Fragments

Xamarin Android Level 2, Chapter 2

Xamarin Inc.

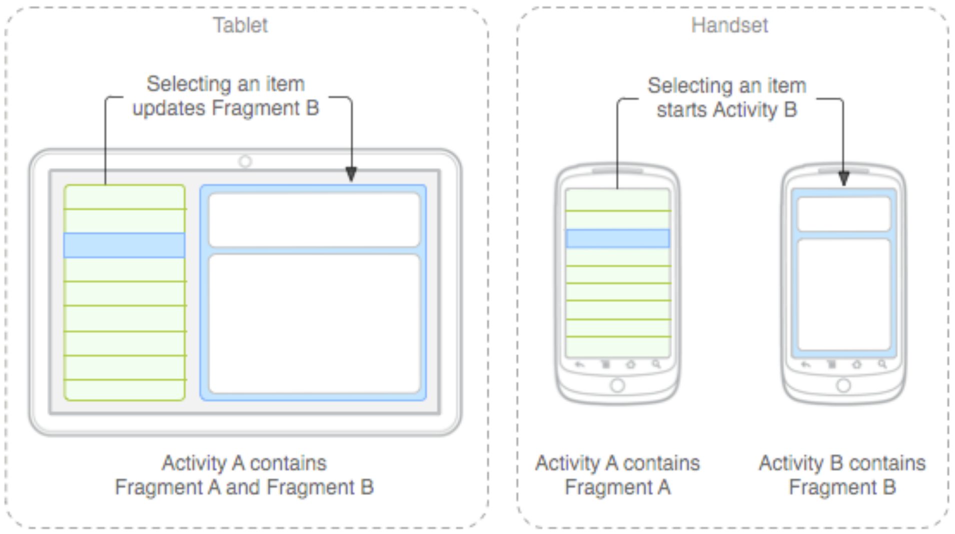
Overview

The larger screen sizes found on most tablets added an extra layer of complexity to Android development—a layout designed for the small screen does not necessarily work as well for larger screens, and vice-versa. In order to reduce the number of complications that this introduced, Android 3.0 added two new features, *Fragments* and *Support Packages*.

Fragments can be thought of as user interface modules. They let the developer divide up the user interface into isolated, reusable parts that can be run in separate Activities. At run time, the Activities themselves will decide which Fragments to use.

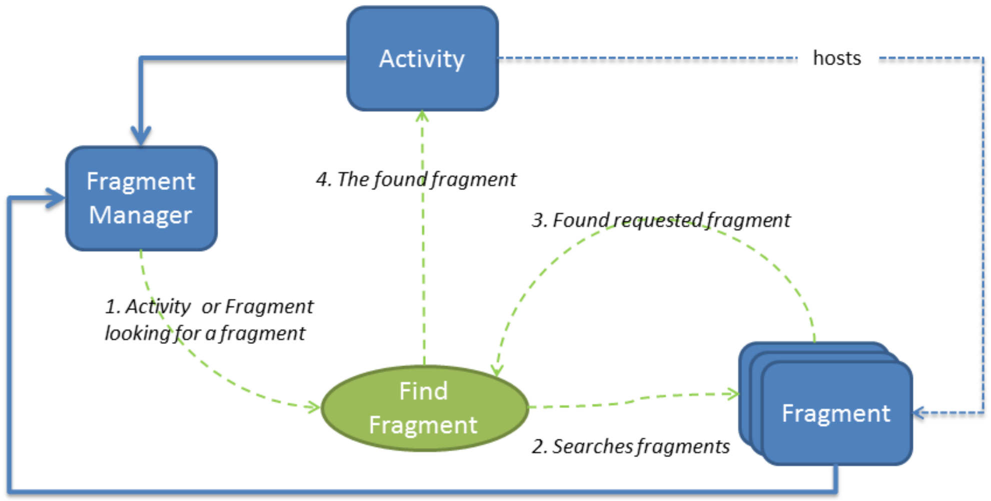
Support Packages were originally called *Compatibility Libraries* and allowed Fragments to be used on devices that run versions of Android prior to Android 3.0 (API Level 11).

For example, the image below illustrates how a single application uses Fragments across varying device form factors.


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

Fragment A contains a list, while *Fragment B* contains details for an item selected in that list. When the application is run on a tablet, it can display both Fragments on the same Activity. When the same application is run on a handset (with its smaller screen size), the Fragments are hosted in two separate Activities. Fragment A and Fragment B are the same on both form factors, but the Activities that host them are different.

To help an Activity coordinate and manage all these Fragments, Android introduced a new class called the *FragmentManager*. Each Activity has its own instance of a `FragmentManager` for adding, deleting, and finding hosted Fragments. The following diagram illustrates the relationship between Fragments and Activities:



In some regards, Fragments can be thought of as composite controls or as mini-Activities. They bundle up pieces of UI into reusable modules that can then be used independently by developers in Activities. A Fragment does have a view hierarchy—just like an Activity—but, unlike an Activity, it can be shared across screens. Views differ from Fragments in that Fragments have their own lifecycle; views do not.

While the Activity is a host to one or more Fragments, it is not directly aware of the Fragments themselves. Likewise, Fragments are not directly aware of other Fragments in the hosting Activity. However, Fragments and Activities are aware of the `FragmentManager` in their Activity. By using the `FragmentManager`, it is possible for an Activity or a Fragment to obtain a reference to a specific instance of a Fragment, and then call methods on that instance. In this way, the Activity or Fragments can communicate and interact with other Fragments.

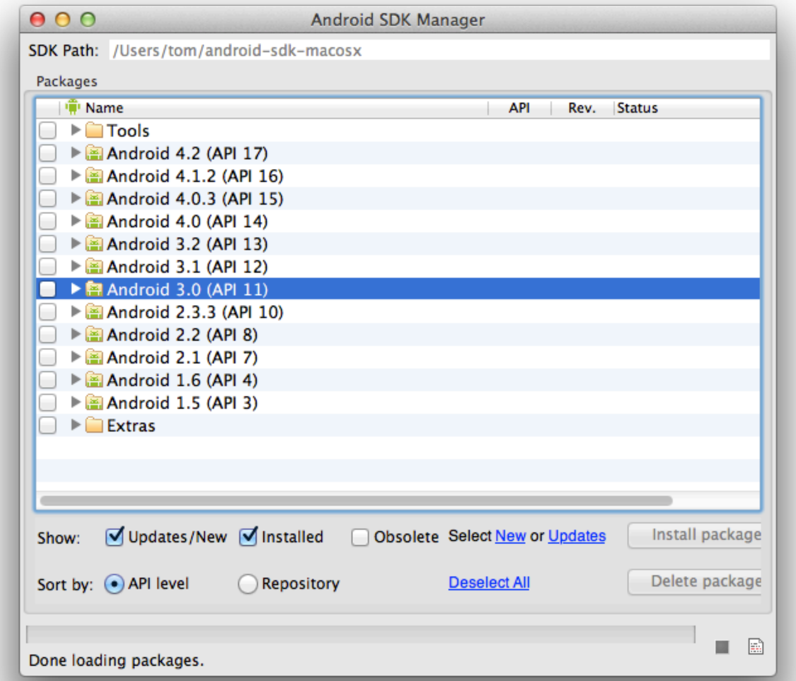
This chapter is an introduction to using Fragments, including:

- ➔ Creating Fragments – How to create a basic Fragment and key methods that must be implemented.
- ➔ Fragment Management and Transactions – How to manipulate Fragments at run time.
- ➔ Android Support Package – How to use the libraries that allow Fragments to be used on older versions of Android.

For a more in depth look at fragments, refer to Xamarin's documentation on Fragments and the Fragments Walkthrough.

Requirements

Fragments are available in the Android SDK starting with API level 11 (Android 3.0), as shown in the following screenshot:


Android SDK Manager

SDK Path: /Users/tom/android-sdk-macosx

Packages

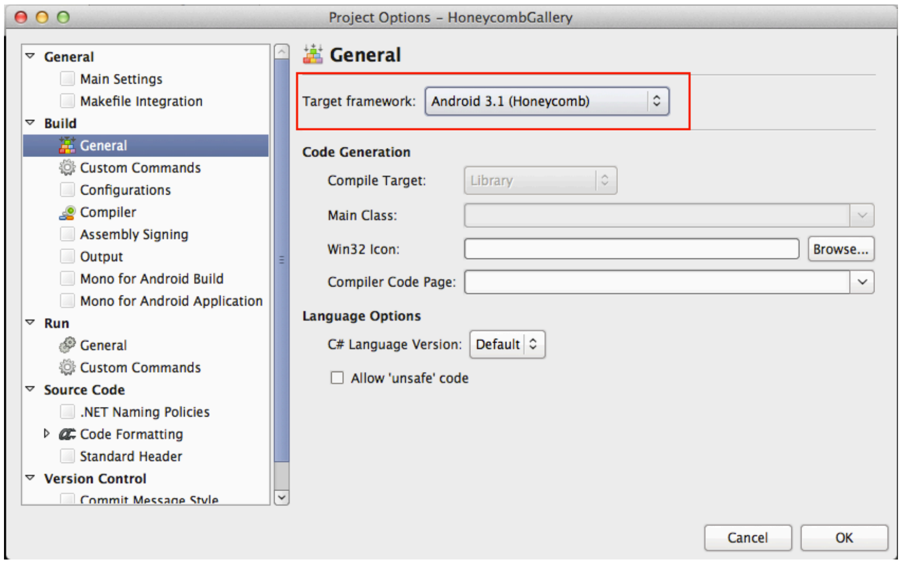
Name	API	Rev.	Status
Tools			
Android 4.2 (API 17)			
Android 4.1.2 (API 16)			
Android 4.0.3 (API 15)			
Android 4.0 (API 14)			
Android 3.2 (API 13)			
Android 3.1 (API 12)			
Android 3.0 (API 11)			
Android 2.3.3 (API 10)			
Android 2.2 (API 8)			
Android 2.1 (API 7)			
Android 1.6 (API 4)			
Android 1.5 (API 3)			
Extras			

Show: [✓] Updates/New [✓] Installed [ ] Obsolete Select New or Updates

Sort by: (●) API level ( ) Repository

Done loading packages.

Fragments are available in Mono for Android 4.0 and higher. A Mono for Android application must target at least API level 11 (Android 3.0) or higher in order to use Fragments. The Target Framework may be set in the Project Options as shown below:


Project Options – HoneycombGallery

General

Target framework: Android 3.1 (Honeycomb)

Code Generation

Compile Target: Library

Main Class:

Win32 Icon:

Compiler Code Page:

Language Options

C# Language Version: Default

[ ] Allow 'unsafe' code

It is possible to use Fragments in older versions of Android by using the Android Support Package and Mono for Android 4.2 or higher. How to do this will be covered in more detail further on in this document.

Using Fragments

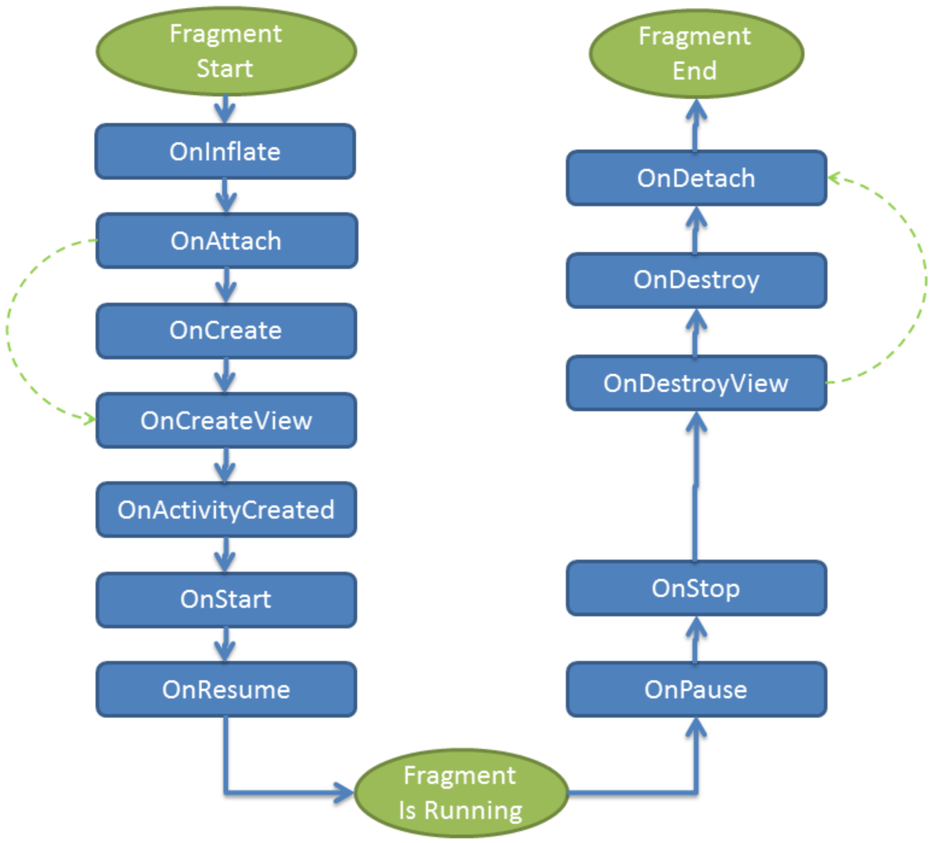
To create a Fragment, a class must inherit from `Android.App.Fragment` and then override the `OnCreateView` method. `OnCreateView` will be called by the hosting Activity when it is time to put the Fragment on the screen, and will return a `View`. The following code is one example of creating the view for a fragment:

```
public override View OnCreateView(LayoutInflater inflater, ViewGroup
container, Bundle savedInstanceState)
{
    return inflater.Inflate(Resource.Layout.Example_Fragment, container,
false);
}
```

In the example above, a layout file is inflated and then added as a child to the ViewGroup specified by the parameter `container`.

Fragment Lifecycle

Fragments have their own lifecycle that is somewhat independent of, but still affected by, the lifecycle of the hosting Activity. For example, when an Activity pauses, all of its associated Fragments are paused. The following diagram outlines the lifecycle of the Fragment.



The table below shows the flow of the various callbacks in the lifecycle of a Fragment as it is being created:

Lifecycle Method	Activity State
`OnInflate()`	Called when the Fragment is being created as part of a view layout. This may be called immediately after the Fragment is created declaratively from an XML layout file. The Fragment is not associated with its Activity yet, but the `Activity`, `Bundle`, and `AttributeSet` from the view hierarchy are passed in as parameters. This method is best used for parsing the `AttributeSet` and for saving the attributes that might be used later by the Fragment.
`OnAttach()`	Called after the Fragment is associated with the Activity. This is the first method to be run when the Fragment is ready to be used. In general, Fragments should not implement a constructor or override the default constructor. Any components that are required for the Fragment should be initialized in this method.
`OnCreate()`	Called by the Activity to create the Fragment. When this method is called, the view hierarchy of the hosting Activity may not be completely instantiated, so the Fragment should not rely on any parts of the Activity's view hierarchy until later on in the Fragment's lifecycle. For example, do not use this method to perform any tweaks or adjustments to the UI of the application. This is the earliest time at which the Fragment may begin gathering the data that it needs. The Fragment is running in the UI thread at this point, so avoid any lengthy processing, or perform that processing on a background thread. This method may be skipped if `SetRetainInstance(true)` is called. This alternative will be described in more detail below.
`OnCreateView()`	Creates the view for the Fragment. This method is called once the Activity's `OnCreate()` method is complete. At this point, it is safe to interact with the view hierarchy of the Activity. This method should return the view that will be used by the Fragment.

`OnActivityCreated()`	Called after `Activity.OnCreate` has been completed by the hosting Activity. Final tweaks to the user interface should be performed at this time.
`OnStart()`	Called after the containing Activity has been resumed. This makes the Fragment visible to the user. In many cases, the Fragment will contain code that would otherwise be in the `OnStart()` method of an Activity.
`OnResume()`	This is the last method called before the user can interact with the Fragment. An example of the kind of code that should be performed in this method would be enabling features of a device that the user may interact with, such as the camera that the location services. Services such as these can cause excessive battery drain, though, and an application should minimize their use in order to preserve battery life.

The next table shows the lifecycle methods that are called as a Fragment is being destroyed:

Lifecycle Method	Activity State
`OnPause()`	The user is no longer able to interact with the Fragment. This situation exists because some other Fragment operation is modifying this Fragment, or the hosting Activity is paused. It is possible that the Activity hosting this Fragment might still be visible, that is, the Activity in focus is partially transparent or does not occupy the full screen. When this method is called, it’s the first indication that the user is leaving the Fragment. The Fragment should save any changes.
`OnStop()`	The Fragment is no longer visible. The host Activity may be stopped, or a Fragment operation is modifying it in the Activity. This callback serves the same purpose as `Activity.OnStop`.

`OnDestroyView()`	This method is called to clean up resources associated with the view. This is called when the view associated with the Fragment has been destroyed.
`OnDestroy()`	This method is called when the Fragment is no longer in use. It is still associated with the Activity, but the Fragment is no longer functional. This method should release any resources that are in use by the Fragment, such as a SurfaceView that might be used for a camera. This method may be skipped if `SetRetainInstance(true)` is called. This alternative will be described in more detail below.
`OnDetach()`	This method is called just before the Fragment is no longer associated with the Activity. The view hierarchy of the Fragment no longer exists, and all resources that are used by the Fragment should be released at this point.

Adding a Fragment to an Activity

There are two ways that a Fragment may be hosted inside an Activity:

- ➔ Declaratively – Fragments can be used declaratively within `.axml` layout files by using the `<Fragment>` tag.
- ➔ Programmatically – Fragments can also be instantiated dynamically and then added to a fragment by using the `FragmentManager` class's API.

Programmatic usage via the `FragmentManager` class will be discussed later in this chapter.

USING A FRAGMENT DECLARATIVELY

Adding a Fragment inside the layout requires using the `<fragment>` tag and then identifying the Fragment by providing either the `class` attribute or the `android:name` attribute. The following snippet shows how to use the `class` attribute to declare a `fragment`:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment class="com.xamarin.sample.fragments.TitlesFragment"
            android:id="@+id/titles_fragment"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent" />
```

This next snippet shows how to declare a `fragment` by using the `android:name` attribute to identify the Fragment class :

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<fragment android:name="com.xamarin.sample.fragments.TitlesFragment"
            android:id="@+id/titles_fragment"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent" />
```

When the Activity is being created, Android will instantiate each Fragment specified in the layout file. The Activity will then call `OnCreateView` on each Fragment and display the View that the Fragment has created.

Fragments that are declaratively added to an Activity are static and will remain on the Activity until it is destroyed; it is not possible to dynamically replace or remove such a Fragment during the lifetime of the Activity to which it is attached.

Each Fragment must be assigned a unique identifier:

- ➔ android:id – As with other UI elements in a layout file, this is a unique ID.
- ➔ android:tag – This attribute is a unique string.

If neither of the previous two methods is used, then the Fragment will assume the ID of the container view. In the following example where neither `android:id` nor `android:tag` is provided, Android will assign the ID `fragment_container` to the Fragment:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
               android:id="+@id/fragment_container"
               android:orientation="horizontal"
               android:layout_width="match_parent"
               android:layout_height="match_parent">

        <fragment class="com.example.android.apis.app.TitlesFragment"
               android:layout_width="match_parent"
               android:layout_height="match_parent" />
</LinearLayout>
```

Note: Android does not allow for uppercase characters in package names; it will throw an exception when trying to inflate the view if a package name contains an uppercase character. However, Mono for Android is more forgiving, and will tolerate uppercase characters in the namespace.

For example, both of the following snippets will work with Mono for Android. However, the second snippet will cause an `android.view.InflateException` to be thrown by a pure Java-based Android application.

```
<fragment                    class="com.example.DetailsFragment"
android:id="@+id/fragment_content"
android:layout_width="match_parent"
android:layout_height="match_parent" />
```

OR

```
<fragment                    class="Com.Example.DetailsFragment"
android:id="@+id/fragment_content"
android:layout_width="match_parent"
android:layout_height="match_parent" />
```

Managing Fragments

To help with managing Fragments and to programmatically add Fragments, Android provides the `FragmentManager` class. Each Activity has an instance of `Android.App.FragmentManager` that will find or dynamically change its Fragments. Each set of these changes is known as a *transaction*, and is performed by using one of the APIs contained in the class `Android.App.FragmentTransation`, which is managed by the `FragmentManager`. An Activity may start a transaction like this:

```
FragmentTransaction fragmentTx = this.FragmentManager.BeginTransaction();
```

These changes to the Fragments are performed in the `FragmentTransaction` instance by using methods such as `Add()`, `Remove()`, and `Replace()`. The changes are then applied by using `Commit()`. The changes in a transaction are not performed immediately. Instead, they are scheduled to run on the Activity's UI thread as soon as possible.

The following example shows how to add a Fragment to an existing container:

```
 // Create a new fragment and a transaction.
FragmentTransaction fragmentTx = this.FragmentManager.BeginTransaction();
DetailsFragment aDifferentDetailsFrag = new DetailsFragment();

// The fragment will have the ID of Resource.Id.fragment_container.
fragmentTx.Add(Resource.Id.fragment_container, aDifferentDetailsFrag);

// Commit the transaction.
fragmentTx.Commit();
```

It's possible to save the Fragment transactions to the Activity's back stack by making a call to `FragmentTransaction.AddToBackStack()`. This allows the user to navigate backwards through Fragment changes when the **Back** button is pressed. Without a call to this method, Fragments that are removed will be destroyed and will be unavailable if the user navigates back through the Activity.

The following example shows how to use the `AddToBackStack` method of a `FragmentTransaction` to replace one Fragment, while preserving the state of the first Fragment on the back stack:

```
// Create a new fragment and a transaction.
FragmentTransaction fragmentTx = this.FragmentManager.BeginTransaction();
DetailsFragment aDifferentDetailsFrag = new DetailsFragment();

// Replace the fragment that is in the View fragment_container (if
applicable).
fragmentTx.Replace(Resource.Id.fragment_container, aDifferentDetailsFrag);

// Add the transaction to the back stack.
fragmentTx.AddToBackStack(null);

// Commit the transaction.
fragmentTx.Commit();
```

COMMUNICATING WITH FRAGMENTS

The *FragmentManager* provides two methods to help find specific instances of Fragments:

- ➔ FindFragmentById – This method will find a Fragment by using the ID that was specified in the layout file or the container ID when the Fragment was added as part of a transaction.
- ➔ FindFragmentByTag – This method is used to find a Fragment that has a tag that was provided in the layout file or that was added in a transaction.

Both Fragments and Activities reference the `FragmentManager`, so the same techniques are used to communicate back and forth between them. An application may find a reference Fragment by using one of these two methods, cast that reference to the appropriate type, and then directly call methods on the Fragment. The following snippet provides an example:

It is also possible for the Activity to use the `FragmentManager` to find Fragments:

```
var emailList =
FragmentManager.FindFragmentById<EmailListFragment>(Resource.Id.email_list
_fragment);
emailList.SomeCustomMethod(parameter1, parameter2);
```

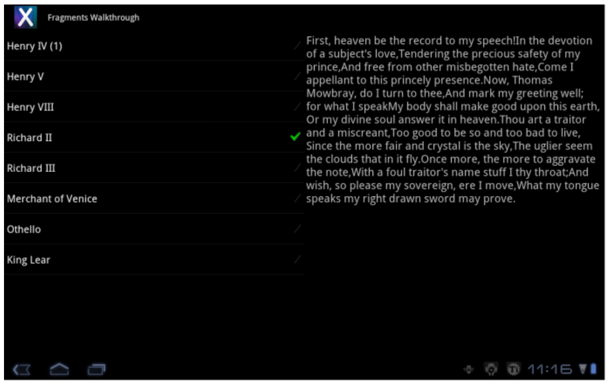
COMMUNICATING WITH THE ACTIVITY

It is possible for a Fragment to use the `Fragment.Activity` property to reference its host. By casting the Activity to a more specific type, it is possible for an Activity to call methods and properties on its host, as shown in the following example:

```
var myActivity = (MyActivity) this.Activity;
myActivity.SomeCustomMethod();
```

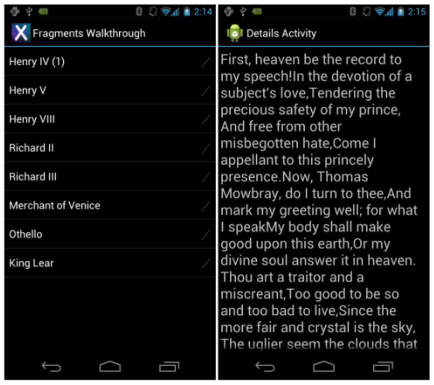
Displaying Lists in a Fragment

The `ListFragment` is a specialized subclass of the `Fragment` class. It is very similar in concept and functionality to the `ListActivity`; it is a wrapper that hosts a ListView in a Fragment. The image below shows a `ListFragment` running on a tablet and a phone:



First, heaven be the record to my speech!In the devotion of a subject's love,Tendering the precious safety of my prince,And free from other misbegotten hate,Come I appellant to this princely presence.Now, Thomas Mowbray, do I turn to thee,And mark my greeting well; for what I speakMy body shall make good upon this earth, Or my divine soul answer it in heaven.Thou art a traitor and a miscreant,Too good to be so and too bad to live, Since the more fair and crystal is the sky,The uglier seem the clouds that in it fly.Once more, the more to aggravate the note,With a foul traitor's name stuff I thy throat;And wish, so please my sovereign, ere I move,What my tongue speaks my right drawn sword may prove.

Tablet



First, heaven be the record to my speech!In the devotion of a subject's love,Tendering the precious safety of my prince, And free from other misbegotten hate,Come I appellant to this princely presence.Now, Thomas Mowbray, do I turn to thee,And mark my greeting well; for what I speakMy body shall make good upon this earth,Or my divine soul answer it in heaven. Thou art a traitor and a miscreant,Too good to be so and too bad to live,Since the more fair and crystal is the sky, The uglier seem the clouds that

Phone

BINDING DATA WITH THE LISTADAPTER

The `ListFragment` class already provides a default layout, so it is not necessary to override `OnCreateView` to display the contents of the `ListFragment`. The `ListView` is bound to data by using a `ListAdapter` implementation. The following example shows how this could be done by using a simple array of strings:

```
public override void OnActivityCreated(Bundle savedInstanceState)
{
    base.OnActivityCreated(savedInstanceState);
    string[] values = new[] { "Android", "iPhone", "WindowsMobile",
               "Blackberry", "WebOS", "Ubuntu", "Windows7", "Max OS X",
               "Linux", "OS/2" };
    this.ListAdapter = new ArrayAdapter<string>(Activity,
Android.Resource.Layout.SimpleExpandableListItem1, values);
}
```

When setting the `ListAdapter`, it is important to use the `ListFragment.ListAdapter` property, and not the `ListView.ListAdapter` property. Using `ListView.ListAdapter` will cause important initialization code to be skipped.

RESPONDING TO USER SELECTION

To respond to user selections, an application must override the `OnListItemClick` method. The following example shows one such possibility:

```
 public override void OnListItemClick(ListView l, View v, int index, long
id)
{
    // We can display everything in place with fragments.
    // Have the list highlight this item and show the data.
    ListView.SetItemChecked(index, true);

    // Check what fragment is shown, replace if needed.
    var details =
FragmentManager.FindFragmentById<DetailsFragment>(Resource.Id.details);
    if (details == null || details.ShownIndex != index)
    {
        // Make new fragment to show this selection.
        details = DetailsFragment.NewInstance(index);

        // Execute a transaction, replacing any existing
        // fragment with this one inside the frame.
        var ft = FragmentManager.BeginTransaction();
        ft.Replace(Resource.Id.details, details);
        ft.SetTransition(FragmentTransit.FragmentFade);
        ft.Commit();
    }
}
```

In the code above, when the user selects an item in the `ListFragment`, a new Fragment is displayed in the hosting Activity, showing more details about the item that was selected.

Providing Backwards Compatibility with the Android Support Package

The usefulness of Fragments would be limited without backwards compatibility with pre-Android 3.0 (API Level 11) devices. To provide this capability, Google introduced the Android Support Package (originally called the *Android Compatibility Library* when it was released) which backports some of the APIs from newer versions of Android to older versions of Android. It is the Android Support Package that enables devices running Android 1.6 (API level 4) to Android 2.3.3. (API level 10).

> **Note:** Not all fragment subclasses are available through the Android Support Package. For more details refer to Xamarin's documentation on Fragments and the Fragments Walkthrough

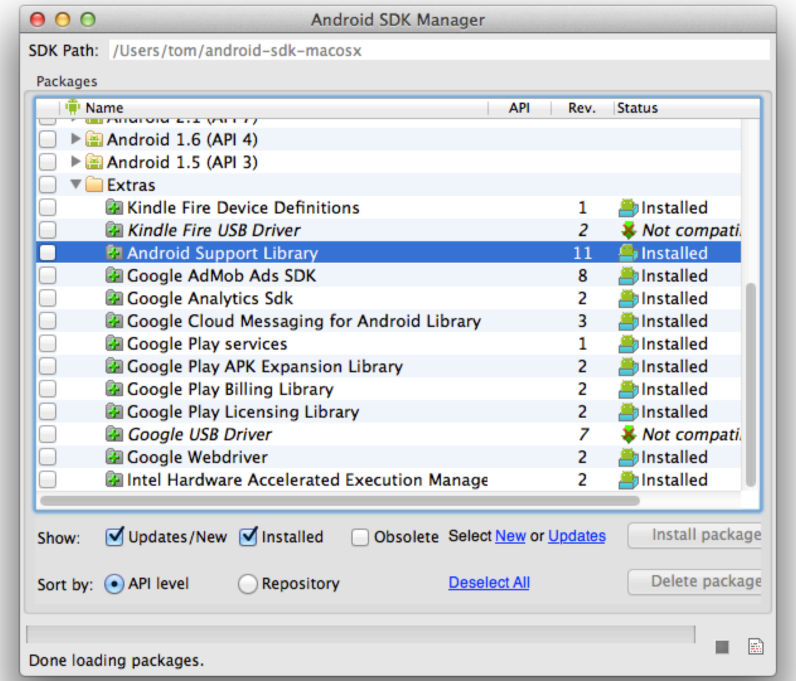
Adding the Support Package

The Android Support Package is not automatically added to a Mono for Android application. In order to add this support package, some or all of the following steps must be performed:

- ➔ **Download the Support Package** – This only needs to be performed once, or when updates are issued by Google to the support package.
- ➔ **Reference Mono.Android.Support.v4** – This is the Mono for Android wrapper that adds the support package to the project.
- ➔ **Add the support package JAR file** – This ensures that the Support APIs are available to the application. This is not necessary in Mono for Android 4.4 or higher.

DOWNLOADING THE SUPPORT PACKAGE

To add the Android Support Package to a Mono for Android application, it must first be downloaded from the Android SDK Manager, under the Extras menu:



SDK Path: /Users/tom/android-sdk-macosx

Packages

Name	API	Rev.	Status
☐ ▶ Android 1.6 (API 4)			
☐ ▶ Android 1.5 (API 3)			
☐ ▼ Extras			
☐ Kindle Fire Device Definitions		1	Installed
☐ *Kindle Fire USB Driver*		*2*	*Not compati*
☐ Android Support Library		11	Installed
☐ Google AdMob Ads SDK		8	Installed
☐ Google Analytics Sdk		2	Installed
☐ Google Cloud Messaging for Android Library		3	Installed
☐ Google Play services		1	Installed
☐ Google Play APK Expansion Library		2	Installed
☐ Google Play Billing Library		2	Installed
☐ Google Play Licensing Library		2	Installed
☐ *Google USB Driver*		*7*	*Not compati*
☐ Google Webdriver		2	Installed
☐ Intel Hardware Accelerated Execution Manage		2	Installed

Show: ☑ Updates/New ☑ Installed ☐ Obsolete

Sort by: ● API level ○ Repository

Done loading packages.

The necessary libraries may be found in the Android SDK directory `<sdk>/extras/android/support`. Each support library will be stored in a subdirectory for each version that is supported. For example, the Support Library that supports Android 1.6 (API level 4) and higher will be found in the folder `<sdk>/extras/android/support/v4`. Older versions of Android require the v4 library of the Android Support Package.

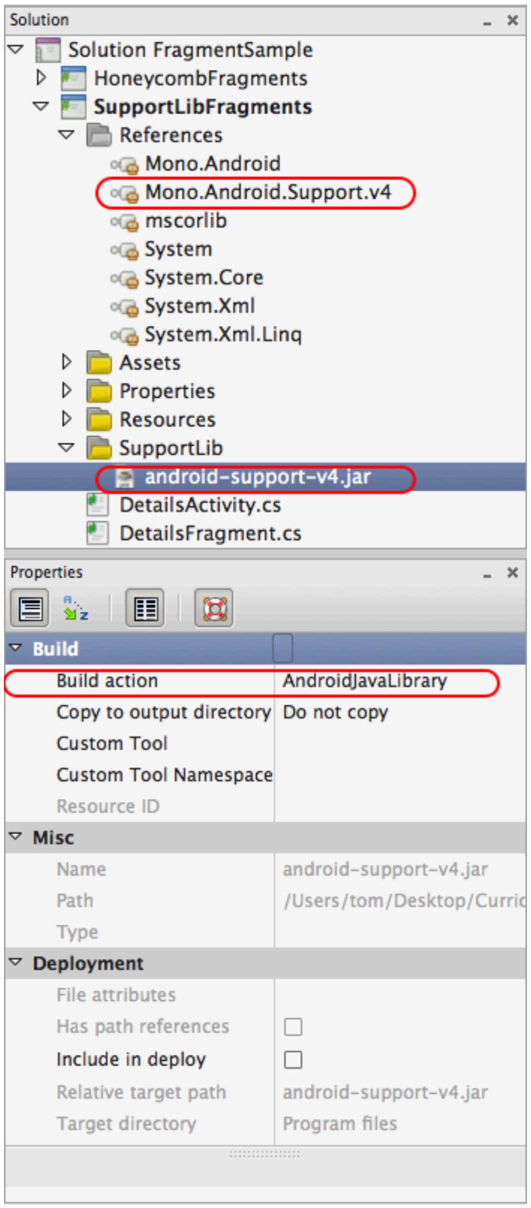
REFERENCE MONO.ANDROID.SUPPORT.V4

The next step in this process is to add a reference to the `Mono.Android.Support.v4` assembly.

ADDING THE V4 LIBRARIES TO A MONO FOR ANDROID PROJECT

This step is not necessary in Mono for Android 4.4 or higher. Earlier version of Mono for Android require the file `android-support-v4.jar` be added to the project. Create a new folder in the project called `SupportLib`, and copy the file `android-support-v4.jar` into it. Mono for Android will automatically set the **BuildAction** of the file to **AndroidJavaLibrary**.

The image below shows what the structure of the Mono for Android project should be like after following these steps:


Solution

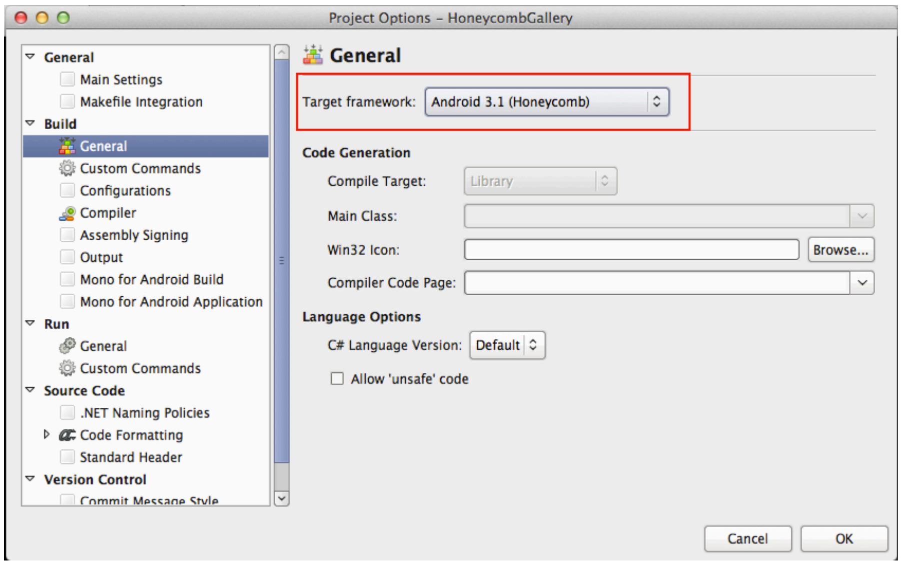
Solution FragmentSample
HoneycombFragments
SupportLibFragments
References
Mono.Android
Mono.Android.Support.v4
mscorlib
System
System.Core
System.Xml
System.Xml.Linq
Assets
Properties
Resources
SupportLib
android-support-v4.jar
DetailsActivity.cs
DetailsFragment.cs

Properties

Build	
Build action	AndroidJavaLibrary
Copy to output directory	Do not copy
Custom Tool	
Custom Tool Namespace	
Resource ID	
Misc	
Name	android-support-v4.jar
Path	/Users/tom/Desktop/Curric
Type	
Deployment	
File attributes	
Has path references	☐
Include in deploy	☐
Relative target path	android-support-v4.jar
Target directory	Program files

After these steps have been performed, it becomes possible to use Fragments in earlier versions of Android. The Fragment APIs will work the same now in these earlier versions, with the following exceptions:

➔ Change the Target Android Version – The application no longer needs to target Android 3.0 or higher, as shown below:



Target framework: Android 3.1 (Honeycomb)

Compile Target: Library

C# Language Version: Default

☐ Allow 'unsafe' code

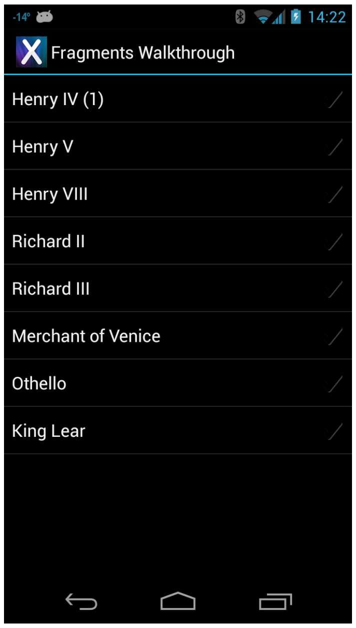
- ➔ Extend FragmentActivity – The Activities that are hosting Fragments must now inherit from `Android.App.FragmentActivity`, and not from `Android.App.Activity`.
- ➔ Update Namespaces – Classes that inherit from `Android.App.Fragment` must now inherit from `Android.Support.v4.Fragment`. Remove the using statement "`using Android.App;`" at the top of the source code file and replace it with "`using Android.Support.v4.App`".
- ➔ Use SupportFragmentManager – `Android.App.FragmentActivity` exposes a `SupportingFragmentManager` property that must be used to get a reference to the `FragmentManager`. For example:

```
FragmentTransaction fragmentTx =
this.SupportingFragmentManager.BeginTransaction();
DetailsFragment detailsFrag = new DetailsFragment();
fragmentTx.Add(Resource.Id.fragment_container, detailsFrag);
fragmentTx.Commit();
```

With these changes in place, it will be possible to run a Fragment-based application on Android 1.6 or 2.x as well as on Honeycomb and Ice Cream Sandwich.

Walkthrough

Now that you understand the theory behind fragments it is time to do a practical walk through of using fragments. This walkthrough will present a list of Shakespearean plays. When the user selects a play, the application will display a passage from the selected play. The application will look different depending on the device it is running on. On a phone, there will be one screen that display a list of Shakespeare's plays:


-14° 14:22

Fragments Walkthrough

Henry IV (1)

Henry V

Henry VIII

Richard II

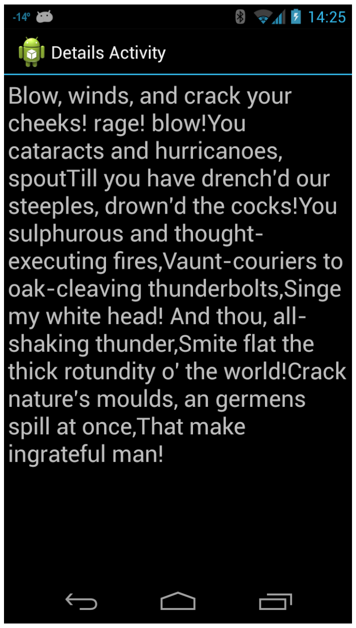
Richard III

Merchant of Venice

Othello

King Lear

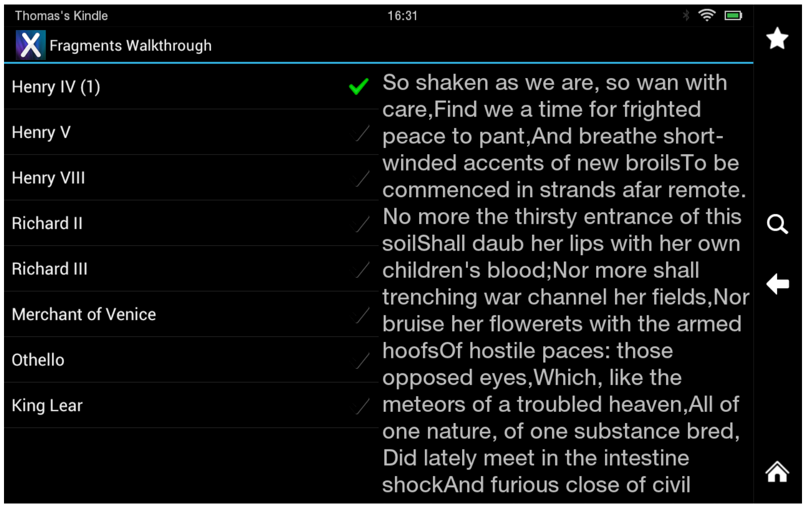
And another list that will display a quote from selected play:


-14° 14:25

Details Activity

Blow, winds, and crack your cheeks! rage! blow!You cataracts and hurricanoes, spoutTill you have drench'd our steeples, drown'd the cocks!You sulphurous and thought-executing fires,Vaunt-couriers to oak-cleaving thunderbolts,Singe my white head! And thou, all-shaking thunder,Smite flat the thick rotundity o' the world!Crack nature's moulds, an germens spill at once,That make ingrateful man!

In the case of a screen we want the application to display a list of plays on the left hand side. When the user selects a play, the quote should be displayed on the right hand side of the screen, as shown in the following screenshot:


Thomas's Kindle 16:31

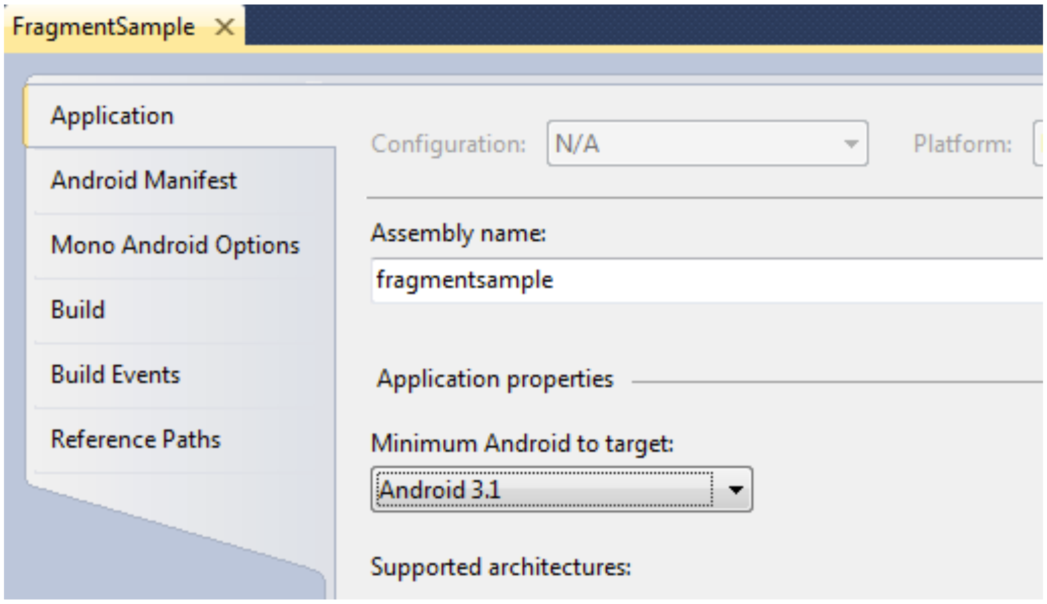
Fragments Walkthrough

Henry IV (1) ✓
Henry V
Henry VIII
Richard II
Richard III
Merchant of Venice
Othello
King Lear

So shaken as we are, so wan with care,Find we a time for frighted peace to pant,And breathe short-winded accents of new broilsTo be commenced in strands afar remote. No more the thirsty entrance of this soilShall daub her lips with her own children's blood;Nor more shall trenching war channel her fields,Nor bruise her flowerets with the armed hoofsOf hostile paces: those opposed eyes,Which, like the meteors of a troubled heaven,All of one nature, of one substance bred, Did lately meet in the intestine shockAnd furious close of civil

With this in mind, lets get started on the walkthrough.

1. Create a new Mono for Android project called `FragmentSample`. The **Minimum Android** version should be set to Android 3.1, as shown in the image below:


FragmentSample

Application
Android Manifest
Mono Android Options
Build
Build Events
Reference Paths

Configuration: N/A Platform:

Assembly name: fragmentsample

Application properties

Minimum Android to target: Android 3.1

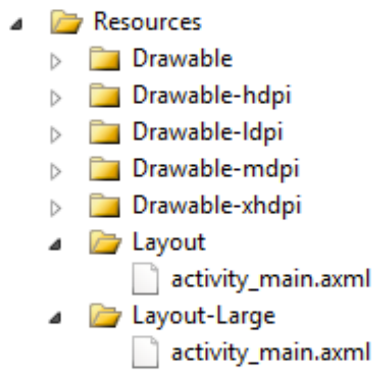
Supported architectures:

2. Next, we need to create the `MainActivity` class. This is the startup Activity for the application. When the Mono for Android project is created, the startup Activity is called Activity1. Rename the file and class to `MainActivity.cs` and `MainActivity`, respectively.

This Activity will host one or both fragments, depending on the screen size. `MainActivity` will do this by loading the layout file that is appropriate for the device.

```
[Activity(Label = "Fragments Walkthrough", MainLauncher = true, Icon =
"@drawable/launcher")]
public class Activity1 : FragmentActivity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SetContentView(Resource.Layout.activity_main);
    }
}
```

3. Now we must create two new layouts, one for portrait and one for landscape orientation. So let's create a new folder, `Resources/Layout-Large`, and create a new layout called `activity_main.axml`. We'll also rename the default layout file as `Resources/Layout/activity_main.axml`. After these changes, the layout folders should resemble the following screen shot:



All devices will load and use the layout file in `Resources/Layout`. It is a very simple layout that just displays a `TitlesFragment`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
  <fragment class="com.xamarin.sample.fragments.TitlesFragment"
            android:id="@+id/titles_fragment"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent" />
</LinearLayout>
```

For devices that have a large screen, Android will load the layout file in `Resources/Layout-Large`. The content of the layout for tablets is as follows:

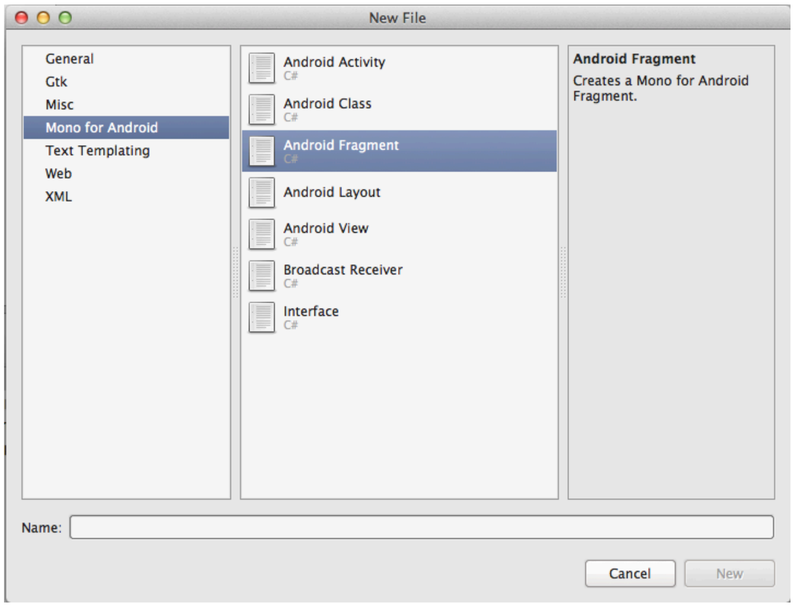
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

  <fragment class="com.xamarin.sample.fragments.TitlesFragment"
            android:id="@+id/titles_fragment"
            android:layout_weight="1"
            android:layout_width="0px"
            android:layout_height="match_parent" />

  <FrameLayout android:id="@+id/details"
               android:layout_weight="1"
               android:layout_width="0px"
               android:layout_height="match_parent" />
```

```
</LinearLayout>
```

The layout file for the larger screens is slightly different. Not only is the `TitlesFragment` displayed in this layout file, but a `FrameLayout` is added right next to the fragment. On the larger screens, the `DetailsFragment` is programmatically added to `MainActivity` when the user selects a play. Later on, we'll explain in more detail how this is done.

> Android 3.2 introduced a new way to specify screen layouts. These new qualifiers specify the amount of space your layout needs, rather than the size of the screen. If this application is intended to run only on Android 3.2 or higher, we would create a `Resource/Layout-sw600dp` folder (instead of the folder `Resource/Layout-Large`) for the layout file `activity_main.axml`. This resource file would be loaded by all devices that have a minimum screen width of 600 density-independent pixels. However, as this application is set to target Android 3.1 or higher, it uses the older resource qualifier.

4. Create the TitlesFragment. `TitlesFragment` will display the titles of the various plays, so let's add a new fragment to the project called `TitlesFragment`:



After `TitlesFragment` has been added, we must change the class so that it inherits from `Android.App.ListFragment.` `ListFragment` is a specialized fragment type that includes list functionality. `TitlesFragment` will also override `OnCreateActivity` (another fragment lifecycle method) and provide an `Adapter` that `ListFragment` will use to populate the list:

```
public override void OnActivityCreated(Bundle savedInstanceState)
{
    base.OnActivityCreated(savedInstanceState);

    var adapter = new ArrayAdapter<String>(Activity,
Android.Resource.Layout.SimpleListItemChecked, Shakespeare.Titles);
```

```
    ListAdapter = adapter;

    if (savedInstanceState != null)
    {
        _currentPlayId = savedInstanceState.GetInt("current_play_id", 0);
    }

    var detailsFrame = Activity.FindViewById<View>(Resource.Id.details);
    _isDualPane = detailsFrame != null && detailsFrame.Visibility ==
ViewStates.Visible;
    if (_isDualPane)
    {
        ListView.ChoiceMode = (int) ChoiceMode.Single;
        ShowDetails(_currentPlayId);
    }
}
```

As previously mentioned, our application has two layouts for `MainActivity`. The code in `OnCreateActivity` detects the presence of the `FrameLayout` and determines which layout file has been loaded. If the `FrameLayout` exists in the layout, then the `_isDualPane` flag is set to `true`. The `_isDualPane` flag is used elsewhere in the Activity, specifically by the `ShowDetails` method. The `ShowDetails` method will be covered in more detail below.

`TitlesFragment` is a list, and must respond to user selections in the list. To do this, `TitlesFragment` will override the method `OnListItemClick`. Inside `OnListItemClick`, a new `DetailsFragment` will be created and displayed in the `FrameLayout`, programmatically. The relevant code inside `TitlesFragment` is:

```
public override void OnListItemClick(ListView l, View v, int position,
long id)
{
    ShowDetails(position);
}

private void ShowDetails(int playId)
{
    _currentPlayId = playId;
    if (_isDualPane)
    {
        // We can display everything in place with fragments.
        // Have the list highlight this item and show the data.
        ListView.SetItemChecked(playId, true);

        // Check what fragment is shown, replace if needed.
        var details =
FragmentManager.FindFragmentById(Resource.Id.details) as DetailsFragment;
        if (details == null || details.ShownPlayId != playId)
        {
            // Make new fragment to show this selection.
            details = DetailsFragment.NewInstance(playId);

            // Execute a transaction, replacing any existing
            // fragment with this one inside the frame.
            var ft = FragmentManager.BeginTransaction();
```

```
                ft.Replace(Resource.Id.details, details);
                ft.SetTransition(FragmentTransaction.TransitFragmentFade);
                ft.Commit();
            }
        }
        else
        {
            // Otherwise we need to launch a new Activity to display
            // the dialog fragment with selected text.
            var intent = new Intent();

            intent.SetClass(Activity, typeof (DetailsActivity));
            intent.PutExtra("current_play_id", playId);
            StartActivity(intent);
        }
    }
```

The code determines from the device how to format and display the quote from the selected play. In the case of tablets, the `_isDualPane` flag will be set to `true`, and so the quote will be displayed next to the `TitlesFragment`. If the selected play `id` is not already displayed, then a new `DetailsFragment` is created, and then loaded into the `FrameLayout` on the Activity. For other devices that do not have a large display—phones, for example—`_isDualPane` will be set to `false` so a new `DetailsActivity` will be started

5. The `DetailsActivity` displays the `DetailsFragment` for smaller devices. To see this, first we'll add a new Activity to the project named `DetailsActivity`. `DetailsActivity` is a very simple Activity. It will create and then host a new `DetailsFragment` for the play `id` that was sent:

```
[Activity(Label = "Details Activity")]
public class DetailsActivity : FragmentActivity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        var index = Intent.Extras.GetInt("current_play_id", 0);

        var details = DetailsFragment.NewInstance(index); //
DetailsFragment.NewInstance is a factory method to create a Details
Fragment
        var fragmentTransaction = FragmentManager.BeginTransaction();
        fragmentTransaction.Add(Android.Resource.Id.Content, details);
        fragmentTransaction.Commit();
    }
}
```

Notice that no layout file is loaded for `DetailsActivity`. Instead, `DetailsFragment` is loaded into the root view of the Activity. This root view has the special ID `Android.Resource.Id.Content`. A new `DetailFragment` is created and then added to this root view inside of a `FragmentTransaction` that is created by the Activity's `FragmentManager`.

5. Now, let's add another fragment to the application named `DetailsFragment`. This fragment will display a quote from the selected play that is displayed in `TitlesFragment`. The following code shows the complete `DetailsFragment`:

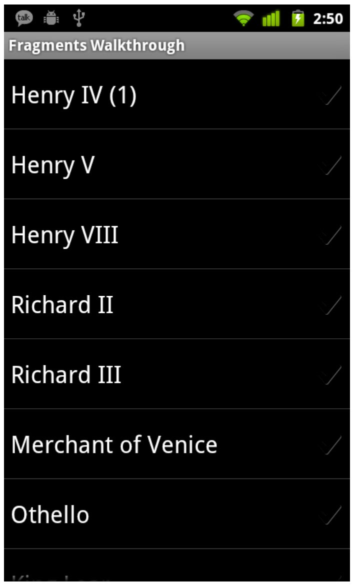
```
internal class DetailsFragment : Fragment
{
    public static DetailsFragment NewInstance(int playId)
    {
        var detailsFrag = new DetailsFragment {Arguments = new Bundle()};
        detailsFrag.Arguments.PutInt("current_play_id", playId);
        return detailsFrag;
    }

    public int ShownPlayId
    {
        get { return Arguments.GetInt("current_play_id", 0); }
    }

    public override View OnCreateView(LayoutInflater inflater, ViewGroup
container, Bundle savedInstanceState)
    {
        if (container == null)
        {
            // Currently in a layout without a container, so no reason to
create our view.
            return null;
        }

        var scroller = new ScrollView(Activity);

        var text = new TextView(Activity);
        var padding =
Convert.ToInt32(TypedValue.ApplyDimension(ComplexUnitType.Dip, 4,
Activity.Resources.DisplayMetrics));
        text.SetPadding(padding, padding, padding, padding);
        text.TextSize = 24;
        text.Text = Shakespeare.Dialogue[ShownPlayId];

        scroller.AddView(text);

        return scroller;
    }
}
```

In order for `DetailsFragment` to function properly, it must have the index of the play that is selected in the `TitlesFragment`. There are many ways to provide this value to `DetailsFragment`; in this example, the play `Id` is placed into a Bundle and that Bundle is stored to the Arguments property of an instance of the `DetailsFragment`. The property `ShownPlayId` is provided for convenience—it will be used by instances of `DetailsFragment` to retrieve that value from the Bundle.

`OnCreateView` is called when the fragment needs to draw its user interface and should return an `Android.Views.View` object. In most cases, this is a `View` inflated from an existing layout file. In the case of the above example, the fragment will programmatically build the view that will be used for display.

Congratulations! You've now created an application that uses fragments to simplify development across form factors. The Activity behaves like a switchboard, coordinating communication between the two Fragments and updating the UI in response to use selections.

In this next section, we're going to extend this application so that it will work on pre-Android 3.0 devices. We will add the Android Support Packages to the application, and then make some changes to the existing codebase. Once this is done, our application will run on older versions of Android. The following screenshots show the application running on Android 2.3:



Fragments Walkthrough

Henry IV (1)

Henry V

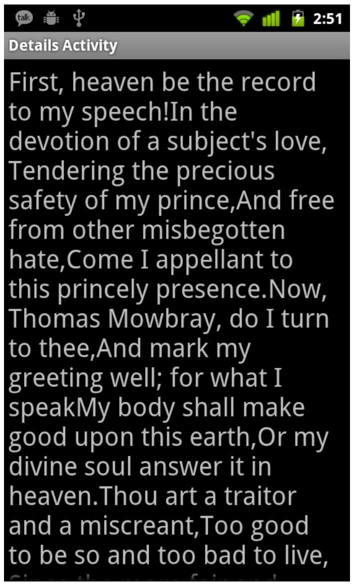
Henry VIII

Richard II

Richard III

Merchant of Venice

Othello



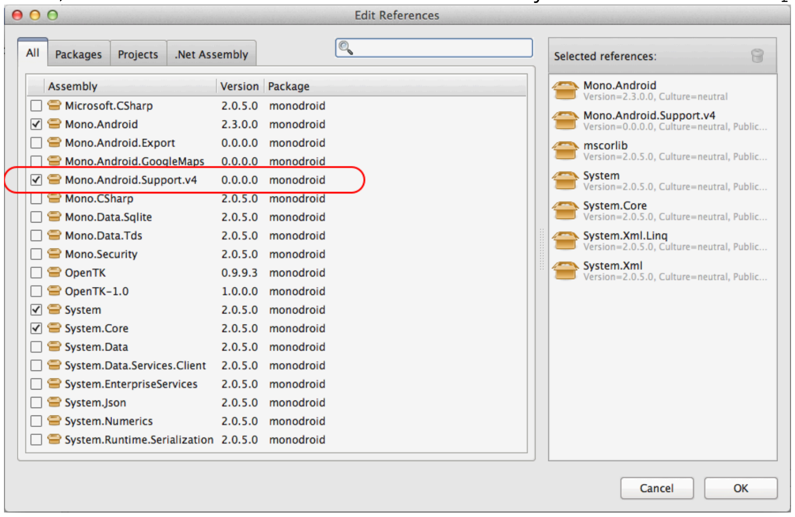
Details Activity

First, heaven be the record to my speech!In the devotion of a subject's love, Tendering the precious safety of my prince,And free from other misbegotten hate,Come I appellant to this princely presence.Now, Thomas Mowbray, do I turn to thee,And mark my greeting well; for what I speakMy body shall make good upon this earth,Or my divine soul answer it in heaven.Thou art a traitor and a miscreant,Too good to be so and too bad to live,

Now, lets get started making changes to this application.

1. The first thing we need to do is to add the *Android Support Package* to our project. This is a three-step process:

   ➔ First, add a reference to the assembly `Mono.Android.Support.v4`:



	Assembly	Version	Package
☐	Microsoft.CSharp	2.0.5.0	monodroid
✓	Mono.Android	2.3.0.0	monodroid
☐	Mono.Android.Export	0.0.0.0	monodroid
☐	Mono.Android.GoogleMaps	0.0.0.0	monodroid
✓	Mono.Android.Support.v4	0.0.0.0	monodroid
☐	Mono.CSharp	2.0.5.0	monodroid
☐	Mono.Data.Sqlite	2.0.5.0	monodroid
☐	Mono.Data.Tds	2.0.5.0	monodroid
☐	Mono.Security	2.0.5.0	monodroid
☐	OpenTK	0.9.9.3	monodroid
☐	OpenTK-1.0	1.0.0.0	monodroid
✓	System	2.0.5.0	monodroid
✓	System.Core	2.0.5.0	monodroid
☐	System.Data	2.0.5.0	monodroid
☐	System.Data.Services.Client	2.0.5.0	monodroid
☐	System.EnterpriseServices	2.0.5.0	monodroid
☐	System.Json	2.0.5.0	monodroid
☐	System.Numerics	2.0.5.0	monodroid
☐	System.Runtime.Serialization	2.0.5.0	monodroid

Selected references:

Mono.Android
Version=2.3.0.0, Culture=neutral

Mono.Android.Support.v4
Version=0.0.0.0, Culture=neutral, Public...

mscorlib
Version=2.0.5.0, Culture=neutral, Public...

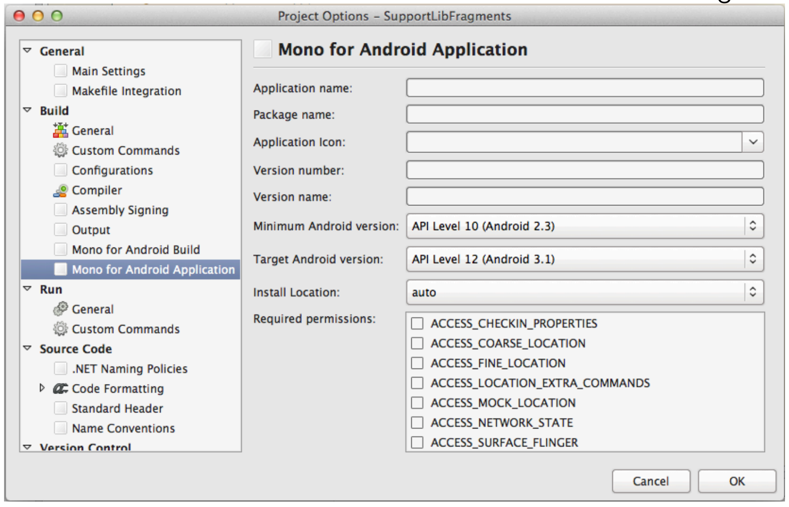
System
Version=2.0.5.0, Culture=neutral, Public...

System.Core
Version=2.0.5.0, Culture=neutral, Public...

System.Xml.Linq
Version=2.0.5.0, Culture=neutral, Public...

System.Xml
Version=2.0.5.0, Culture=neutral, Public...

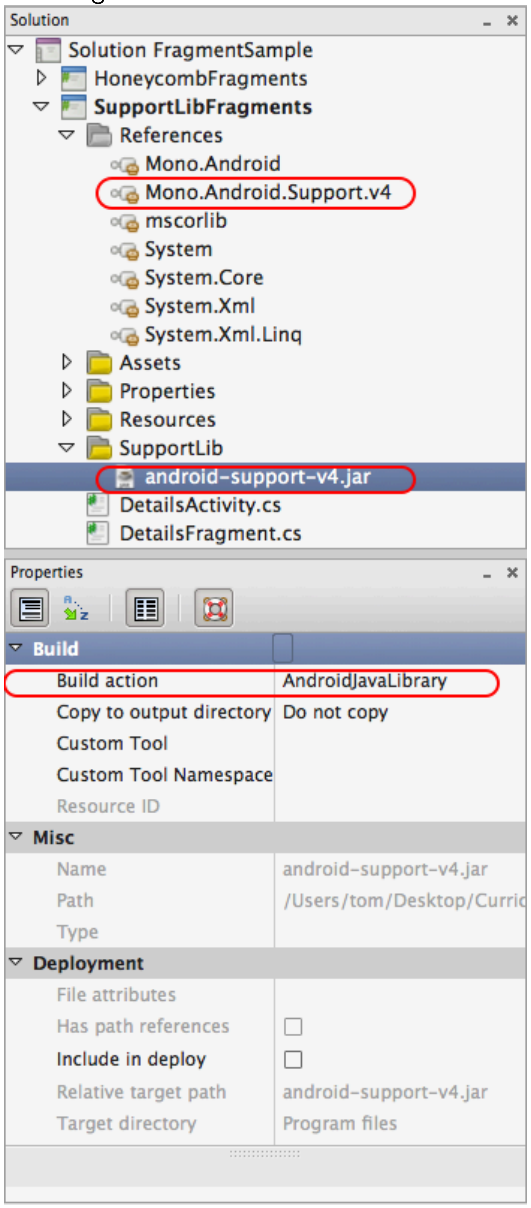
➔ Next we need set the minimum version of Android. In this example set the Minimum Android version to Android 2.3, but keep the Target Android version to 3.1 as shown in the following screenshot:



Mono for Android Application

Application name:
Package name:
Application Icon:
Version number:
Version name:
Minimum Android version: API Level 10 (Android 2.3)
Target Android version: API Level 12 (Android 3.1)
Install Location: auto
Required permissions:
- [ ] ACCESS_CHECKIN_PROPERTIES
- [ ] ACCESS_COARSE_LOCATION
- [ ] ACCESS_FINE_LOCATION
- [ ] ACCESS_LOCATION_EXTRA_COMMANDS
- [ ] ACCESS_MOCK_LOCATION
- [ ] ACCESS_NETWORK_STATE
- [ ] ACCESS_SURFACE_FLINGER

➔ If you are running Mono for Android 4.4 or higher, you can safely skip this step. Copy the file `android-support-v4.jar` from the Extras folder of the Android SDK folder `<sdk>/extras/android/support` to a new folder in the project called `SupportLib`. The BuildAction of the file will be automatically set to AndroidJavaLibrary by Mono for Android, as soon as the JAR file is added to the project. The image below shows what the structure of the Mono for Android project should look like after following these steps:



Build action	AndroidJavaLibrary
Copy to output directory	Do not copy
Custom Tool	
Custom Tool Namespace	
Resource ID	
Name	android-support-v4.jar
Path	/Users/tom/Desktop/Curric
Type	
File attributes	
Has path references	☐
Include in deploy	☐
Relative target path	android-support-v4.jar
Target directory	Program files

2. Any Activity that uses fragments must inherit from `Android.Support.V4.App.FragmentActivity`. This class is a necessary part of the Support Package because it allows fragments to be hosted by activities, regardless of the version of Android. Let's make this change to `MainActivity`:

```
[Activity(Label = "Fragments Walkthrough", MainLauncher = true, Icon =
"@drawable/launcher")]
public class MainActivity : FragmentActivity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SetContentView(Resource.Layout.activity_main);
    }
}
```

3. DetailsActivity must also be changed from an Activity to a FragmentActivity. As FragmentManager is not compatible with pre-Honeycomb versions of Android, the Android Support Package includes a wrapper class, *SupportFragmentManager*, that provides backward compatibility. Each FragmentActivity has a SupportFragmentManager property, and DetailsActivity is changed to use the SupportFragmentManager instead of the FragmentManager:

```
[Activity(Label = "Details Activity")]
public class DetailsActivity : FragmentActivity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        var index = Intent.Extras.GetInt("index", 0);

        var details = DetailsFragment.NewInstance(index);
        var fragmentTransaction =
SupportFragmentManager.BeginTransaction(); // Notice the change from
FragmentManager to SupportFragmentManager
        fragmentTransaction.Add(Android.Resource.Id.Content, details);
        fragmentTransaction.Commit();
    }
}
```

At this point sit back and give yourself a pat on the back. You just created your first application that will run on all current versions of Android, and will take advantage of the larger screens that tablets provide.

Summary

Fragments provide a way to decompose an Android application into re-usable components that can be hosted in Activities. This chapter introduced Fragments, and then went on to explain the Fragment lifecycle. Then you were shown how to add a Fragment to an application and use it in Activities. The specialized ListFragment was introduced and discussed. To help with backwards compatibility you were then introduced to the Android Support Package. Finally we finished up with a walkthrough, creating an application that used a ListFragment and a Fragment that would dynamically adapt its display to the device it's running on.