Chapter 6

Button clicks

The components of a graphical user interface can be divided roughly into views that are used for presentation, which display information to the user, and interaction, which obtain input from the user. While the Label is the most basic presentation view, the Button is probably the archetypal interactive view. The Button signals a command. It’s the user’s way of telling the program to initiate some action—to do something.

A Xamarin.Forms button displays text, with or without an accompanying image. (Only text buttons are described in this chapter; adding an image to a button is covered in Chapter 13, “Bitmaps.”) When the user’s finger presses on a button, the button changes its appearance somewhat to provide feedback to the user. When the finger is released, the button fires a Clicked event. The two arguments of the Clicked handler are typical of Xamarin.Forms event handlers:

- The first argument is the object firing the event. For the Clicked handler, this is the particular Button object that’s been tapped.
- The second argument sometimes provides more information about the event. For the Clicked event, the second argument is simply an EventArgs object that provides no additional information.

Once an application begins implementing user interaction, some special needs arise: The application should make an effort to save the results of that interaction if the program happens to be terminated before the user has finished working with it. For that reason, this chapter also discusses how an application can save transient data, particularly in the context of application lifecycle events. These are described in the section “Saving transient data.”

Processing the click

Here’s a program named ButtonLogger with a Button that shares a StackLayout with a ScrollView containing another StackLayout. Every time the Button is clicked, the program adds a new Label to the scrollable StackLayout, in effect logging all the button clicks:

```csharp
public class ButtonLoggerPage : ContentPage
{
    StackLayout loggerLayout = new StackLayout();

    public ButtonLoggerPage()
    {
        // Create the Button and attach Clicked handler.
        Button button = new Button
```
Chapter 6  Button clicks

```csharp
{
    Text = "Log the Click Time"
};
button.Clicked += OnButtonClicked;

this.Padding = new Thickness(5, Device.OnPlatform(20, 0, 0), 5, 0);

// Assemble the page.
this.Content = new StackLayout
{
    Children =
    {
        button,
        new ScrollView
        {
            VerticalOptions = LayoutOptions.FillAndExpand,
            Content = loggerLayout
        }
    }
};

void OnButtonClicked(object sender, EventArgs args)
{
    // Add Label to scrollable StackLayout.
    loggerLayout.Children.Add(new Label
    {
        Text = "Button clicked at " + DateTime.Now.ToString("T")
    });
}
```

In the programs in this book, event handlers are given names beginning with the word On, followed by some kind of identification of the view firing the event (sometimes just the view type), followed by the event name. The resultant name in this case is OnButtonClicked.

The constructor attaches the Clicked handler to the Button right after the Button is created. The page is then assembled with a StackLayout containing the Button and a ScrollView with another StackLayout, named loggerLayout. Notice that the ScrollView has its VerticalOptions set to FillAndExpand so that it can share the StackLayout with the Button and still be visible and scrollable.

Here's the display after several Button clicks:
As you can see, the Button looks a little different on the three screens. That’s because the button is rendered natively on the individual platforms: on the iPhone it’s a UIButton, on Android it’s an Android Button, and on Windows 10 Mobile it’s a Windows Runtime Button. By default the button always fills the area available for it and centers the text inside.

Button defines several properties that let you customize its appearance:

- **FontFamily of type** string
- **FontSize of type** double
- **FontAttributes of type** FontAttributes
- **TextColor of type** Color (default is Color.Default)
- **BorderColor of type** Color (default is Color.Default)
- **BorderWidth of type** double (default is 0)
- **BorderRadius of type** double (default is 5)
- **Image** (to be discussed in Chapter 13)

Button also inherits the BackgroundColor property (and a bunch of other properties) from VisualElement and inherits HorizontalOptions and VerticalOptions from View.

Some Button properties might work a little differently on the various platforms. As you can see, none of the buttons in the screenshots has a border. (However, the Windows Phone 8.1 button has a visible white border by default.) If you set the BorderWidth property to a nonzero value, the border
becomes visible only on the iPhone, and it’s black. If you set the BorderColor property to something other than Color.Default, the border is visible only on the Windows 10 Mobile device. If you want a visible border on both iOS and Windows 10 mobile devices, set both BorderWidth and BorderColor. But the border still won’t show up on Android devices unless you also set the BackgroundColor property. Customizing a button border is a good opportunity for using Device.OnPlatform (as you’ll see in Chapter 10, “XAML markup extensions”).

The BorderRadius property is intended to round off the sharp corners of the border, and it works on iOS and Android if the border is displayed, but it doesn’t work on Windows 10 and Windows 10 Mobile. The BorderRadius works on Windows 8.1 and Windows Phone 8.1, but if you use it with BackgroundColor, the background is not enclosed within the border.

Suppose you wrote a program similar to ButtonLogger but did not save the loggerLayout object as a field. Could you get access to that StackLayout object in the Clicked event handler?

Yes! It’s possible to obtain parent and child visual elements by a technique called walking the visual tree. The sender argument to the OnButtonClicked handler is the object firing the event, in this case the Button, so you can begin the Clicked handler by casting that argument:

```csharp
Button button = (Button)sender;
```

You know that the Button is a child of a StackLayout, so that object is accessible from the Parent property. Again, some casting is required:

```csharp
StackLayout outerLayout = (StackLayout)button.Parent;
```

The second child of this StackLayout is the ScrollView, so the Children property can be indexed to obtain that:

```csharp
ScrollView scrollView = (ScrollView)outerLayout.Children[1];
```

The Content property of this ScrollView is exactly the StackLayout you were looking for:

```csharp
StackLayout loggerLayout = (StackLayout)scrollView.Content;
```

Of course, the danger in doing something like this is that you might change the layout someday and forget to change your tree-walking code similarly. But the technique comes in handy if the code that assembles your page is separate from the code handling events from views on that page.

**Sharing button clicks**

If a program contains multiple Button views, each Button can have its own Clicked handler. But in some cases it might be more convenient for multiple Button views to share a common Clicked handler.

Consider a calculator program. Each of the buttons labeled 0 through 9 basically does the same
thing, and having 10 separate Clicked handlers for these 10 buttons—even if they share some common code—simply wouldn’t make much sense.

You’ve seen how the first argument to the Clicked handler can be cast to an object of type Button. But how do you know which Button it is?

One approach is to store all the Button objects as fields and then compare the Button object firing the event with these fields.

The TwoButtons program demonstrates this technique. This program is similar to the previous program but with two buttons—one to add Label objects to the StackLayout, and the other to remove them. The two Button objects are stored as fields so that the Clicked handler can determine which one fired the event:

```csharp
public class TwoButtonsPage : ContentPage
{
    Button addButton, removeButton;
    StackLayout loggerLayout = new StackLayout();

    public TwoButtonsPage()
    {
        // Create the Button views and attach Clicked handlers.
        addButton = new Button
        {
            Text = "Add",
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };
        addButton.Clicked += OnButtonClicked;

        removeButton = new Button
        {
            Text = "Remove",
            HorizontalOptions = LayoutOptions.CenterAndExpand,
            IsEnabled = false
        };
        removeButton.Clicked += OnButtonClicked;

        this.Padding = new Thickness(5, Device.OnPlatform(20, 0, 0), 5, 0);

        // Assemble the page.
        this.Content = new StackLayout
        {
            Children =
            {
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    Children =
                    {
                        addButton,
                        removeButton
                    }
                }
            }
        };
    }
}
```
Both buttons are given a `HorizontalOptions` value of `CenterAndExpand` so that they can be displayed side by side at the top of the screen by using a horizontal `StackLayout`:
Notice that when the `Clicked` handler detects `removeButton`, it simply calls the `RemoveAt` method on the `Children` property:

```csharp
loggerLayout.Children.RemoveAt(0);
```

But what happens if there are no children? Won’t `RemoveAt` raise an exception?

It can’t happen! When the `TwoButtons` program begins, the `IsEnabled` property of the `removeButton` is initialized to `false`. When a button is disabled in this way, a dim appearance signals to the user that it’s nonfunctional. It does not provide feedback to the user and it does not fire `Clicked` events. Toward the end of the `Clicked` handler, the `IsEnabled` property on `removeButton` is set to `true` only if the `loggerLayout` has at least one child.

This illustrates a good general rule: if your code needs to determine whether a button `Clicked` event is valid, it’s probably better to prevent invalid button clicks by disabling the button.

**Anonymous event handlers**

As with any event handler, you can define a `Clicked` handler as an anonymous lambda function. Here’s a program named `ButtonLambdas` that has a `Label` displaying a number and two buttons. One button doubles the number, and the other halves the number. Normally, the number and `Label` variables would be saved as fields. But because the anonymous event handlers are defined right in the constructor after these variables are defined, the event handlers have access to these local variables:

```csharp
public class ButtonLambdasPage : ContentPage
{
```
public ButtonLambdasPage()
{
    // Number to manipulate.
    double number = 1;

    // Create the Label for display.
    Label label = new Label
    {
        Text = number.ToString(),
        FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.CenterAndExpand
    };

    // Create the first Button and attach Clicked handler.
    Button timesButton = new Button
    {
        Text = "Double",
        FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Button)),
        HorizontalOptions = LayoutOptions.CenterAndExpand
    };
    timesButton.Clicked += (sender, args) =>
    {
        number *= 2;
        label.Text = number.ToString();
    };

    // Create the second Button and attach Clicked handler.
    Button divideButton = new Button
    {
        Text = "Half",
        FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Button)),
        HorizontalOptions = LayoutOptions.CenterAndExpand
    };
    divideButton.Clicked += (sender, args) =>
    {
        number /= 2;
        label.Text = number.ToString();
    };

    // Assemble the page.
    this.Content = new StackLayout
    {
        Children =
        {
            label,
            new StackLayout
            {
                Orientation = StackOrientation.Horizontal,
                VerticalOptions = LayoutOptions.CenterAndExpand,
                Children =
                {
                    timesButton,
                    divideButton
                }
            }
        }
    };
}
Notice the use of Device.GetNamedSize to get large text for both the Label and the Button. When used with Label, the second argument of GetNamedSize should indicate a Label, and when used with the Button it should indicate a Button. The sizes for the two elements might be different.

Like the previous program, the two buttons share a horizontal StackLayout:

![StackLayout with buttons and labels](image)

The disadvantage of defining event handlers as anonymous lambda functions is that they can’t be shared among multiple views. (Actually they can, but some messy reflection code is involved.)

**Distinguishing views with IDs**

In the TwoButtons program, you saw a technique for sharing an event handler that distinguishes views by comparing objects. This works fine when there aren’t very many views to distinguish, but it would be a terrible approach for a calculator program.

The Element class defines a StyleId property of type string specifically for the purpose of identifying views. It’s not used for anything internal to Xamarin.Forms, so you can set it to whatever is convenient for the application. You can test the values by using if and else statements or in a switch...
and case block, or you can use a Parse method to convert the strings into numbers or enumeration members.

The following program isn’t a calculator, but it is a numeric keypad, which is certainly part of a calculator. The program is called SimplestKeypad and uses a StackLayout for organizing the rows and columns of keys. (One of the intents of this program is to demonstrate that StackLayout is not quite the right tool for this job!)

The program creates a total of five StackLayout instances. The mainStack is vertically oriented, and four horizontal StackLayout objects arrange the 10 digit buttons. To keep things simple, the keypad is arranged with telephone ordering rather than calculator ordering:

```csharp
public class SimplestKeypadPage : ContentPage
{
    Label displayLabel;
    Button backspaceButton;

    public SimplestKeypadPage()
    {
        // Create a vertical stack for the entire keypad.
        StackLayout mainStack = new StackLayout
        {
            VerticalOptions = LayoutOptions.Center,
            HorizontalOptions = LayoutOptions.Center
        };

        // First row is the Label.
        displayLabel = new Label
        {
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            VerticalOptions = LayoutOptions.Center,
            HorizontalTextAlignment = TextAlignment.End
        };
        mainStack.Children.Add(displayLabel);

        // Second row is the backspace Button.
        backspaceButton = new Button
        {
            Text = String.Format(\u21E6),
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Button)),
            IsEnabled = false
        };
        backspaceButton.Clicked += OnBackspaceButtonClicked;
        mainStack.Children.Add(backspaceButton);

        // Now do the 10 number keys.
        StackLayout rowStack = null;

        for (int num = 1; num <= 10; num++)
        {
            if ((num - 1) % 3 == 0)
            {
rowStack = new StackLayout
{
    Orientation = StackOrientation.Horizontal
};
mainStack.Children.Add(rowStack);
}

Button digitButton = new Button
{
    Text = (num % 10).ToString(),
    FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Button)),
    StyleId = (num % 10).ToString()
};
digitButton.Clicked += OnDigitButtonClicked;

// For the zero button, expand to fill horizontally.
if (num == 10)
{
    digitButton.HorizontalOptions = LayoutOptions.FillAndExpand;
}
rowStack.Children.Add(digitButton);

this.Content = mainStack;

void OnDigitButtonClicked(object sender, EventArgs args)
{
    Button button = (Button)sender;
displayLabel.Text += (string)button.StyleId;
backspaceButton.IsEnabled = true;
}

void OnBackspaceButtonClicked(object sender, EventArgs args)
{
    string text = displayLabel.Text;
displayLabel.Text = text.Substring(0, text.Length - 1);
backspaceButton.IsEnabled = displayLabel.Text.Length > 0;
}

The 10 number keys share a single Clicked handler. The StyleId property indicates the number associated with the key, so the program can simply append that number to the string displayed by the Label. The StyleId happens to be identical to the Text property of the Button, and the Text property could be used instead, but in the general case, things aren’t always quite that convenient.

The backspace Button is sufficiently different in function to warrant its own Clicked handler, although it would surely be possible to combine the two methods into one to take advantage of any code they might have in common.

To give the keypad a slightly larger size, all the text is given a FontSize using NamedSize.Large. Here are the three renderings of the SimplestKeypad program:
Of course, you’ll want to press the keys repeatedly until you see how the program responds to a really large string of digits, and you’ll discover that it doesn’t adequately anticipate such a thing. When the Label gets too wide, it begins to govern the overall width of the vertical StackLayout, and the buttons start shifting as well.

Moreover, if the buttons contain letters or symbols rather than numbers, the buttons will be misaligned because each button width is based on its content.

Can you fix this problem with the Expands flag on the HorizontalOptions property? No. The Expands flag causes extra space to be distributed equally among the views in the StackLayout. Each view will increase additively by the same amount, but the buttons start out with different widths, and they will always have different widths. For example, take a look at the two buttons in the TwoButtons or ButtonLambdas program. Those buttons have their HorizontalOptions properties set to Fill-AndExpand, but they are different widths because the width of the button content is different.

A better solution for these programs is the layout known as the Grid, coming up in Chapter 17.

Saving transient data

Suppose you’re entering an important number in the SimplestKeypad program and you’re interrupted—perhaps with a phone call. Later on, you shut off the phone, effectively terminating the program.

What should happen the next time you run SimplestKeypad? Should the long string of numbers you entered earlier be discarded? Or should it seem as though the program resumed from the state
you last left it? Of course, it doesn’t matter for a simple demo program like SimplestKeypad, but in
the general case, users expect mobile applications to remember exactly what they were doing the last
time they interacted with the program.

For this reason, the Application class supports two facilities that help the program save and re-
store data:

- The Properties property of Application is a dictionary with string keys and object
  items. The contents of this dictionary are automatically saved prior to the application being ter-
  minated, and the saved contents become available the next time the application runs.

- The Application class defines three protected virtual methods, named OnStart, OnSleep,
  and OnResume, and the App class generated by the Xamarin.Forms template overrides these
  methods. These methods help an application deal with what are known as application lifecycle
  events.

To use these facilities, you need to identify what information your application needs to save so that
it can restore its state after being terminated and restarted. In general, this is a combination of applica-
tion settings—such as colors and font sizes that the user might be given an opportunity to set—and
transient data, such as half-entered entry fields. Application settings usually apply to the entire applica-
tion, while transient data is unique to each page in the application. If each item of this data is an entry
in the Properties dictionary, each item needs a dictionary key. However, if a program needs to save
a large file such as a word-processing document, it shouldn’t use the Properties dictionary, but in-
stead should access the platform’s file system directly. (That’s a job for Chapter 20, “Async and file
I/O.”)

Also, you should restrict the data types used with Properties to the basic data types supported by
.NET and C#, such as string, int, and double.

The SimplestKeypad program needs to save only a single item of transient data, and the dictionary
key “displayLabelText” seems reasonable.

Sometimes a program can use the Properties dictionary to save and retrieve data without getting
involved with application lifecycle events. For example, the SimplestKeypad program knows exactly
when the Text property of displayLabel changes. It happens only in the two Clicked event han-
dlers for the number keys and the delete key. Those two event handlers could simply store the new
value in the Properties dictionary.

But wait: Properties is a property of the Application class. Do we need to save the instance of
the App class so that code in the SimplestKeypadPage can get access to the dictionary? No, it’s not
necessary. Application defines a static property named Current that returns the current applica-
tion’s instance of the Application class.

To store the Text property of the Label in the dictionary, simply add the following line at the bot-
tom of the two Clicked event handlers in SimplestKeypad:

Don’t worry if the displayLabelText key does not yet exist in the dictionary: The Properties dictionary implements the generic IDictionary interface, which explicitly defines the indexer to replace the previous item if the key already exists or to add a new item to the dictionary if the key does not exist. That behavior is exactly what you want here.

The SimplestKeypadPage constructor can then conclude by initializing the Text property of the Label with the following code, which retrieves the item from the dictionary:

```csharp
IDictionary<string, object> properties = Application.Current.Properties;
if (properties.ContainsKey("displayLabelText"))
{
    displayLabel.Text = properties["displayLabelText"] as string;
    backspaceButton.IsEnabled = displayLabel.Text.Length > 0;
}
```

This is all your application needs to do: just save information in the Properties dictionary and retrieve it. Xamarin.Forms itself is responsible for the job of saving and loading the contents of the dictionary in platform-specific application storage.

In general, however, it’s better for an application to interact with the Properties dictionary in a more structured manner, and here’s where the application lifecycle events come into play. These are the three methods that appear in the App class generated by the Xamarin.Forms template:

```csharp
public class App : Application
{
    public App()
    {
    ...
    }

    protected override void OnStart()
    {
        // Handle when your app starts
    }

    protected override void OnSleep()
    {
        // Handle when your app sleeps
    }

    protected override void OnResume()
    {
        // Handle when your app resumes
    }
}
```

The most important is the OnSleep call. In general, an application goes into sleep mode when it no longer commands the screen and has become inactive (apart from some background jobs it might
have initiated). From this sleep mode, an application can be resumed (signaled by an OnResume call) or terminated. But this is important: After the OnSleep call, there is no further notification that an application is being terminated. The OnSleep call is as close as you get to a termination notification, and it always precedes a termination. For example, if your application is running and the user turns off the phone, the application gets an OnSleep call as the phone is shutting down.

Actually, there are some exceptions to the rule that a call to OnSleep always precedes program termination: a program that crashes does not get an OnSleep call first, but you probably expect that. But here’s a case that you might not anticipate: When you are debugging a Xamarin.Forms application, and use Visual Studio or Xamarin Studio to stop debugging, the program is terminated without a preceding OnSleep call. This means that when you are debugging code that uses these application lifecycle events, you should get into the habit of using the phone itself to put your program to sleep, to resume the program, and to terminate it.

When your Xamarin.Forms application is running, the easiest way to trigger an OnSleep call on a phone or simulator is by pressing the phone’s Home button. You can then bring the program back to the foreground and trigger an OnResume call by selecting the application from the home menu (on iOS devices or Android devices) or by pressing the Back button (on Android and Windows Phone devices).

If your Xamarin.Forms program is running and you invoke the phone’s application switcher—by pressing the Home button twice on iOS devices, by pressing the Multitask button on Android devices (or by holding down the Home button on older Android devices), or by holding down the Back button on a Windows Phone—the application gets an OnSleep call. If you then select that program, the application gets an OnResume call as it resumes execution. If you instead terminate the application—by swiping the application’s image upward on iOS devices or by tapping the X on the upper-right corner of the application’s image on Android and Windows Phone devices—the program stops executing with no further notification.

So here’s the basic rule: Whenever your application gets a call to OnSleep, you should ensure that the Properties dictionary contains all the information about the application you want to save.

If you’re using lifecycle events solely for saving and restoring program data, you don’t need to handle the OnResume method. When your program gets an OnResume call, the operating system has already automatically restored the program contents and state. If you want to, you can use OnResume as an opportunity to clear out the Properties dictionary because you are assured of getting another OnSleep call before your program terminates. However, if your program has established a connection with a web service—or is in the process of establishing such a connection—you might want to use OnResume to restore that connection. Perhaps the connection has timed out in the interval that the program was inactive. Or perhaps some fresh data is available.

You have some flexibility when you restore the data from the Properties dictionary to your application as your program starts running. When a Xamarin.Forms program starts up, the first opportunity you have to execute some code in the Portable Class Library is the constructor of the App class. At that
time, the Properties dictionary has already been filled with the saved data from platform-specific storage. The next code that executes is generally the constructor of the first page in your application instantiated from the App constructor. The OnStart call in Application (and App) follows that, and then an overridable method called OnAppearing is called in the page class. You can retrieve the data at any time during this startup process.

The data that an application needs to save is usually in a page class, but the OnSleep override is in the App class. So somehow the page class and the App class must communicate. One approach is to define an OnSleep method in the page class that saves the data to the Properties dictionary and then call the page’s OnSleep method from the OnSleep method in App. This approach works fine for a single-page application—indeed, the Application class has a static property named MainPage that is set in the App constructor and which the OnSleep method can use to get access to that page—but it doesn’t work nearly as well for multipage applications.

Here’s a somewhat different approach: You first define all the data you need to save as public properties in the App class, for example:

```csharp
public class App : Application
{
    public App()
    {
        ...
    }

    public string DisplayLabelText { set; get; }
    ...
}
```

The page class (or classes) can then set and retrieve those properties when convenient. The App class can restore any such properties from the Properties dictionary in its constructor prior to instantiating the page and can store the properties in the Properties dictionary in its OnSleep override.

That’s the approach taken by the PersistentKeypad project. This program is identical to SimplestKeypad except that it includes code to save and restore the contents of the keypad. Here’s the App class that maintains a public DisplayLabelText property that is saved in the OnSleep override and loaded in the App constructor:

```csharp
namespace PersistentKeypad
{
    public class App : Application
    {
        const string displayLabelText = "displayLabelText";

        public App()
        {
            if (Properties.ContainsKey(displayLabelText))
            {
                DisplayLabelText = (string)Properties[displayLabelText];
            }
        }
    }
}
```
MainPage = new PersistentKeypadPage(); }

public string DisplayLabelText { set; get; } 

protected override void OnStart()
{
    // Handle when your app starts 
}

protected override void OnSleep()
{
    // Handle when your app sleeps
    Properties[displayLabelText] = DisplayLabelText;
}

protected override void OnResume()
{
    // Handle when your app resumes 
}

}

To avoid spelling errors, the App class defines the string dictionary key as a constant. It’s the same as the property name except that it begins with a lowercase letter. Notice that the DisplayLabelText property is set prior to instantiating PersistentKeypadPage so that it’s available in the PersistentKeypadPage constructor.

An application with many more items might want to consolidate them in a class named AppSettings (for example), serialize that class to an XML or a JSON string, and then save the string in the dictionary.

The PersistentKeypadPage class accesses that DisplayLabelText property in its constructor and sets the property in its two event handlers:

public class PersistentKeypadPage : ContentPage
{
    Label displayLabel;
    Button backspaceButton;

    public PersistentKeypadPage()
    {

        // New code for loading previous keypad text.
        App app = Application.Current as App;
        displayLabel.Text = app.DisplayLabelText;
        backspaceButton.IsEnabled = displayLabel.Text != null &&
            displayLabel.Text.Length > 0;
    }
void OnDigitButtonClicked(object sender, EventArgs args)
{
    Button button = (Button)sender;
    displayLabel.Text += (string)button.StyleId;
    backspaceButton.IsEnabled = true;

    // Save keypad text.
    App app = Application.Current as App;
    app.DisplayLabelText = displayLabel.Text;
}

void OnBackspaceButtonClicked(object sender, EventArgs args)
{
    string text = displayLabel.Text;
    displayLabel.Text = text.Substring(0, text.Length - 1);
    backspaceButton.IsEnabled = displayLabel.Text.Length > 0;

    // Save keypad text.
    App app = Application.Current as App;
    app.DisplayLabelText = displayLabel.Text;
}

When testing programs that use the Properties dictionary and application lifecycle events, you’ll want to occasionally uninstall the program from the phone or simulator. Uninstalling a program from a device also deletes any stored data, so the next time the program is deployed from Visual Studio or Xamarin Studio, the program encounters an empty dictionary, as though it were being run for the very first time.