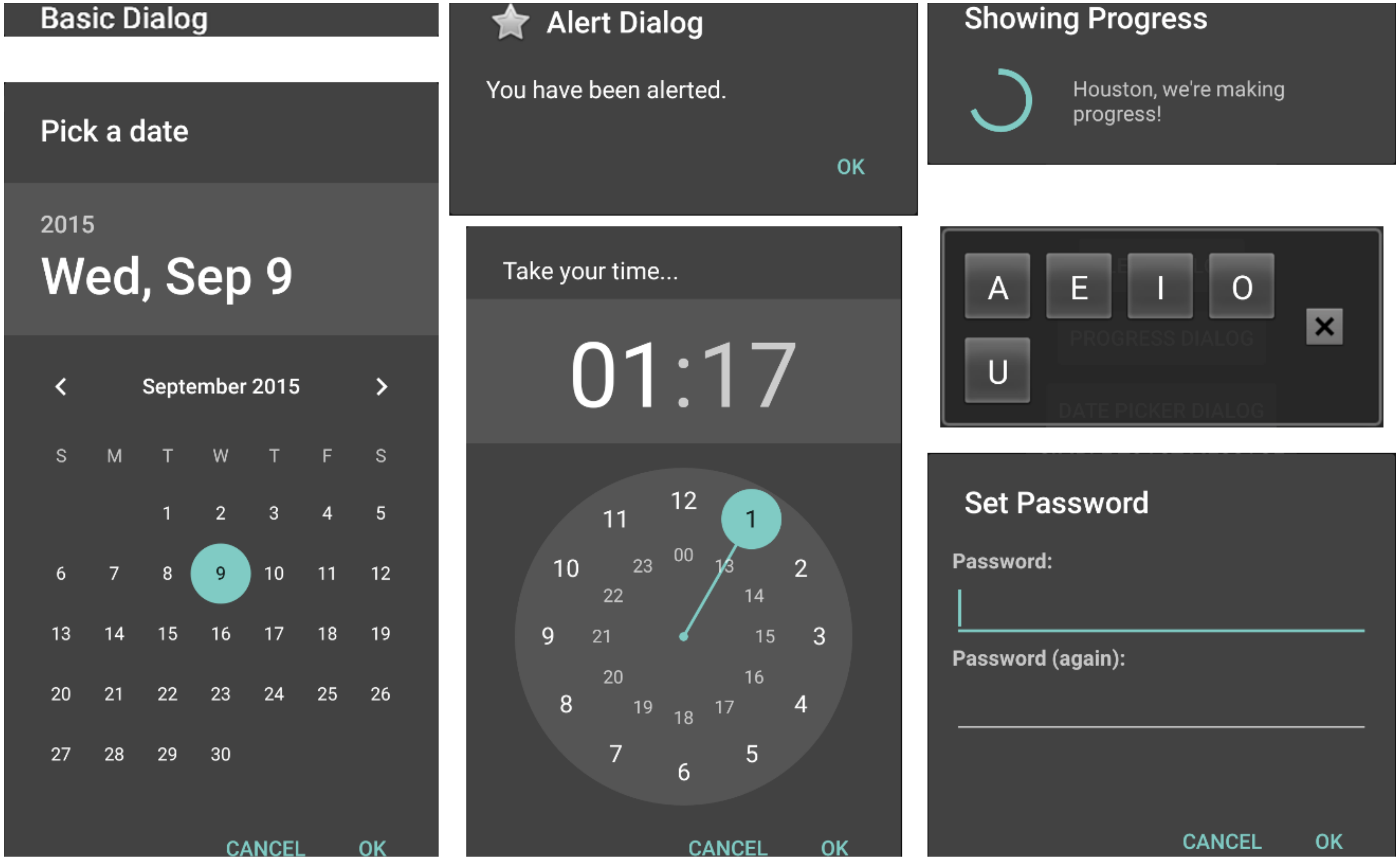
## Choosing Your Dialog Implementation

- This type of `Dialog` implementation works with the newest versions of the Android platform.
- Fragment-based dialogs are backward compatible with older devices as long as you incorporate the latest Android Support Package into your application.
- Fragment-based dialogs are the recommended choice for moving forward with the Android platform.

## Exploring the Different Types of Dialogs

- A number of different Dialog types are available within the Android SDK:
  - Toast
  - AlertDialog
  - ProgressDialog
  - DatePickerDialog
  - TimePickerDialog
  - CharacterPickerDialog
- You can create custom Dialog windows with your specific layout requirements.

# Exploring the Different Types of Dialogs


Basic Dialog

Alert Dialog

You have been alerted.

Showing Progress

Houston, we're making progress!

Pick a date

2015

Wed, Sep 9

Take your time...

01:17

Set Password

Password:

Password (again):

## Working with Dialogs and Dialog Fragments

- An `Activity` can use dialogs to organize information and react to user-driven events.
    - For example, an `Activity` might display a dialog informing the user of a problem or asking the user to confirm an action such as deleting a data record.
- Using dialogs for simple tasks helps keep the number of application activities manageable.

## Working with Dialogs and Dialog Fragments

- Most Activity classes should be "Fragment aware."
- In most cases, dialogs should be coupled with user-driven events within specific fragments.
- There is a special subclass of Fragment called a `DialogFragment` (`android.app.DialogFragment`) that can be used for this purpose. **Depracated. Now, use:**

  `androidx.fragment.app.DialogFragment`

- A `DialogFragment` is the best way to define and manage dialogs within your user interface.

## Tracing the Lifecycle of a Dialog and DialogFragment

- Each `Dialog` must be defined within the `DialogFragment` in which it is used.
- A `Dialog` may be launched once or used repeatedly.
- Understanding how a `DialogFragment` manages the `Dialog` lifecycle is important to implementing a `Dialog` correctly.

## Tracing the Lifecycle of a Dialog and DialogFragment

- The Android SDK manages dialog fragments in the same way that fragments are managed.
- We can also be sure that a `DialogFragment` follows nearly the same lifecycle as a `Fragment`.
- Critical additional callback:

  `onCreateDialog()`

  Override this callback to provide a Dialog for the fragment to manage and display.

# Tracing the Lifecycle of a Dialog and DialogFragment

- Other key methods that a `DialogFragment` must use to manage a `Dialog`:
  - `show()` method
    - Used to display the Dialog
  - `dismiss()` method
    - Used to stop showing the `Dialog`
  - `onDismiss()` method
    - Used to perform any additional code on dismiss

## Tracing the Lifecycle of a Dialog and DialogFragment

- Adding a `DialogFragment` with a `Dialog` to an Activity involves several steps:
  - Define a unique class that extends `DialogFragment`
    - You can define this class within your Activity, but if you plan to reuse this `DialogFragment` in other activities, define this class in a separate file.
    - This class must define a new `DialogFragment` class method that instantiates and returns a new instance of itself.

# Tracing the Lifecycle of a Dialog and DialogFragment

- Define a `Dialog` within the `DialogFragment` class.
  - Override the `onCreateDialog()` method and define your `Dialog` here.
  - There are various `Dialog` attributes you are able to define for your `Dialog` using methods such as `setTitle()`, `setMessage()`, or `setIcon()`.
  - Simply return the `Dialog` from this method.
- In your `Activity` class, create a new `DialogFragment` instance, and once you have the `DialogFragment` instance, show the `Dialog` using the `show()` method.

## Defining a DialogFragment

- A `DialogFragment` class can be defined within an `Activity` or within a `Fragment`.
- The type of `Dialog` you are creating will determine the type of data that you must supply to the `Dialog` definition inside the `DialogFragment` class.

## Setting Dialog Attributes

- A `Dialog` is not too useful without its contextual elements being set.
- One way of doing this is to define one or more of the attributes made available by the `Dialog` class.
- The base `Dialog` class and all of the `Dialog` subclasses define a `setTitle()` method.
- Setting the title usually helps a user determine for what the `Dialog` is used.
- The type of `Dialog` you are implementing determines the methods that are made available to you for setting different `Dialog` attributes.

## Showing a Dialog

- You can display any `Dialog` within an Activity by calling the `show()` method of the `DialogFragment` class on a valid `DialogFragment` object identifier.

## Dismissing a Dialog

- Most types of dialogs have automatic dismissal circumstances.
- If you want to force a `Dialog` to be dismissed, simply call the `dismiss()` method on the `Dialog` identifier.

# DialogFragment Example

```
public static class AlertDialogFragment extends DialogFragment {
        ….
        @Override
        public Dialog onCreateDialog(Bundle savedInstanceState) {
                AlertDialog.Builder alertDialog =
                    new AlertDialog.Builder(getActivity());
                alertDialog.setTitle("Alert Dialog");
                alertDialog.setMessage("You have been alerted.");
                alertDialog.setIcon(android.R.drawable.btn_star);
                alertDialog.setPositiveButton(android.R.string.ok,
                            new DialogInterface.OnClickListener() {
                          @Override
                        public void onClick(DialogInterface dialog, int which) {
                                 Toast.makeText(getActivity(), "Clicked OK!",
                              Toast.LENGTH_SHORT).show();
                                 return;
                        }
             });
             return alertDialog.create();
         }
}
```

## DialogFragment Example

```
public class SimpleFragDialogActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        // Handle Alert Dialog Button
        Button launchAlertDialog =
          (Button) findViewById(R.id.Button_AlertDialog);
        launchAlertDialog.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                DialogFragment newFragment =
                  AlertDialogFragment.newInstance();
                showDialogFragment(newFragment);
            }
        });
    }
    ….
}
```

## DialogFragment Example

```
public class SimpleFragDialogActivity extends Activity {
	….
	void showDialogFragment(DialogFragment newFragment) {
		newFragment.show(getFragmentManager(), null);
	}
}
```

## Working with Custom Dialogs

- When the `Dialog` types do not suit your purpose exactly, you can create a custom `Dialog`.
- One easy way to create a custom `Dialog` is to begin with an `AlertDialog` and use an `AlertDialog.Builder` class to override its default layout.

## Working with Custom Dialogs

- In order to create a custom `Dialog` this way, perform the following steps:
  - `Design` a custom layout resource to display in the `AlertDialog`.
  - Define the custom `Dialog` identifier in the `Activity`.
  - Use a `LayoutInflater` to inflate the custom layout resource for the `Dialog`.
  - Launch the `Dialog` using the `show()` method.

## Working with Dialog Fragments

- First, we must import the support version of the `DialogFragment` class:
  - `android.support.v4.app.DialogFragment`

  Now: `androidx.fragment.app.DialogFragment`
- Then, just as before, you need to implement your own `DialogFragment` class.
  - This class simply needs to be able to return an instance of the object that is fully configured and needs to implement the `onCreateDialog` method, which returns the fully configured `AlertDialog`, much as it did using the legacy method.

## Working with Dialog Fragments

```
public class MyAlertDialogFragment extends DialogFragment {
    public static MyAlertDialogFragment
            newInstance(String fragmentNumber) {
        MyAlertDialogFragment newInstance =
            new MyAlertDialogFragment();
        Bundle args = new Bundle();
        args.putString("fragnum", fragmentNumber);
        newInstance.setArguments(args);
        return newInstance;
    }
    ….
}
```

## Working with Dialog Fragments

```
public class MyAlertDialogFragment extends DialogFragment {
    ….
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        final String fragNum = getArguments().getString("fragnum");
        AlertDialog.Builder alertDialog = new AlertDialog.Builder(getActivity());
        alertDialog.setTitle("Alert Dialog");
        alertDialog.setMessage("This alert brought to you by " + fragNum );
        alertDialog.setIcon(android.R.drawable.btn_star);
        alertDialog.setPositiveButton(android.R.string.ok,
                new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                ((SimpleFragDialogActivity) getActivity()).doPositiveClick(fragNum);
                return;
            }   });
        return alertDialog.create();
    }
}
```

# Working with Dialog Fragments

- Now that you have defined your `DialogFragment`, you can use it within your `Activity` like any `Fragment`.
  - But this time you must use the support version of the `FragmentManager` by calling `getSupportFragmentManager()`.
- In your Activity file, you need to import two support classes for this implementation to work:
  - `androidx.fragment.app.DialogFragment`
  - `androidx.fragment.app.FragmentActivity`

## Working with Dialog Fragments

- Be sure to extend your `Activity` class from `FragmentActivity`, and not Activity as in the previous example, or your code will not work.
- The `FragmentActivity` class is a special class that makes fragments available with the Support Package.

## Working with Dialog Fragments

```
public class SupportFragDialogActivity extends FragmentActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Button launchAlertDialog = (Button) findViewById(R.id.Button_AlertDialog);
        launchAlertDialog.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                String strFragmentNumber = "Fragment Instance One";
                DialogFragment newFragment =
                    MyAlertDialogFragment.newInstance(strFragmentNumber);
                showDialogFragment(newFragment, strFragmentNumber);
            }
        });
    }
    ….
}
```

# Working with Dialog Fragments

```
public class SupportFragDialogActivity extends
FragmentActivity {
….
    void showDialogFragment(DialogFragment newFragment,
            String strFragmentNumber) {
        newFragment.show(getSupportFragmentManager(),
            strFragmentNumber);
    }

    public void doPositiveClick(String strFragmentNumber) {
        Toast.makeText(getApplicationContext(),
            "Clicked OK! (" + strFragmentNumber + ")",
            Toast.LENGTH_SHORT).show();
    }
}
```

## Working with Dialog Fragments

- `DialogFragment` instances can be traditional pop-ups (as shown in the previous slide) or they can be embedded like any other `Fragment`.
- You may want to embed a `Dialog` in these circumstances:
  - You've created a picture gallery application and implemented a custom `Dialog` that displays a larger image when you click a thumbnail.

## Working with Dialog Fragments

- On small-screen devices, you might want this to be a pop-up `Dialog`.
- But on a tablet or TV, you might have the screen space to show the larger graphic off to the right or below the thumbnails.
- This would be a good opportunity to take advantage of code reuse and simply embed your `Dialog`.

## Summary

- We have covered different navigation scenarios, including entry, lateral, descendant, back, ancestral, and external navigation.
- We have learned about different Android design patterns for navigation such as targets, tabs, navigation drawer, and master detail flow.
- We have learned how menus, action bars, dialogs, and other user interface controls can influence action from users.
- We have learned that Fragment-based dialogs are the recommended Dialog implementation.

## Summary

- We have learned about the different types of dialogs available in the Android SDK.
- We have learned that dialogs are defined within dialog fragments.
- We are now able to define a DialogFragment class.
- We have learned how to create custom dialogs.
- We are now able to implement dialog fragments that work on devices all the way back to API Level 4.

## References and More Information

- Android Design: Pure Android: "Confirming & Acknowledging":
  - http://d.android.com/design/patterns/confirming-acknowledging.html
- Android Design: Pure Android: "Notifications":
  - http://d.android.com/design/patterns/notifications.html
- Android Training: "Designing Effective Navigation":
  - http://d.android.com/training/design-navigation/index.html
- Android Training: "Implementing Effective Navigation":
  - http://d.android.com/training/implementing-navigation/index.html
- Android Training: "Notifying the User":
  - http://d.android.com/training/notify-user/index.html
- Android Training: "Managing the System UI":
  - http://d.android.com/training/system-ui/index.html
- Android API Guides: "Dialogs":
  - http://d.android.com/guide/topics/ui/dialogs.html
- Android DialogFragment Reference: "Selecting Between Dialog or Embedding":
  - http://d.android.com/reference/android/app/DialogFragment.html#DialogOrEmbed