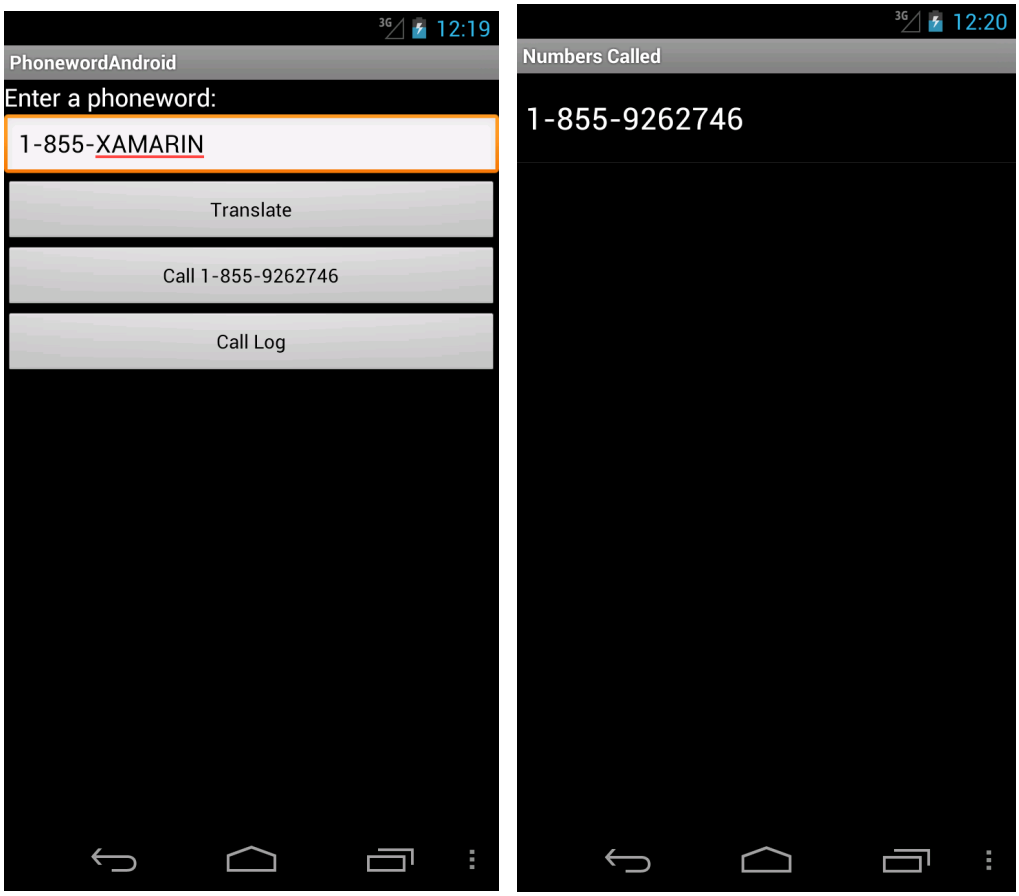
Hello, Android Multiscreen Applications

Evolve Fundamentals Track, Chapter 5

Xamarin Inc.

Overview

In the last chapter, we created a simple PhoneWord application that translated a word into a phone number and then dialed the phone number for you. This chapter will expand on this application and display a second Activity that will show a list of phone numbers that were called by the current session of our application. The following screenshot shows an example of what this new Activity will look like:



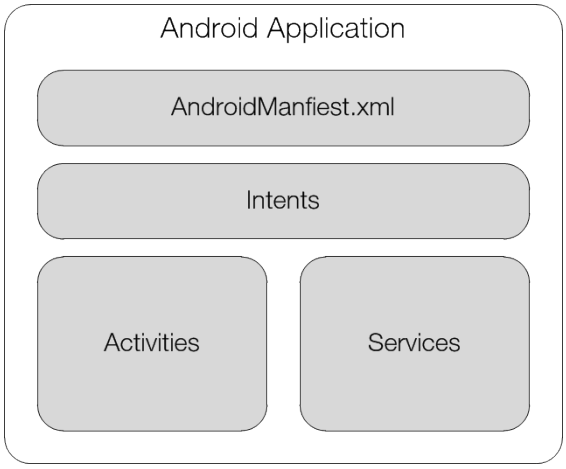
To understand how to do this, we will first discuss the anatomy of an Android application in a bit more detail. We will introduce *Intents* and how they are used to start additional Activities. We will also briefly touch on how to use the `ListActivity` class to display data in lists.

Anatomy of an Android Application

Android applications are very different from traditional client applications found on platforms such as Windows and Mac OSX, or even those that run on mobile platforms such as iOS. These platforms have a single-entry point into the application that creates an instance of an application that then launches, loads, and manages its screens. By contrast, Android applications consist of a set of loosely coupled screens, represented by `Activity` classes and `Service` classes, which are long running background processes. This is somewhat similar to web

applications (in that they can be started from various URLs), except that typically even web applications retain an application instance.

The following diagram illustrates the components of a basic Android application:



This loosely coupled architecture presents an interesting problem for multiscreen applications. Since each Activity is essentially decoupled from all others, there needs to be a way to launch additional Activities and optionally pass data to them. On Android, this is accomplished by using *Intents*. Intents are classes that describe a message: both what the desired action of the message is and a data payload to send along with it.

Let's explore Activities and Intents a little more.

Activities

As mentioned, Activities are classes that provide user interfaces. An Activity is given a window in which to add a user interface. This means that creating multiscreen applications involves creating multiple Activities and transitioning between them.

The `Activity` class inherits from the abstract `Context` class.

Context

`Context` is the closest Android gets to a reference to the current application that provides a mechanism for accessing the Android system. A `Context` is needed to perform many operations in Android such as:

- ➔ Accessing Android services
- ➔ Accessing preferences
- ➔ Creating views
- ➔ Accessing device resources

An Android application needs a `Context` to know which permissions the application has, how to create controls, how to access preferences, etc. However, because (as we described above) Android apps are a set of loosely coupled Activities and Services, no single static application `Context` exists. Therefore, each

Activity and Service inherits from the `Context` class, which contains all the information the application needs. Every time a new Activity is created, it's passed a `Context` from the Activity that created it.

Since an Activity derives from `Context`, any call that takes a `Context` as an argument can take an Activity.

Activity Lifecycle Overview

Every Activity has a lifecycle associated with it, ranging from creation to destruction. Activities can be paused or destroyed by the system at any time. The lifecycle provides a way for Activities to handle the various lifecycle methods that Android will call and gives them an opportunity to save or rehydrate state, so screens can continue to operate normally. After completing this Getting Started series, we highly recommend checking out the Activity Lifecycle document for a detailed discussion on this subject.

In short, the Android Activity lifecycle comprises a collection of methods exposed within the Activity class that provide the developer with a resource management framework. This framework allows implementers to meet the unique state management requirements of each Activity within an application. The Activity lifecycle thus assists the developer by providing a consistent framework in which to handle resource management within the application.

Intents

Intents are used throughout Android to make things happen by sending messages. Intents are most commonly used to launch Activities within applications. To launch a new Activity, we create a new Intent, set the `Context` and the `Activity` classes to launch, and then tell the OS to handle the Intent, which launches the Activity.

Additionally, Intents can be used to tell the OS to launch Activities from other applications as well. For example, an application can have the intention to dial a phone number when the user taps a button. The way an application announces this intention is via an Intent for a phone-dialing action. However, the actual dialing of the number is handled by an Activity in the Phone Dialer Application.

AndroidManifest.xml

Every Android application needs to include a file called *AndroidManifest.xml.* This file contains information about the application such as:

- ➔ Component Registration – The components that make up the app, including registration of Activities and Intents.
- ➔ Required Permissions – The permissions the app requires.
- ➔ OS Version Compatibility – The Android API level the application is meant to run in, and optionally the minimum API level that the application can be run on.

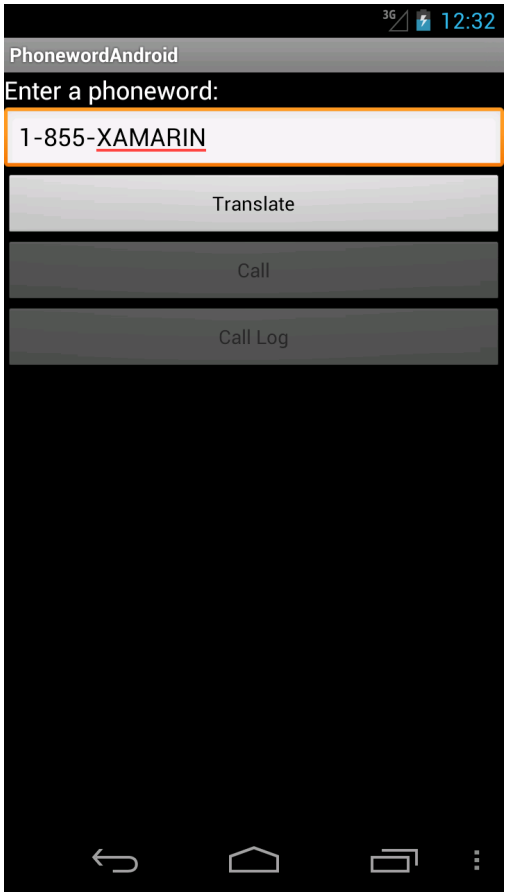
We'll examine the manifest later on in this document, during the application walkthrough. For more details, see the AndroidManifest.xml documentation on the Android Developers site.

Application Walkthrough

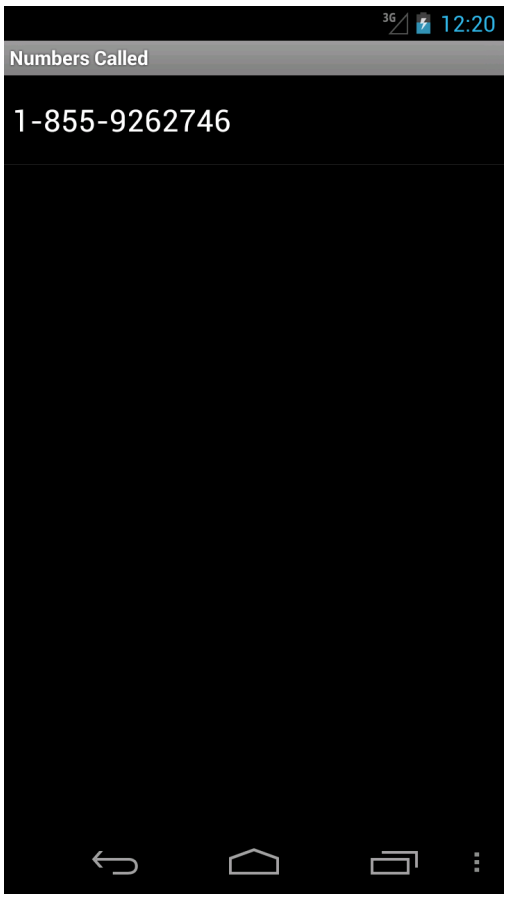
Now that we've explained some of the more unusual concepts that we use to build Android applications, let's create a simple, multiscreen application.

We are going to make a new application where the first Activity will include a button. Tapping the button will load another Activity. For this example, we'll demonstrate how to load a second Activity explicitly from a class defined within the application.

Here is a screenshot showing the application after it is launched:



Notice the addition of the Call Log button. Once a call has been placed by this application, this button will be enabled. When the user clicks on it, a new Activity will be started that shows all the calls that have been placed. An example of the Call Log Activity can be seen in this next screenshot:



Setting Up

We want to update the PhoneWord application from the previous example, and make changes to it to support this new functionality. Unzip the contents of `PhoneWord_MultiScreens_Start.zip` and open the solution in Xamarin.Studio. To save you some typing throughout this tutorial, we have added some string resources to the file `Resources/values/Strings.xml`. If you open this file up, you will see that the contents of this file are:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
	<string name="Enter">Enter a phoneword:</string>
	<string name="Number">1-855-XAMARIN</string>
	<string name="Translate">Translate</string>
	<string name="Call">Call</string>
	<string name="app_name">Phoneword</string>
	<string name="CallLogButtonLabel">Call Log</string>
	<string name="CallLogActivityLabel">Numbers Called</string>
</resources>
```

Using ActivityAttribute

Notice that the `MainActivity` class is decorated with an *ActivityAttribute* as shown below:

```
[Activity (Label = "@string/app_name", MainLauncher = true)]
public class MainActivity : Activity
```

We mentioned earlier that Activities have to be registered in the `AndroidManifest.xml` file in order for them to be located by the OS, and the `ActivityAttribute` aids in doing just that. During compilation, the Xamarin.Android build process gathers all the attribute data from all the class files in the project and then creates a new `AndroidManifest.xml` file. This new file is based on the `AndroidManifest.xml` file in the project, but the attribute data is merged into it. The compiler then bundles that manifest into the resulting compiled application (APK file).

By using C# attributes, Xamarin.Android is able to keep the declarative information that is used to register an Activity together with the class definition. Failure to include this attribute will result in Android not being able to locate the Activity, since your Activity would not be registered with the system.

Additionally, the attribute sets `MainLauncher` to `true`. This Activity is then registered as the Activity to launch by the compiler when the application starts. During installation of an app, Android will populate the application launcher with the icon of the launchable Activity.

Manifest File

When the `ActivityAttribute` is merged into the manifest, it will result in XML similar to the following:

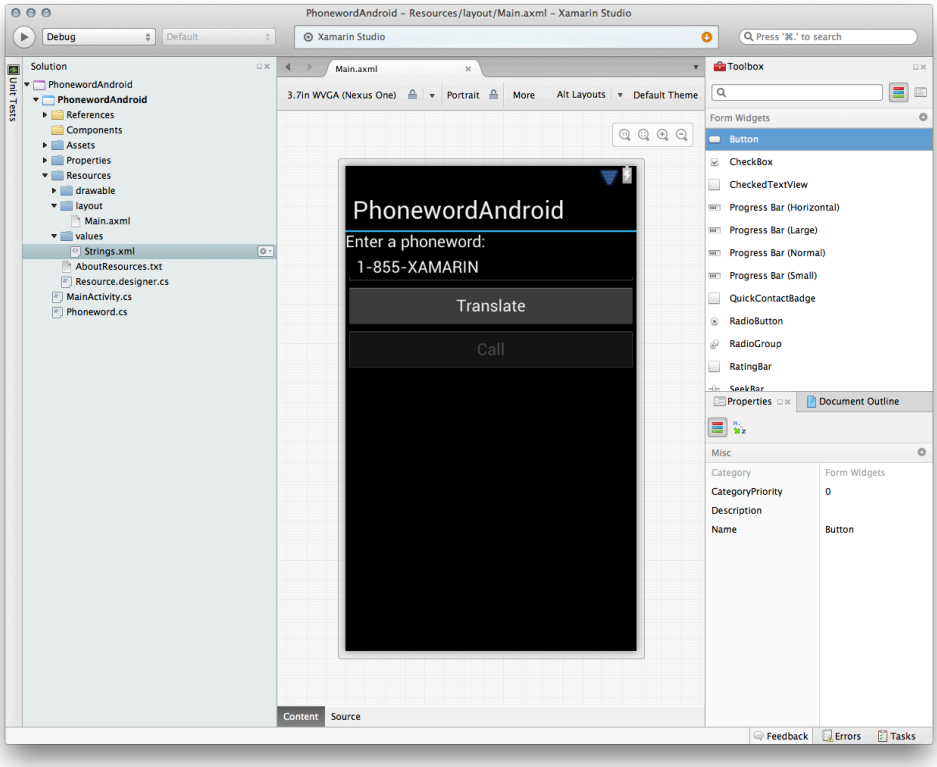
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
android:versionCode="1" android:versionName="1.0"
package="PhonewordAndroid.PhonewordAndroid">
  <uses-sdk android:minSdkVersion="8" />
  <application android:label="PhonewordAndroid"
android:name="mono.android.app.Application" android:debuggable="true">
    <activity android:label="@string/app_name"
android:name="phonewordandroid.MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <provider android:name="mono.MonoRuntimeProvider"
android:exported="false" android:initOrder="2147483647"
android:authorities="PhonewordAndroid.PhonewordAndroid.mono.MonoRuntimePro
vider.__mono_init__" />
    <receiver android:name="mono.android.Seppuku">
      <intent-filter>
        <action android:name="mono.android.intent.action.SEPPUKU" />
        <category
android:name="mono.android.intent.category.SEPPUKU.PhonewordAndroid.Phonew
ordAndroid" />
      </intent-filter>
    </receiver>
  </application>
  <uses-permission android:name="android.permission.CALL_PHONE" />
  <uses-permission android:name="android.permission.INTERNET" />
```

```
</manifest>
```

In the XML above, `MainActivity` uses an Intent filter to register with the system as being enabled for launch, with the Intent filter's action set to `android.intent.action.MAIN` and its category set to `android.intent.category.LAUNCHER`. Intent filters register what actions a particular Activity can support, such as being able to handle application launch and serving as the entry point (the intention of the Activity in this case).

Update the UI for MainActivity

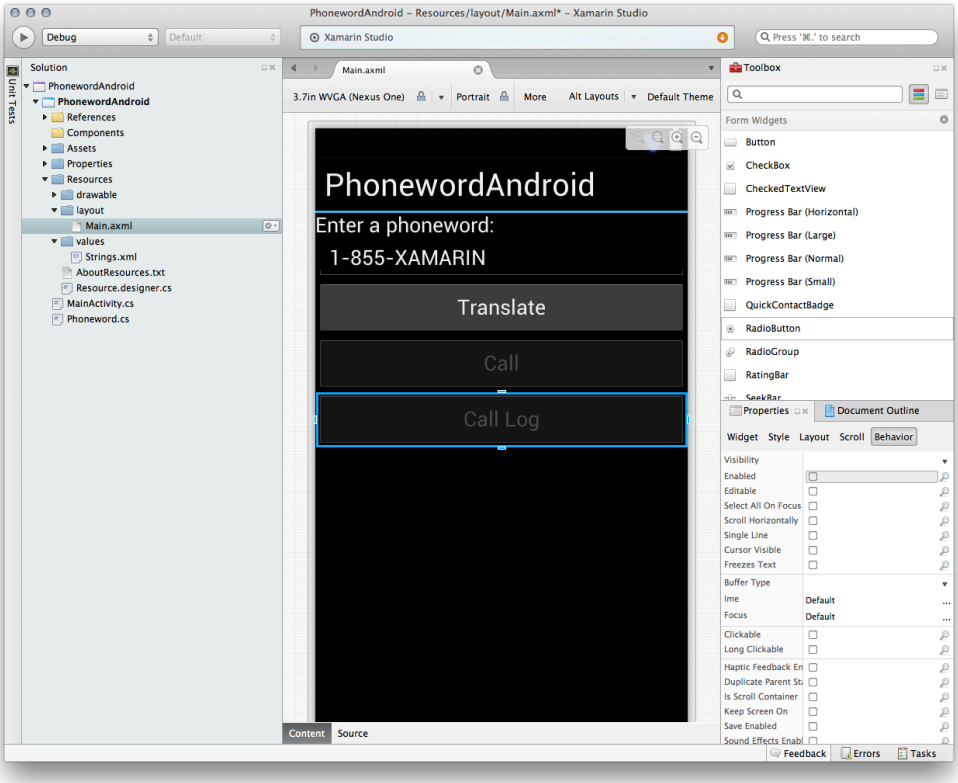
Recall from the previous tutorial that the PhoneWord UI was declaratively created by using XML that was saved in files with an `axml` file name extension. We want to modify this file to add a button to enter the call log screen, so double-click on the file `Resources/layout/Main.axml` to load it in the Xamarin.Android Designer:



Drag a **Button** widget from the Toolbox and drop it under the **Call** button. Next, set some properties on the widget according to the following table:

Property	Value
Id	`@+id/CallLogButton`
Text	`@string/CallLogButtonLabel`
Enabled	`False`

When you are done, `Main.axml` should look something like the following screenshot:

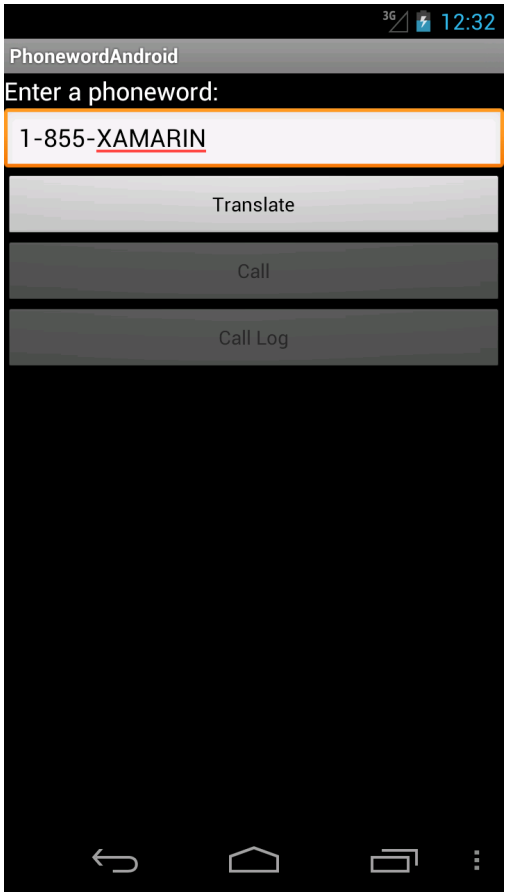


The complete XML for the layout file is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:text="@string/Enter"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/textView1" />
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/PhoneNumberText"
        android:text="@string/Number" />
    <Button
        android:text="@string/Translate"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/TranslateButton" />
    <Button
```

```
            android:text="@string/Call"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:id="@+id/Call"
            android:enabled="false" />
        <Button
            android:text="@string/CallLogButtonLabel"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:id="@+id/CallLogButton"
            android:enabled="false" />
    </LinearLayout>
```

If we run the application at this point, it should look like the following screenshot:



Track the Dialed Phone Numbers

Now that the user interface has been updated with a new **Button**, we need to change the code so that it does two things:

- ➔ Keeps track of the phone numbers that were dialed.
- ➔ Displays the Call Log Activity when the user clicks `CallLogButton`.

To keep things simple, our application will use a `List<String>` to store the phone numbers that the user chooses to dial. Let's change the code in `MainActivity` to

do that. First, open up the file `MainActivity.cs`, and add the following instance variable to the class `MainActivity`:

```
private List<String> _phoneNumbers = new List<String>();
```

The next thing we need our application to do is to record each phone number that is dialed. To do this, look in the `OnCreate` method of `MainActivity`. We need to obtain a reference to the new button we just added to the layout, so after the calling `SetContentView(Resource.Layout.Main)` add this line of code:

```
var callLogButton = FindViewById<Button>(Resource.Id.CallLogButton);
```

Next, locate the code that initializes the **Neutral** button on the **alert** dialog; it should look something like this:

```
dialogBuilder.SetNeutralButton("Call", delegate
{
        StartActivity(callIntent);
});
```

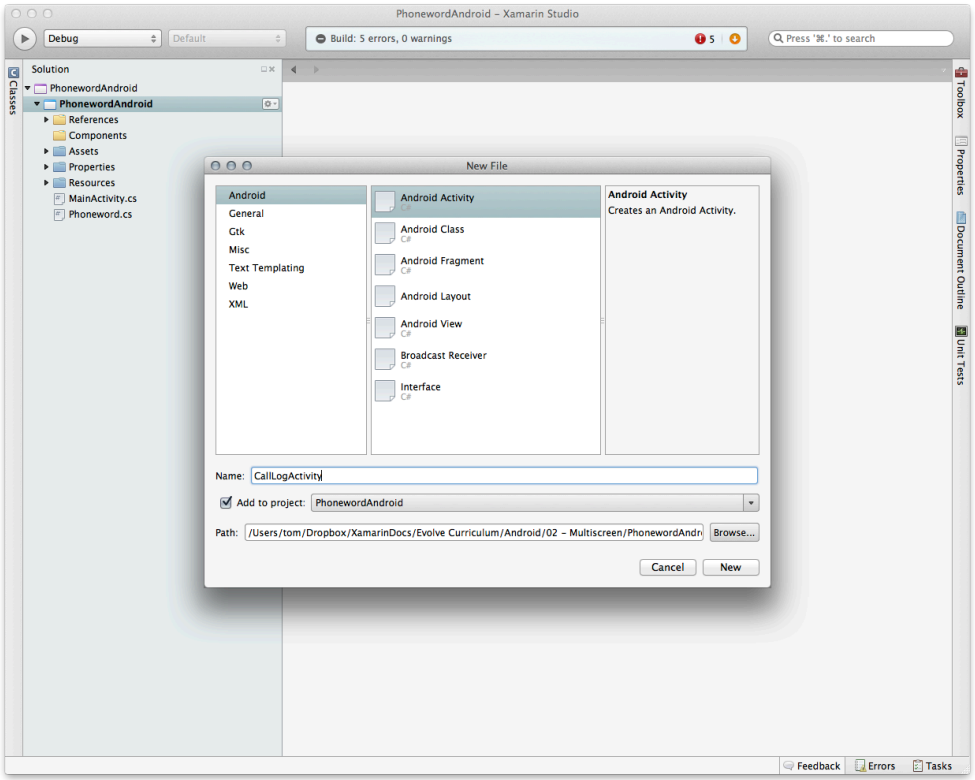
Change the delegate so that it looks like the following code:

```
dialogBuilder.SetNeutralButton("Call", delegate
{
        _phoneNumbers.Add(_translatedNumber);
        callLogButton.Enabled = true;
        StartActivity(callIntent);
});
```

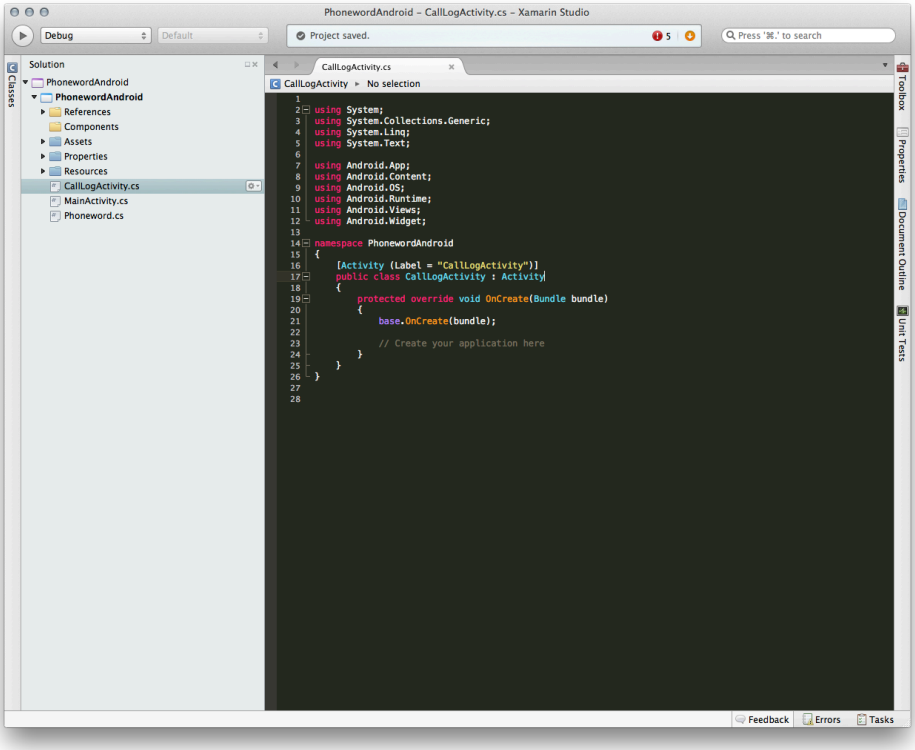
In this snippet, all we do is add the current `_translatedNumber` to the instance variable `_phoneNumbers`, and then enable the **Call Log** button. After completing this change, let's look at how we can start up a second Activity to display the contents of `_phoneNumbers`. Let us leave `MainActivity` alone for now, and create the second Activity.

Creating The Second Activity

We need to add another Activity to our application, one to which the `MainActivity` can pass a list of phone numbers. Let's do that now. In Xamarin.Studio, select **File > New > File...** from the menu. This will call up the **New File** dialog, as shown in the following screenshot:



Select **Android Activity**, give it the name `CallLogActivity`, and then click **New**. This will add a new Activity to the project. At this point, Xamarin.Studio should look something like the following screenshot:



Edit the contents of this file so that the `CallLogActivity` class resembles the following code snippet:

```
[Activity (Label = "@string/CallLogActivityLabel")]
public class CallLogActivity : ListActivity
{
	protected override void OnCreate(Bundle bundle)
	{
		base.OnCreate(bundle);
		var phoneNumbers =
Intent.Extras.GetStringArrayList("phone_numbers") ?? new
string[0];
		this.ListAdapter = new ArrayAdapter<string>(this,
Android.Resource.Layout.SimpleListItem1, phoneNumbers);
	}
}
```

That's it! This is all the code that we need to display the phone numbers. Let's examine this code in detail.

Just like the first Activity, the class `CallLogActivity` is adorned with the `ActivityAttribute` and the label that displays the value of the string resource `@string/CallLogActivityLabel`.

Next, notice that `CallLogActivity` subclasses a `ListActivity`. `ListActivity` is a specialized subclass of Activity that contains logic for display data in lists. A `ListActivity` contains a single list that occupies the entire screen and encapsulates the logic for binding data to that list. We will go into more detail

about Lists in the next tutorial. Before we discuss how the data is bound to the list, let's look at how we will send data from one Activity to another.

Sending Data Between Activities

`CallLogActivity` will be loaded by Android when an Intent is published by some other Activity. In this example, the data to display is stored in a key-value dictionary of the Intent that was used to start this Activity.

We need to modify `MainActivity` so that when the `CallLogButton` is clicked it will create an Intent that will start up `CallLogActivity`. Add the following code to the `OnCreate` method of `MainActivity`:

```
callLogButton.Click += (object sender, EventArgs e) => {
        var intent = new Intent(this, typeof(CallLogActivity));
        intent.PutStringArrayListExtra("phone_numbers", _phoneNumbers);
        StartActivity(intent);
};
```

In this code for the event handler for the `Click` event of `CallLogButton`, we create an Intent and specify the type of Activity that we want to start. There are many methods available that can store data in an Intent—in this case, we can use the `PutStringArrayListExtra` method, because we have a list of strings.

Let's look at the code in our `CallLogActivity` class. We retrieve the phone number list from the Intent with the following line of code:

```
var phoneNumbers = Intent.Extras.GetStringArrayList(
        PhonewordTranslator.PhoneNumbersKey) ?? new string[0];
```

We used the *null coalescing operator* ("??") there because the Intent data may be `null` when we access it. After we have a list of phone numbers, we need to display them in the Activity.

Display The Data

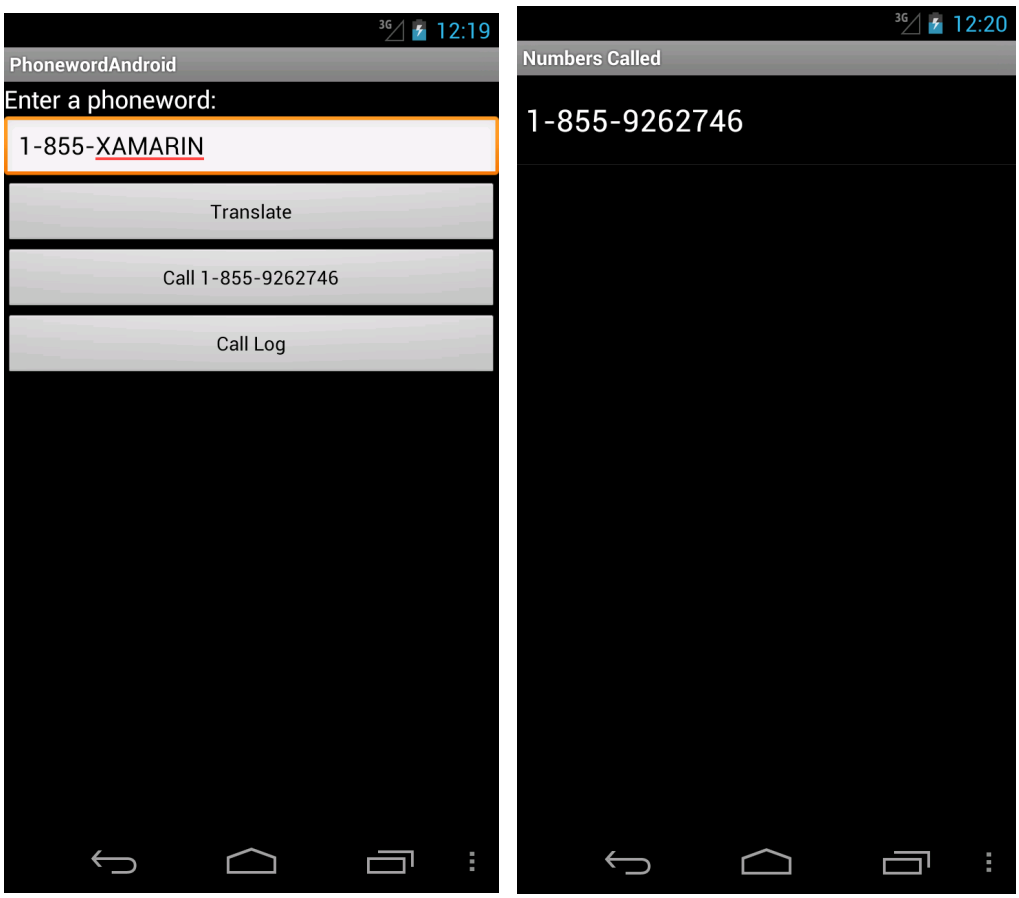
The final thing we need to do is display the phone numbers in CallLogActivity. Android includes the class `ArrayAdapter<T>`, which we can use to bind simple data to a ListView. We can bind the phone numbers with this one line of code:

```
this.ListAdapter = new ArrayAdapter<string>(this,
Android.Resource.Layout.SimpleListItem1, phoneNumbers);
```

`ArrayAdapter<T>` has a constructor that takes the following parameters:

- **`Context`** – This is the context in which this Adapter will exist. In this case, it is the current instance of `CallLogActivity`.
- **`Android.Resource.Layout.SimpleListItem1`** – This is a built-in layout provided by Android, consisting of a single `TextView`.
- **`IList<string>`** - In this case, we just provide the phone numbers that were provided by the Intent.

Congratulations! At this point, you should have a working app that passes data between two Activities. If you run the application, everything should be working as shown in the following screenshots:



Before we jump into the next chapter, we're going to do a quick detour to examine the Activity Lifecycle.

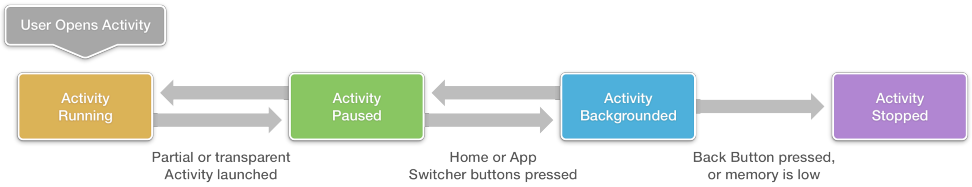
Activity Lifecycle

During app/device usage, Activities transition between a number of states as users launch apps, navigate around, etc. This changes state of the Activities that are running causing them to go through what's known as the Activity Lifecycle.

The Android Activity lifecycle comprises a collection of methods exposed within the Activity class that provide the developer with a resource management framework. This framework allows developers to meet the unique state management requirements of each Activity within an application and properly handle resource management.

Activity States

The Android OS uses a priority queue to assist in managing activities running on the device. Based on the state a particular Android Activity is in, it will be assigned a certain priority within the OS. This priority system helps Android identify activities that are no longer in use, allowing the OS to reclaim memory and resources. The following diagram illustrates the states an Activity can go through, during its lifetime:



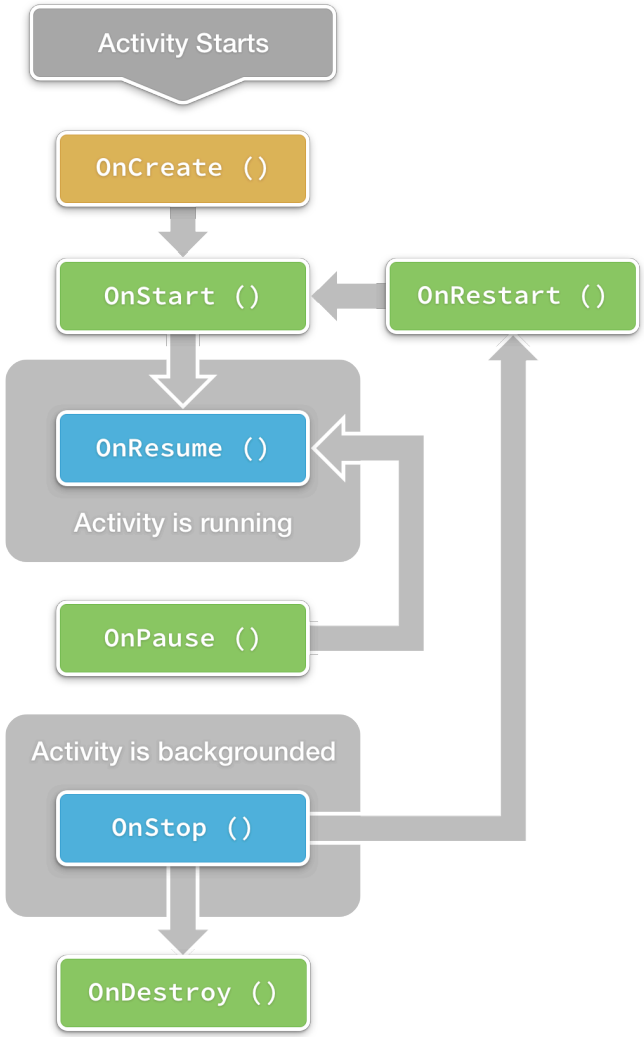
These states can be broken into 4 main groups as follows:

- ➔ **Active or Running** - Activities are considered active or running if they are in the foreground, also known as the top of the Activity stack. This is considered the highest priority Activity in Android, and as such will only be killed by the OS in extreme situations, such as if the Activity tries to use more memory than is available on the device as this could cause the UI to become unresponsive.
- ➔ **Paused** - When the device goes to sleep, or an Activity is still visible but partially hidden by a new, non-full-sized or transparent Activity, the Activity is considered paused. Paused activities are still alive: that is, they maintain all state and member information and remain attached to the window manager. This is considered to be the second highest priority Activity in Android and, as such, will only be killed by the OS if killing this Activity will satisfy the resource requirements needed to keep the Active/Running Activity stable and responsive.
- ➔ **Stopped/Backgrounded** - Activities that are completely obscured by another Activity are considered stopped or in the background. Stopped activities still try to retain their state and member information for as long as possible, but stopped activities are considered to be the lowest priority of the three states and, as such, the OS will kill activities in this state first to satisfy the resource requirements of higher priority activities.
- ➔ **Restarted** – It is possible that an Activity that went from paused to stopped was dropped from memory by Android. If the user navigates back to the Activity it must be restarted, restored to its previously saved state, and then displayed to the user.

Activity Lifecycle Methods

The Android SDK and, by extension, the Xamarin.Android framework provide a powerful model for managing the state of activities within an application. When an Activity's state is changing, the Activity is notified by the OS, which calls specific methods on that Activity.

The following diagram illustrates these methods in relationship to the Activity Lifecycle:



As a developer, you can handle state changes by overriding these methods within an Activity. It's important to note, however, that *all* lifecycle methods are called on the UI thread and will block the OS from performing the next piece of UI work, such as hiding the current Activity, displaying a new Activity, etc. As such, code in them should be as brief as possible to make an application feel performant. Any long-running tasks should be executed on a background thread.

Let's do a quick examination the most important methods here and their use. For a more thorough examination, please see the Activity Lifecycle guide on the Xamarin Developer Center. We will also examine them in more detail in the Advanced Track during the Backgrounding and Advanced Android App Lifecycle chapters.

The following lifecycle methods are the most important, and the only ones we need concern ourselves with right now, as they handle 90% of all use cases:

OnCreate

This is the first method to be called when an Activity is created. `OnCreate` is always overridden to perform any startup initializations that may be required by an Activity such as:

- ➔ Creating views
- ➔ Initializing variables
- ➔ Binding static data to lists

`OnCreate` takes a `Bundle` parameter, which is a dictionary for storing and passing state information and objects between activities or the state. If the bundle is not `null`, this indicates the Activity is restarting and it should restore its state from the previous instance. The following code illustrates how to retrieve values from the bundle:

```
protected override void OnCreate(Bundle bundle)
{
   base.OnCreate(bundle);

   string extraString;
   bool extraBool;

   if (bundle != null)
   {
      intentString = bundle.GetString("myString");
      intentBool = bundle.GetBoolean("myBool");
   }

   // Set our view from the "main" layout resource
   SetContentView(Resource.Layout.Main);
}
```

Once `OnCreate` has finished, Android will call `OnStart`.

OnResume

The system calls this method when the Activity is ready to start interacting with the user. Activities should override this method to perform tasks such as:

- ➔ Ramping up frame rates (a common task in game building)
- ➔ Starting animations
- ➔ Listening for GPS updates
- ➔ Display any relevant alerts or dialogs
- ➔ Wire up external event handlers.

As an example, the following code snippet shows how to initialize the camera:

```
public void OnResume()
{
   base.OnResume(); // Always call the superclass first.

   if (_camera==null)
```

```
    {
        // Do camera initializations here
    }
}
```

`OnResume` is important, because *any* operation that is done in `OnPause` should be un-done in `OnResume`, since it's the only lifecycle method that is guaranteed to be executed after `OnPause`, in the case of bringing the Activity back to life.

OnPause

This method is called when the system is about to put the Activity into the background or when the Activity becomes partially obscured. Activities should override this method if they need to:

- ➔ Commit unsaved changes to persistent data
- ➔ Destroy or clean up other objects consuming resources
- ➔ Ramp down frame rates and pausing animations
- ➔ Unregister external event handlers or notification handlers (i.e. those that are tied to a Service). This must be done to prevent Activity memory leaks.
- ➔ Likewise, if the Activity has displayed any dialogs or alerts, they must be cleaned up with the `.Dismiss()` method.

As an example, the following code snippet will release the camera, as the Activity cannot make use of it while Paused:

```
public void OnPause()
{
    base.OnPause(); // Always call the superclass first

    // Release the camera as other activities might need it
    if (_camera != null)
    {
        _camera.Release();
        _camera = null;
    }
}
```

There are two possible lifecycle methods that will be called after `OnPause`:

- ➔ `OnResume` will be called if the Activity is to be returned to the foreground
- ➔ `OnStop` will be called if the Activity is being placed in the background.

Summary

In this tutorial, we examined the constituent parts that make up an Android app, and showed how they work together. We also showed how Activities can be used to represent the screens in an Android application and how to use Intents to navigate between them. Then we introduced the `AndroidManifest.xml` file, and showed how Xamarin.Android uses C# attributes to make registering Activities in

the manifest more concise. We saw how to pass data from one Activity to another by using Intents. Finally, we briefly touched on how to use `ArrayAdapter` and `ListView` to display data in a list.