Android UI Controls

Xamarin Android Level 2, Chapter 1

Xamarin Inc.

Overview

Most applications rely on UI controls for user interaction. This chapter will cover discuss some of the common UI controls that are found in Android applications. This chapter will cover some of the more common controls that make up Android applications and how to use them. As many mobile applications are multi-threaded, we will also cover how to update the user interface from a background thread.

Common UI Controls

Android provides a lot of different controls that can be used to build user interface. Despite the differences between all these controls in behavior and appearance, they do all share some common features. Let's quickly discuss how to create user interfaces with the various user controls and how to safely update them from background threads.

Creating User Interfaces

There are two ways to add controls to a user interface:

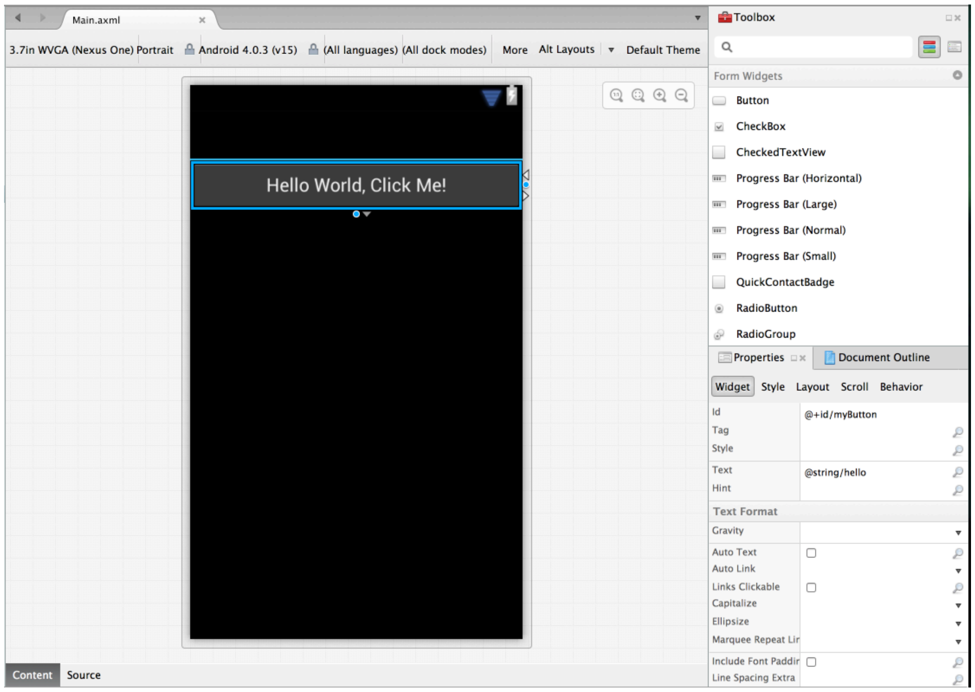
➔ Declaratively – With this technique a user interface is defined in an XML layout file. This layout file is then inflated by Android to create a View.

➔ Programatically – A View is constructed entirely by C# code.

Regardless of how a user interface is created, the controls that make up the user interface many be manipulated with C# in your code.

USING AN AXML LAYOUT FILE

There are two ways of creating the layout file, one way is to use the UI Designer from Mono for Android, as shown in the following screenshot:



Id @+id/myButton
Text @string/hello

The other approach is to manually edit the AXML file, adding controls and setting their properties. Below is a sample of the same layout file that is displayed in the designer above:

```
<?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">
    <Button
        android:id="@+id/myButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
</LinearLayout>
```

Once the layout file has been created, it is inflated by an Activity in the `OnCreate` lifecycle method. The following snippet shows how the layout file, created above, is inflated by an Activity:

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

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

        // Get our button from the layout resource,
        // and attach an event to it
        Button button = FindViewById<Button> (Resource.Id.myButton);

        button.Click += delegate {
                button.Text = string.Format ("{0} clicks!", count++);
        };
}
```

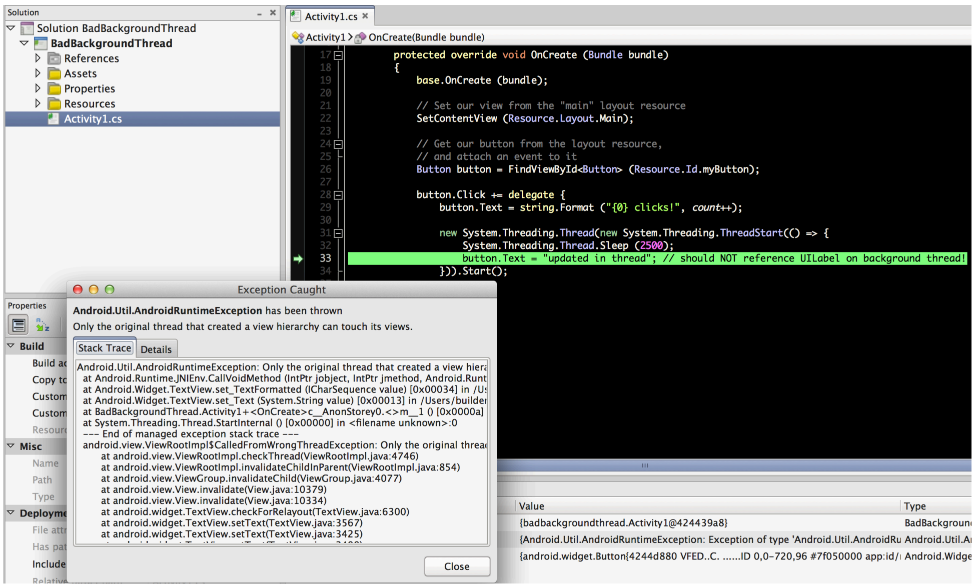
USING C#

When using C#, there is no layout file involved. Instead the various UI classes are instantiated. These objects are then added to a special view known as a *ViewGroup*. ViewGroups are views that can contain other views. There are many examples of a ViewGroup, the RelativeLayout and LinearLayout are two such examples. Once the view hierarchy has been built, a call to `SetContentView` is made which will install the ViewGroup as the layout of the Activity. Here is a simple example of creating the sample a user interface in C#:

```
protected override void OnCreate (Bundle bundle)
{
        base.OnCreate (bundle);
        var layout = new LinearLayout (this);

        var button = new Button (this);
        button.Text = Resources.GetString (Resource.String.hello);
        layout.AddView (button);

        button.Click += delegate {
                button.Text = string.Format ("{0} clicks!", count++);
        };

        SetContentView (layout);
}
```

About the UI Thread

Application user interfaces are always single threaded. Applications themselves may utilize multiple threads for tasks such as downloading files, but the user interface itself is always single-threaded. All changes to the user interface must be performed on the UI thread. Attempts by other threads to update the UI will result in an `Android.Util.AndroidRuntimeException` being thrown, as shown in the following screenshot:



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

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

    // Get our button from the layout resource,
    // and attach an event to it
    Button button = FindViewById<Button> (Resource.Id.myButton);

    button.Click += delegate {
        button.Text = string.Format ("{0} clicks!", count++);

        new System.Threading.Thread(new System.Threading.ThreadStart(() => {
            System.Threading.Thread.Sleep (2500);
            button.Text = "updated in thread"; // should NOT reference UILabel on background thread!
        })).Start();
```

Exception Caught

Android.Util.AndroidRuntimeException has been thrown

Only the original thread that created a view hierarchy can touch its views.

```
Android.Util.AndroidRuntimeException: Only the original thread that created a view hiera
  at Android.Runtime.JNIEnv.CallVoidMethod (IntPtr jobject, IntPtr jmethod, Android.Runt
  at Android.Widget.TextView.set_TextFormatted (ICharSequence value) [0x00034] in /Us
  at Android.Widget.TextView.set_Text (System.String value) [0x00013] in /Users/builder
  at BadBackgroundThread.Activity1+<OnCreate>c__AnonStorey0.<>m__1 () [0x0000a]
  at System.Threading.Thread.StartInternal () [0x00000] in <filename unknown>:0
  --- End of managed exception stack trace ---
  android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original threa
      at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:4746)
      at android.view.ViewRootImpl.invalidateChildInParent(ViewRootImpl.java:854)
      at android.view.ViewGroup.invalidateChild(ViewGroup.java:4077)
      at android.view.View.invalidate(View.java:10379)
      at android.view.View.invalidate(View.java:10334)
      at android.widget.TextView.checkForRelayout(TextView.java:6300)
      at android.widget.TextView.setText(TextView.java:3567)
      at android.widget.TextView.setText(TextView.java:3425)
```

Value	Type
{badbackgroundthread.Activity1@424439a8}	BadBackgroun
{Android.Util.AndroidRuntimeException: Exception of type 'Android.Util.AndroidRu	Android.Util.A
{android.widget.Button{4244d880 VFED..C. ......ID 0,0-720,96 #7f050000 app:id/	Android.Widge

If code executing on a background thread has update the user interface, then an Action delegate may be passed to `RunOnUiThread` like this:

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

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

    // Get our button from the layout resource,
    // and attach an event to it
    Button button = FindViewById<Button>(Resource.Id.myButton);

    button.Click += delegate
    {
        button.Text = string.Format("{0} clicks!", count++);

        new System.Threading.Thread(new System.Threading.ThreadStart(() =>
{
            System.Threading.Thread.Sleep(2500);
            RunOnUiThread(() => {
                button.Text = "updated in thread";});
        })).Start();

    };
}
```

You will not have to worry about threading for the remainder of the examples in this chapter, but it is an important concept to remember while developing applications.

Buttons

A button is a UI element that performs some action when the user touches it. There are three ways to add a button to a button to a layout:

➔ Text Only – This is the simplest and most common implementation of a button:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
... />
```

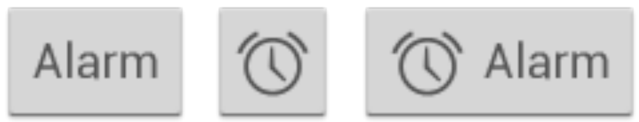
➔ Image Only – To display a Button with just an image an `ImageButton` should be used as shown in the following XML snippet:

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/button_icon"
    ... />
```

➔ Image and Text – To display a Button with both and image as text the Button class is used with the `text` and `drawableLeft` attributes set, as show below:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    android:drawableLeft="@drawable/button_icon"
    ... />
```

The following screen shot shows an example of each of the three possible combinations:



To respond to a user's touch it is necessary to respond to the Click event of the button as shown in the following example:

```
Button button = FindViewById<Button>(Resource.Id.myButton);
button.Click += delegate
{
    button.Text = string.Format("{0} clicks!", count++);
};
```

Custom Background

It is possible to redefine the appearance of a button by assigning a state-list drawable to the image background.

1. Create a new Xamarin.Android project, and copy the following three images from the file `CustomButton.zip` into the folder `/Resource/drawables`.

To ensure that the images fit all sizes of buttons, these are special bitmaps known as *NinePatch* images. NinePatch images are standard PNG images that Android can resize according to the View in which the image has been placed. The easiest way to create a NinePatch image is to use the Android Asset Studio.

2. The next step is to create a state-list drawable. Add a file to the folder `/Resource/drawables` called `android_button.xml`. Ensure that the **Build Action** of the file is set to **AndroidResource**. Edit the file so that it contains the following XML to the file:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:drawable="@drawable/android_pressed"
        android:state_pressed="true" />
  <item android:drawable="@drawable/android_focused"
        android:state_focused="true" />
  <item android:drawable="@drawable/android_normal" />
</selector>
```

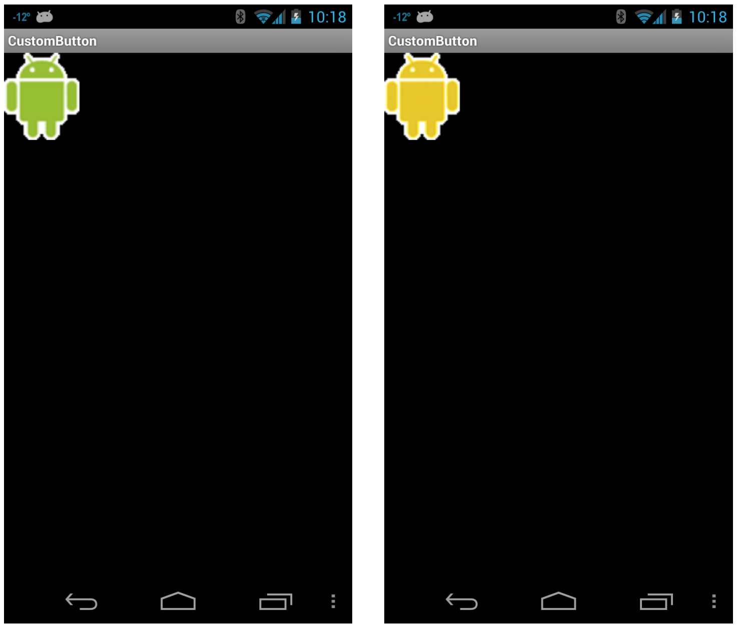
The above XML defines a state-list drawable, which will change the image displayed by the button as its state changes. The first `<item>` defines `android_pressed.png` as the image when the button is pressed (it's been activated); the second `<item>` defines `android_focused.png` as the image when the button is focused (when the button is highlighted using the trackball or directional pad); and the third `<item>` defines `android_normal.png` as the image for the normal state (when neither pressed nor focused).

Note: The order of the `<item>` elements is important. When this **drawable** is referenced, the `<item>`s are traversed in-order to determine which one is appropriate for the current button state. Because the "normal" image is last, it is only applied when the conditions `android:state_pressed` and `android:state_focused` have both evaluated false.

3. The final thing to do is to add a button and apply the drawable to the background of the button. Edit the file `/Resources/drawables/Main.axml`, as shown:

```
<?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">
    <Button
        android:id="@+id/myButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@drawable/android_button"
         />
</LinearLayout>
```

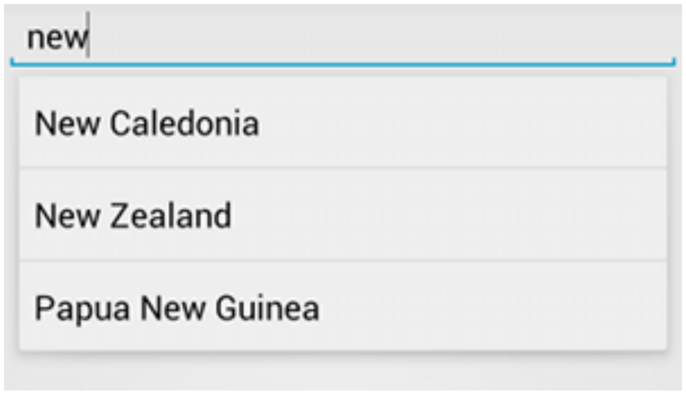
At this point it is possible to run the application and click on the button to see the state-list drawable in action. If you run the application, and touch the button, it should look something like the two screeshots:



Text Fields

There are three types of text fields in an Android application:

- ➔ TextView – This is a read-only view for displaying data to the user.
- ➔ EditText – This view allows the user to enter data into the application.
- ➔ AutoCompleteTextView – This control will provide suggestions to users as they type in text.



Each of these controls will be discussed in more detail below.

TextView

The `TextView` is a UI control that will display read-only data to a user. The following XML shows how to add a `TextView` to a layout file:

```
<TextView
    android:id="@+id/myTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize=”30sp”
    android:text="Hello!"
/>
```

Some of the more common properties for a `TextView`:

- ➔ android:textSize – this sets the size of the text. This must be a floating point number appended with one of the following units of measure: px (pixels), dp (density-independent pixels), sp (scaled pixels), in (inches), or mm (millimetres). The preferred unit of measure is scaled pixels.
- ➔ android:textStyle – sets the style of the text, can be `normal`, `bold`, or `italic`.
- ➔ android:textColor – This sets the color of the text.

The following code snippet shows how to add a TextView using C# code:

```
var layout = new LinearLayout(this); // this is the activity
var textView = new TextView(this);
textView.Text = "Hello world!";
textView.SetTextSize(Android.Util.ComplexUnitType.Sp, 30);
layout.AddView(textView);
```

There are more properties that can be set on a `TextView`. For more information, consult the TextView documentation.

EditText

Edit text is a control that collects text input from a user. The following XML snippet shows how to add an `EditText` control to a layout:

```
<EditText
        android:id="@+id/myEditText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        />
```

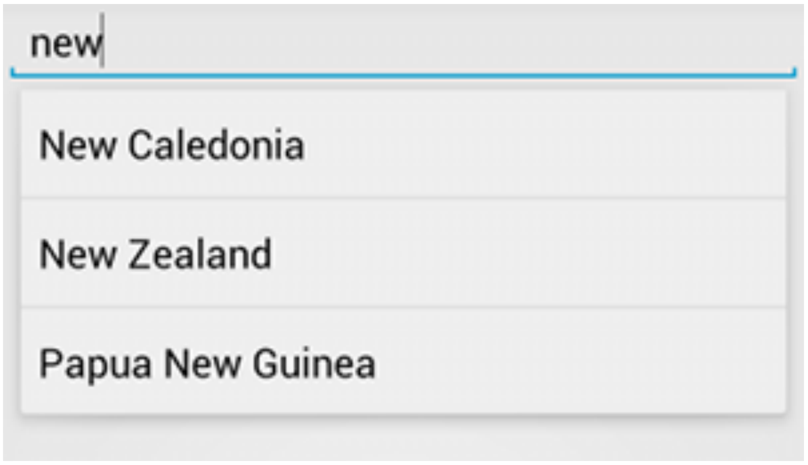
`EditText` does allow for some control over the type of data that can be entered in it by setting the `inputType` attribute on the control. For example, to specify a numeric keyboard for the keyboard:

```
    <EditText
        android:id="@+id/myEditText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:inputType="numberDecimal"
        />
```

There are many other types of keyboard that can be specified for an EditText. Please see the Android documentation for a list of possible android:inputType values.

AutoCompleteTextView

The `AutoCompleteTextView` is a specialized `EditText` that will provide suggestions to users based on the text that the user types. The suggestions will appear in a drop down list. The user may select an item from the list to replace the contents of the edit box with. And example of an `AutoCompleteTextView` can be seen in the following screenshot:



The following XML snippet is an example of how to add an `AutoCompleteTextView` to a layout.

```
<AutoCompleteTextView
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/autocomplete_scotch"
    android:layout_width="fill_parent"
    android:complettionThreshold="3"
    android:layout_height="wrap_content" />
```

In order to provide the suggestions to the user, an `AutoCompleteTextView` requires an Adapter. The following code snippet shows how to apply an instance of `ArrayAdapter<String>` to an `AutoCompleteTextView` so that it may provide the suggestions:

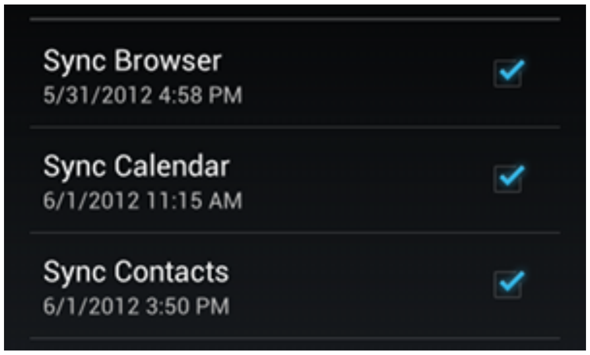
```
public class Activity1 : Activity
{
    private static readonly string[] COUNTRIES = new string[] {
        "New Caledonia", "France", "Italy", "Germany", "Spain", "New
Zealand", "Papau New Guinea"
    };

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

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

        var adapter = new ArrayAdapter<String>(this,
Android.Resource.Layout.SimpleDropDownItem1Line, COUNTRIES);
        var autoComplete =
FindViewById<AutoCompleteTextView>(Resource.Id.autocomplete_country);
```

```
            autoComplete.Adapter = adapter;
        }
    }
```

Consult the Android document to find out more about the AutoCompleteTextView.

CheckBoxes

Checkboxes are a control that allow users to select or unselect a value. The following screenshot shows a list of three checkboxes:



There are two ways to change the state of the checkbox, by using the `.Toggle()` method or using the `.Checked` property.

```
<?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">

    <CheckBox android:id="@+id/myCoolCheckbox"

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:text="My cool checkbox!" />

</LinearLayout>
```

The Activity that hosts the check box can implement code to respond to user clicks on the checkbox, as shown in the following code:

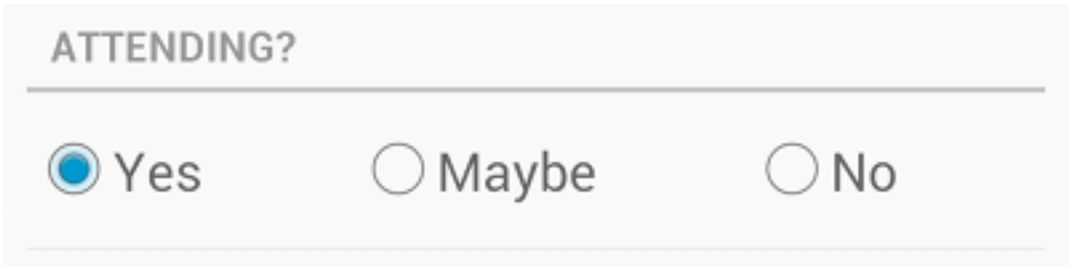
```
    var checkbox = FindViewById<CheckBox>(Resource.Id.myCoolCheckbox);
    checkbox.Click += (object sender, EventArgs e) => {
        var isChecked = ((CheckBox) sender).Checked;

        if (isChecked) {
            // Do something with the checked value.
        }
        else {
            // Do something because the value isn't checked.
        }
    };
```

The value of the check box can be changed programmatically by using the `.Toggle()` method or by setting the `.Checked` property as shown in the following code snippet:

```
checkbox.Checked = true;
checkbox.Toggle();
```

Radio Buttons

Radio buttons are similar to checkboxes in that they display a list of options in a set. Unlike checkboxes, radio buttons are logically grouped together and a user may only select on radio button out of the group. The following screen shot shows an example of this:



Radio buttons are added to a view layout by hosting them in a RadioGroup. RadioGroup is a sublass of LinearLayout. The following XML snippet shows an example of using RadioGroup and RadioButtons:

```
<RadioGroup xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <RadioButton android:id="@+id/radio_yes "
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Yes" />
    <RadioButton android:id="@+id/radio_maybe"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Maybe" />
    <RadioButton android:id="@+id/radio_no"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="No " />
</RadioGroup>
```

Initially, all of the radio buttons in the group will be unchecked. When a radio button is checked, any siblings in that group will automatically be unchecked.

There are two choices for responding to a user selection in the radio group:

➔ Controlling the RadioGroup – the RadioGroup class provides some methods and events for selecting a hosted radio button, or for ensuring that all radio buttons are un-selected

➔ Controlling Individual RadioButton's – similar to a CheckBox, you can set properties on a RadioButton

Controlling the RadioGroup

The first is to assign a handler to the `CheckChanged` event of the radio group. This event will be passed a instance of `RadioGroup.CheckedChangeEventArgs` that holds the resource id of the radio button is selected. The follow code snippet is an example of this:

```
var rg = FindViewById<RadioGroup>(Resource.Id.attendenceRadioGroup);
rg.CheckedChange += (object sender, RadioGroup.CheckedChangeEventArgs e)
=>  {
    switch (e.CheckedId)
    {
        case Resource.Id.radio_yes:
            // The user is going to the meeting;
            break;
        case Resource.Id.radio_maybe:
            // They might go
            break;
        case Resource.Id.radio_no:
            // The user is not going to the meeting.
            break;
    }
};
```

To select a radio button in the radio group, invoke the `.Checked` method and pass the resource id of the radio button to set:

```
rg.Check(Resource.Id.radio_maybe);
```

To unselect all the radio buttons, call the .ClearChecked() method on the radio group:

```
rg.ClearCheck();
```

Controlling Radio Buttons

The `RadioButton` class has a similar API to the `CheckBox` class. The RadioButton has a Clicked event:

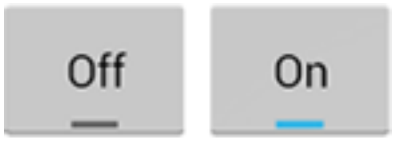
```
var yesRadioButton = FindViewById<RadioButton>(Resource.Id.radio_yes);
yesRadioButton.Click += (object sender, EventArgs e) => {
    var isChecked = ((RadioButton)sender).Checked;
    if (isChecked)
    {
        // the user is going to the meeting for sure
    }
};
```

It is possible to check and uncheck the radio button using the `.Checked` property and `.ClearCheck()` method:

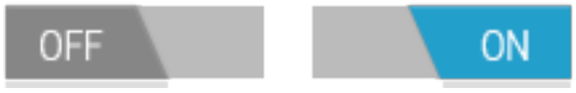
```
yesRadioButton.Checked = true;
yesRadioButton.Toggle();
```

ToggleButtons and Switches

Toggle buttons allow a user to change a setting between one of two states.



Android 4.0 introduced a new kind of toggle button called a switch. A switch appears and behaves more like a Slider control and allows the user to toggle between two states, such as OFF or ON.



Toggle Buttons

The following XML shows how to add a toggle button to a layout:

```
<ToggleButton
    android:id="@+id/togglebutton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="Vibrate on"
    android:textOff="Vibrate off"
 />
```

To respond to user clicks on the toggle button

```
var toggleButton = FindViewById<ToggleButton>(Resource.Id.togglebutton);
toggleButton.Click += (object sender, EventArgs e) => {
    var on = ((ToggleButton)sender).Checked;
    if (on)
    {
        // enable device vibration
    } else
    {
        // disable device vibration
    }
};
```

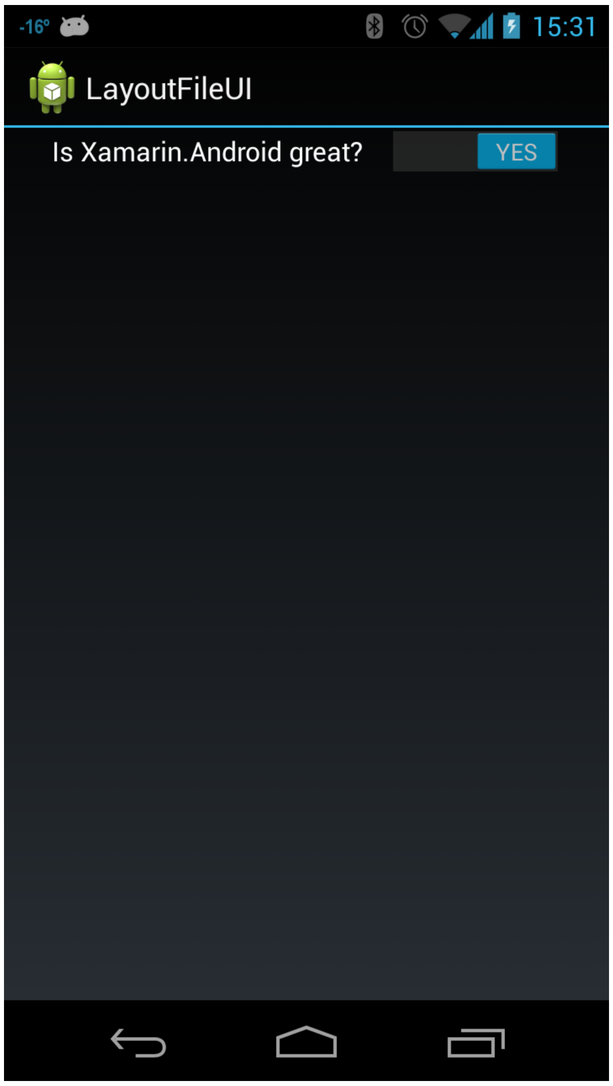
Switches

The following XML shows an example of how to add a switch to a layout:

```
<Switch android:text="Is Xamarin.Android great?"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
```

```
            android:textOn="YES"
            android:textOff="NO" />
```

This markup produces the following screenshot at runtime:

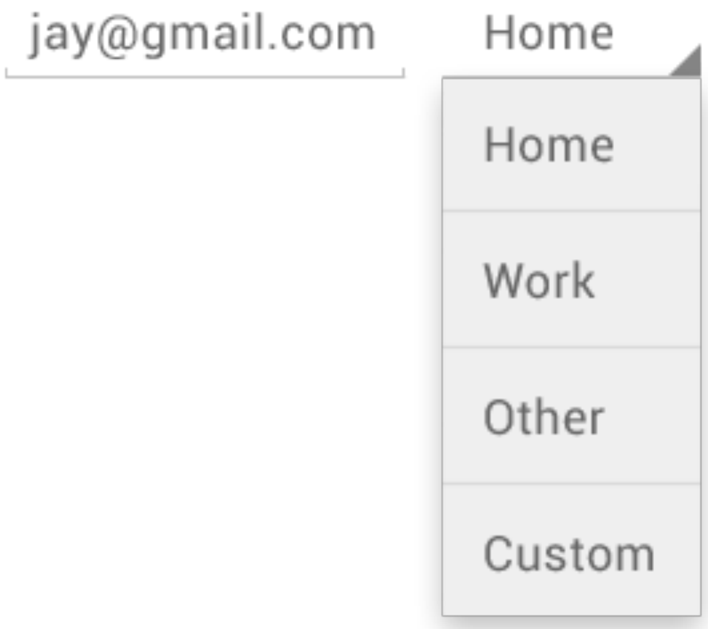


When the value of a switch is changed, it will raise a `CheckChange` event. The following example shows how to add a handler for the event and how to raise a `Toast` with a message based on the `IsChecked` value of the Switch:

```
Switch s = FindViewById<Switch> (Resource.Id.monitored_switch);

s.CheckedChange += delegate(object sender,
CompoundButton.CheckedChangeEventArgs e) {
    var toast = Toast.MakeText (this, "Your answer is " +
        (e.IsChecked ?  "correct" : "incorrect"), ToastLength.Short);
    toast.Show ();
};
```

Spinners

A spinner is very similar to a combo box or a select box in other UI frameworks. Normally the spinner will only display the selected value. When the user selects the spinner, it will display a drop-down list of possible values.



The following XML shows how to add a Spinner to a layout:

```
<Spinner
        android:id="@+id/spinner"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:prompt="@string/planet_prompt"
    />
```

To populate a spinner, you must specify an Adapter in your source code. The following code snippet shows how to use and initialize a spinner:

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    SetContentView(Resource.Layout.Main);

    var spinner = FindViewById<Spinner>(Resource.Id.spinner);
    var adapter = ArrayAdapter<String>.CreateFromResource(this,
Resource.Array.planets_array, Android.Resource.Layout.SimpleSpinnerItem);

adapter.SetDropDownViewResource(Android.Resource.Layout.SimpleSpinnerDropD
ownItem);
    spinner.Adapter = adapter;

    spinner.ItemSelected += (object sender,
AdapterView.ItemSelectedEventArgs e) => {
        var spinnner = (Spinner)  sender;
        var selectedPlanet =
(string)spinner.GetItemAtPosition(e.Position );
```

```
        Toast.MakeText(this, String.Format("The planet is {0}.",
selectedPlanet), ToastLength.Short).Show();
    } ;

}
```

This code requires a bit of explanation. First a reference to the spinner is obtained. Next an `ArrayAdapter<String>` is created from an array resource called `planets_array`. This array resource is defined in `/value/strings.xml`, the contents of which is in the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="planet_prompt">Choose a planet</string>
    <string-array name="planets_array">
        <item>Mercury</item>
        <item>Venus</item>
        <item>Earth</item>
        <item>Mars</item>
        <item>Jupiter</item>
        <item>Saturn</item>
        <item>Uranus</item>
        <item>Neptune</item>
    </string-array>
</resourc
```

The helper method `ArrayAdapter.CreateFromResource` is invoked, which will bind the string-array to an `ArrayAdapter<String>`. The built in layout `Android.Resource.Layout.SimpleSpinnerItem` is specified for the appearance of the spinner. We set the layout for each item in the spinner by calling `SetDropDownViewResource`, passing the built-in resource `Android.Resource.Layout.SimpleSpinnerItem`.

When the user selects a value from a spinner, the `ItemSelected` event is raised. In the code above, a lambda expression is provided as a handler for the event. Using the `Position` property on `ItemEventArgs`, we get a reference to the selected item in the `Spinner`. In this example, each item in the spinner is a `System.String` object, so we can just cast that and display a `Toast`.

WebView

The WebView control is used to embed a web browser in an Activity or to create your own web browser. The WebView control is useful for displaying HTML, but your application needs to behave more like a brower, then there is some extra work required.

Let us walk through adding a WebView to a layout and making it behave like a web browser.

1. First create a new Xamarin.Android project name `HelloWebView`.

2. Next we need to add the `INTERNET` permission to the application. To do this, edit the file `/Properties/AssemblyInfo.cs` and add the following line at the end:

```
 [assembly:
Android.App.UsesPermission(Android.Manifest.Permission.Internet)]
```

3. Now we need to add a WebView to a layout. Edit the file `/Resources/Layout/Main.axml` so that it contains the following XML

```
<?xml version="1.0" encoding="utf-8"?>
<WebView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/webview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" />
```

4. Edit the class `Activity1` and add the following instance variable to the class:

```
private Android.Webkit.WebView _webView;
```

5. Next we need to load the layout, so modify `OnCreate` so that it looks like the following code:

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    SetContentView(Resource.Layout.Main);

    _webView = FindViewById<Android.Webkit.WebView>(Resource.Id.webview);
    _webView.Settings.JavaScriptEnabled = true;
    _webView.LoadUrl(http://forums.xamarin.com);
}
```

This initializes the instance variable `_webView` and enables JavaScript on it, and then the initial web page is loaded. However, there are some problems with this example up to this point. When you click a link in the `WebView` control, the default Android browser handles the Intent instead of our `_webView`. To address this problem, we can create our own `WebViewClient` subclass which will enable our `_webView` to handle it's own URL requests. Lets do that next.

6. Add new class to the project called HelloWebViewClient with the following code:

```
class HelloWebViewClient : Android.Webkit.WebViewClient
{
    public override bool ShouldOverrideUrlLoading(Android.Webkit.WebView
view, string url)
    {
        view.LoadUrl(url);
        return true;
    }
}
```

This class will act as a handler for any URL's that will be requested by the WebView. Returning true says that method has handled the URL and that the event should not propagate.

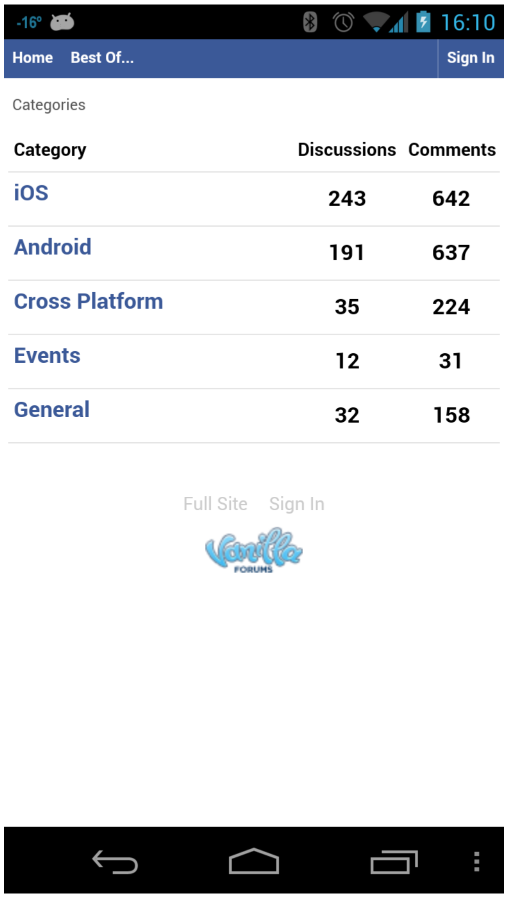
7. Next we need to ensure that `_webView` uses an instance of `HelloWebViewClient` to handle the URL requests. We need to edit `OnCreate` and add the following line anywhere after `_webView` is initialized:

```
_webView.SetWebViewClient(new HelloWebViewClient());
```

8. Now our Activity will handle URL requests, but when the user presses the BACK button it will close the Activity, and not navigate to the previous web page. To handle the BACK button key presses, we have to override `OnKeyDown` in the Activity. Edit the class Activity1 and add the following method:

```
public override bool OnKeyDown(Keycode keyCode, KeyEvent e)
{
    if (Keycode.Back.Equals(keyCode) && _webView.CanGoBack())
    {
        _webView.GoBack();
        return true;
    }
    return base.OnKeyDown(keyCode, e);
}
```

9. Finally, run the application. It should look something like the following sceenshot:



Categories

Category	Discussions	Comments
iOS	243	642
Android	191	637
Cross Platform	35	224
Events	12	31
General	32	158

With all these changes, our homemade web browser is complete. Congratulations!

Summary

In this chapter we learned how to use some common user interface controls to display information and collection user input. We have been introduced to TextViews, Spinners, ListViews, Buttons, Switches and the WebView. We also briefly discussed updating these UI controls from a background thread.