CHAPTER 4

User Interaction Using Controls

Users choose dates, times, text, integers, doubles, and other values on mobile devices by using tactile controls. Touch-sensitive screens have user interaction that differs slightly from mouse-driven UIs: most is done with the thumbs and forefingers on the touchscreen. From the user's standpoint, this results in a hands-on control-panel interface with switches, icons, sliders, keyboards, and pickers that sometimes look—but more important, feel—like physical mechanisms.

Chapter 2 covered some of the basic Xamarin.Forms views such as the `Label`, `Button`, and `Image`. In this chapter, you'll explore additional controls available on each platform, the gestures and events that make them work, and their outputs.

Many of the controls in this chapter are picker-style (pick a date, pick an option, pick a time, and so forth). These controls tend to look and work better when displayed in a *modal dialog box*, a pop-up box that overlays the app and maintains focus until dismissed by the user. Xamarin.Forms handles this for you by automatically placing pickers in modals. For Android and iOS, this chapter covers platform-specific ways to build modal picker dialog boxes.

This chapter is a gallery and a reference of the most commonly used selection controls.

Xamarin.Forms Views

Xamarin.Forms views can perform these basic input functions:

- `Picker` : A pop-up to select a value from a simple list
- `DatePicker`: A pop-up for selecting month, date, and year
- `TimePicker`: A pop-up for selecting hour, minute, and AM/PM
- `Stepper`: Increment/decrement buttons for discrete values
- `Slider`: Sliding input lever for continuous values
- `Switch`: Boolean on/off control

Android Controls

These are some of the primary Android selection widgets:

- `Spinner`: A simple drop-down list
- `DatePicker`: A control for selecting month, date, and year
- `TimePicker`: A control for selecting hour, minute, and AM/PM
- `SeekBar`: Sliding input lever for continuous values

- `CheckBox`: A standard Boolean check-box control
- `Switch`: A Boolean on/off switch
- `RadioButton`: Button groups for single or multiple selection

iOS Controls

iOS controls perform a range of user interactions:

- `UIPickerView`: A simple drop-down list in a spinner
- `UIDatePicker`: A date and/or time spinner
- `UIStepper`: Increment/decrement buttons for discrete values
- `UISlider`: Sliding input lever for continuous values
- `UISwitch`: Boolean on/off control

■ **Note** The iOS and Android controls for label, text view, button, scroll view, and image are out of scope for this book. Please consult an iOS or Android primer for those.

Xamarin.Forms Views

XAMARIN.FORMS

Xamarin.Forms views provide a range of controls that mimic and extend their iOS and Android counterparts. All of the views covered here allow selection and populate at least one property with a data value specified by the user, sometimes more. Let's look at each view in turn.

Xamarin.Forms views often provide the selected value in two places: a handler event property (for example, `e.NewValue`) provides the most recent value, and a general-use property on the view provides the selected value for use throughout the page. You will create two labels to display both of those values: `eventValue` and `pageValue`.

Create a new `ContentPage` called `Controls` and declare two `Label` views to hold the results of control selection:

```
public partial class Controls : ContentPage
{
    Label eventValue;
    Label pageValue;

    public Controls()
    {
        eventValue= new Label();
        eventValue.Text = "Value in Handler";
        pageValue = new Label();
        pageValue.Text = "Value in Page";
```

Create a StackLayout at the end of your Controls() constructor to assign to your page's Content property. Center all of the controls in the StackLayout by using HorizontalOptions = LayoutOptions.Center. All of the Xamarin.Forms examples in this chapter can be found in the source listing Controls.cs in the ControlExamples solution, shown in Listing 4-1 at the end of this section.

Remember to add each view to your StackLayout as you go!

■ **Tip** All example code solutions, including the XAML versions of these C# examples, can be found at the Apress web site (www.apress.com) or on GitHub at https://github.com/danhermes/xamarin-book-examples.

Picker

The Picker view provides a pop-up to select a value from a simple list.

■ **Note** The Picker view is used for quick selection of short words, phrases, or numbers. Complex lists with composite cells containing multiple strings and images are covered in the next chapter.

First, create the picker and give it a title:

```
Picker picker = new Picker
{
    Title = "Option",
    VerticalOptions = LayoutOptions.CenterAndExpand
};
```

Next, populate the list:

```
var options = new List<string> { "First", "Second", "Third", "Fourth" };

foreach (string optionName in options)
{
    picker.Items.Add(optionName);
}
```

Option names are placed into the list and then added to the Items collection in the picker.

The Entry view in Figure 4-1 starts as a data entry field, similar to Xamarin.Forms.Entry displaying the value of the Title property.

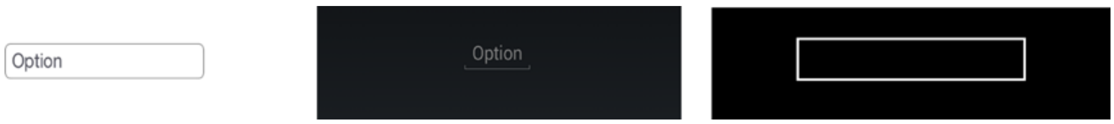


Figure 4-1. Entry views often have inline labels instead of side labels

When this field is tapped, a modal dialog appears, containing the list of items (Figure 4-2).

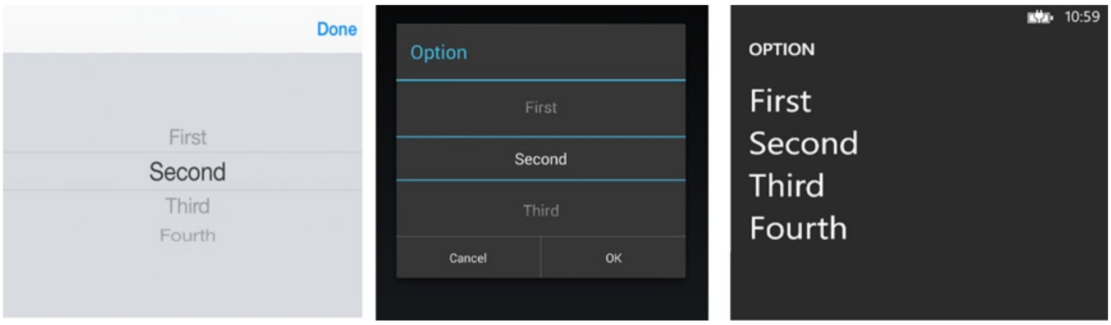


Figure 4-2. *Each picker looks a bit different, depending on the platform*

The list spins when swiped, and the highlighted value becomes the selected value. The selection is automatically populated into the original entry field so the user can see the effect of the change. The SelectedIndexChanged event also fires, which can be assigned a handler method or handled inline like this:

```
picker.SelectedIndexChanged += (sender, args) =>
{
    pageValue.Text = picker.Items[picker.SelectedIndex];
};
```

This implementation assigns the selected string to the Text property of the pageValue label.

■ **Tip** The selected index in the Picker.SelectedIndex property is a zero-based integer index. If Cancel is selected, the SelectedIndex remains unchanged.

DatePicker

XAMARIN.FORMS

The DatePicker view creates a pop-up for selection of month, date, and year. Create a DatePicker view like this:

```
DatePicker datePicker = new DatePicker
{
    Format = "D",
    VerticalOptions = LayoutOptions.CenterAndExpand
};
```

The Format property is set to D for the full month/day/year display. More date formats are provided later in this section.

The DatePicker view starts as a data entry field (Figure 4-3), similar to Xamarin.Forms.Entry displaying the value of the Date property.

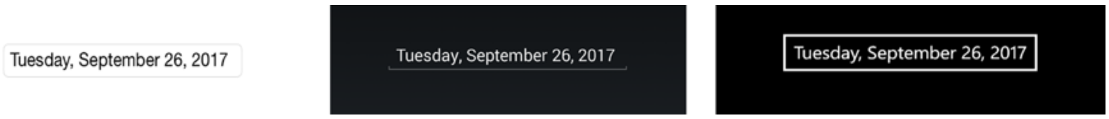

Tuesday, September 26, 2017
Tuesday, September 26, 2017
Tuesday, September 26, 2017

Figure 4-3. *DatePicker begins as an Entry view waiting for a tap*

When the date field is tapped, a modal dialog appears (Figure 4-4).

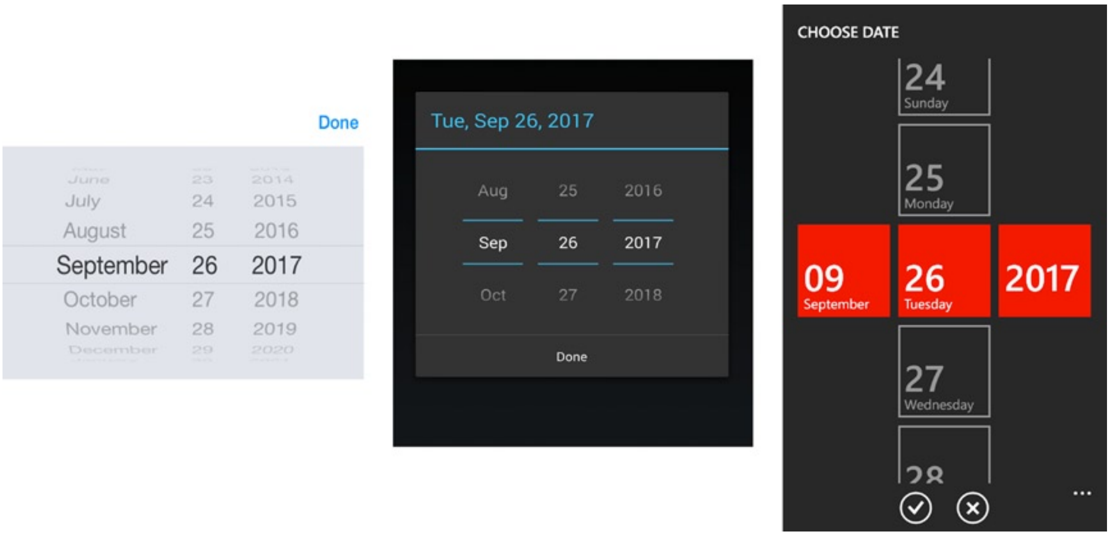


Figure 4-4. *DatePicker is a modal dialog*

Each column spins individually when swiped, and the highlighted values become the selected values. When Done is tapped, the selected date is automatically populated into the original entry field so the user can see the effect of the change. The `DateSelected` event also fires, which can be assigned a handler method or handled inline like this:

```
datePicker.DateSelected += (object sender, DateChangedEventArgs e) =>
{
        eventValue.Text = e.NewDate.ToString();
        pageValue.Text = datePicker.Date.ToString();
 };
```

The properties `e.OldDate` and `e.NewDate` are available within this event to provide the old and new selected date values. In general cases, however, the value entered by the user is stored in the `Date` property. All of these properties use type `DateTime`.

The format of the Date field is customizable with the Format property—for example, myDate.Format = "D". Other values are as follows:

- D: Full month, day, and year (Monday, March 5, 2018)
- d: Month, day, and year (3/5/2018)
- M: Month and day (March 5)
- Y: Month and year (March 2018)
- yy: Last two digits of the year (18)
- yyyy: Full year (2018)
- MM: Two-digit month (03)
- MMMM: Month (March)
- dd: Two-digit day (05)
- ddd: Abbreviated day of the week (Mon)
- dddd: Full day of the week (Monday)

You set a date range for selection by using MaximumDate and MinimumDate:

```
datePicker.MaximumDate = Convert.ToDateTime("1/1/2019");
datePicker.MinimumDate = Convert.ToDateTime("1/1/2014");
```

■ **Tip** On Android, the Format and MaximumDate/MinimumDate properties affect the DatePicker entry field but not the modal selection dialog at the time of this writing.

TimePicker

XAMARIN.FORMS

The TimePicker view creates a pop-up for selecting hour, minute, and AM/PM. Create a TimePicker view like this:

```
TimePicker timePicker = new TimePicker
{
    Format = "T",
    VerticalOptions = LayoutOptions.CenterAndExpand
};
```

The Format property set to T displays the full time. More time formats follow.

The TimePicker view starts as a data entry field similar to Xamarin.Forms.Entry, displaying the value of the Time property (Figure 4-5).

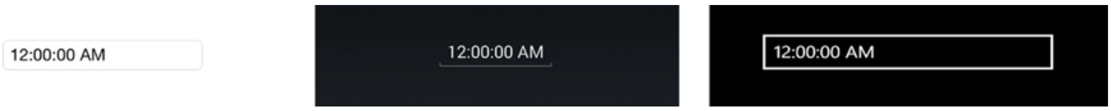


Figure 4-5. *TimePicker waits for a tap*

When the time field is tapped, a modal dialog appears (Figure 4-6).

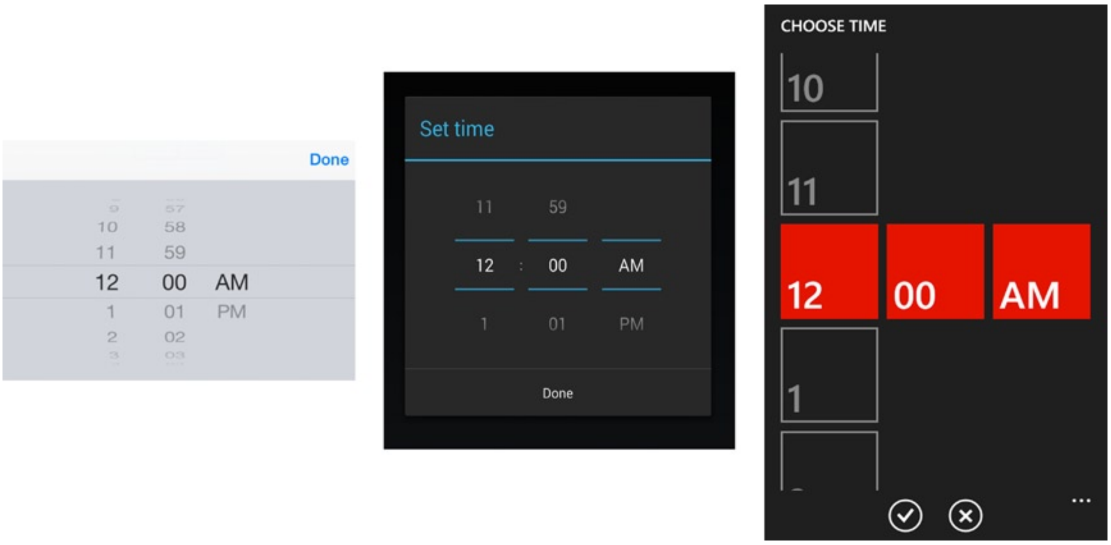


Figure 4-6. *TimePicker is a dialog box*

Each column spins individually when swiped, and the highlighted values become the selected values. When Done is tapped, the selected time is automatically populated into the original entry field so the user can see the effect of the change.

There is no TimeSelected event that triggers when a value is selected. Instead, use the `PropertyChanged` event in Xamarin.Forms data-binding to track changes to this view:

```
timePicker.PropertyChanged += (sender, e) =>
{
    if (e.PropertyName == TimePicker.TimeProperty.PropertyName)
    {
        pageValue.Text = timePicker.Time.ToString();
    }
};
```

The `timePicker.Time` property is set with the selected value as type `TimeSpan`.

The format of the `Time` field is customizable with the `Format` property (for example, `Format = "T"`). Other values are as follows:

- `T`: Full time with hours, minutes, seconds, and AM/PM (9:30:25 AM)
- `t`: Full time with hours, minutes, and AM/PM (9:30 AM)
- `hh`: Two-digit hours (09)
- `mm`: Two-digit minutes (30)
- `ss`: Two-digit seconds (25); seconds are not selectable in the dialog box
- `tt`: AM/PM designator (AM)

■ **Tip** `Format` affects the `TimePicker` entry field but not the dialog box at the time of this writing.

Stepper

XAMARIN.FORMS

The `Stepper` view creates increment and decrement buttons for discrete adjustments to the values:

```
Stepper stepper = new Stepper
{
    Minimum = 0,
    Maximum = 10,
    Increment = 1,
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.CenterAndExpand
};
```

`Minimum`, `Maximum`, and `Increment` properties are set for the steppable value. The start value can be set in the `Stepper.Value` property. Here is a shortcut constructor:

```
public Stepper (Double Minimum , Double  Maximum , Double  StartValue,
                Double Increment)
```

This constructor can be implemented like this:

```
Stepper stepper = new Stepper(0 ,10 ,0 , 1);
```

Figure 4-7 shows what the `Stepper` view looks like.

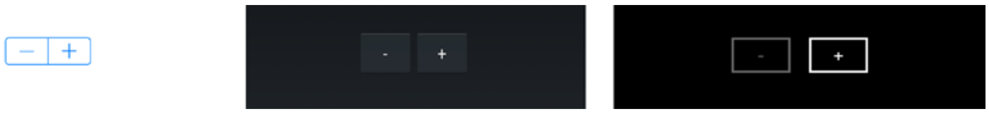


Figure 4-7. *Plus and minus for increment and decrement*

Tapping a plus or minus button changes the adjustable value and fires the Stepper.ValueChanged event. It can be handled in a method or inline, like so:

```
stepper.ValueChanged += (sender, e) =>
{
    eventValue.Text = String.Format("Stepper value is {0:F1}", e.NewValue);
    pageValue.Text = stepper.Value.ToString();
};
```

The properties e.OldValue and e. NewValue are available within this event to provide the old and new selected values. In general cases, however, the value entered by the user is stored in the Stepper's Value property. All these properties are type Double.

Slider

XAMARIN.FORMS

The Slider view is a sliding input control providing a continuum of selection:

```
Slider slider = new Slider
{
    Minimum = 0,
    Maximum = 100,
    Value = 50,
    VerticalOptions = LayoutOptions.CenterAndExpand,
    WidthRequest = 300
};
```

Minimum and Maximum properties are set for the slidable value. The start value can be set in the Slider.Value property. The value changes by increments by one-tenth of a unit (0.1) as the slider is moved. The WidthRequest property sets the width of the view without changing minimum or maximum values. Here is a shortcut constructor:

```
public Slider (Double Minimum , Double  Maximum , Double  StartValue)
```

This constructor can be implemented like this:

```
Slider  slider = new Slider  (0 ,100 ,50);
```

Figure 4-8 shows what the Slider view looks like (with Value = 100).

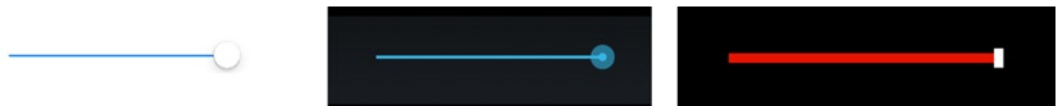


Figure 4-8. *Slider view at its max value*

Sliding the slider changes the adjustable value and fires the `slider.ValueChanged` event. It can be handled inline or as a method, like so:

```
slider.ValueChanged += (sender, e) =>
{
    eventValue.Text = String.Format("Slider value is {0:F1}", e.NewValue);
    pageValue.Text = slider.Value.ToString();
};
```

The properties `e.OldValue` and `e.NewValue` are available within this event to provide the old and new selected values. In general cases, the slidable value is also stored in the `slider.Value` property. All these properties are of type `Double`.

Switch

XAMARIN.FORMS

The `Switch` view is a Boolean on/off control:

```
Switch switcher = new Switch
{
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.CenterAndExpand
};
```

Figure 4-9 shows what the `Switch` view looks like off.

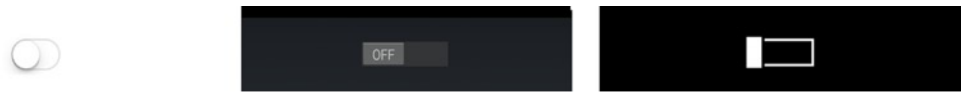


Figure 4-9. *Switch off*

And Figure 4-10 shows what the same view looks like on.

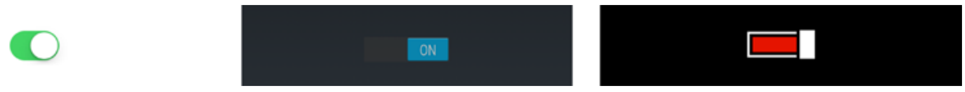


Figure 4-10. *Switch on*

Tapping the switch changes the Boolean value and fires the `Switch.Toggled` event. It can be handled inline or as a method, like so:

```
switcher.Toggled += (sender, e) =>
{
    eventValue.Text = String.Format("Switch is now {0}", e.Value);
    pageValue.Text = switcher.IsToggled.ToString();
};
```

The property `e.Value` is available within this event to provide the new switch value. In general cases, the value is also stored in the `Switch.IsToggled` property. These properties are of type `Boolean`.

Scale, Rotation, Opacity, Visibility, and Focus

XAMARIN.FORMS

You can alter the appearance and behavior of Xamarin.Forms views by using members of the `View` superclass, `VisualElement`.

You give focus to a view by using the `Focus()` method, which returns `true` if successful. This example sets focus on an `Entry` view (which pops up the keyboard):

```
var gotFocus = entry.Focus();
```

Here are some key properties that can be set on a view:

- `Scale`: Change the size of a view without affecting the views around it. The default value is 1.0.

```
switcher.Scale = 0.7;
```

- `IsVisible`: Make a view invisible, or visible again.

```
label.IsVisible = false;
```

- `IsEnabled`: Disable and reenable a view.

```
label.IsEnabled = false;
```

- `Opacity`: Fade a view in and out. The default value is 1.0.

```
label.Opacity = 0.5;
```

- `Rotation`: View rotation can be achieved on all axes by using the `Rotation`, `RotationX`, and `RotationY` properties. These rotate the view around the point set by `AnchorX` and `AnchorY`.

CODE COMPLETE: Xamarin.Forms Controls

XAMARIN.FORMS

Listing 4-1 contains the complete code listing for all Xamarin.Forms selection control examples in this chapter.

■ **XAML** The XAML version of this example can be found at the Apress web site (`www.apress.com`), or on GitHub at `https://github.com/danhermes/xamarin-book-examples`. The Xamarin.Forms solution for Chapter 4 is `ControlExamples.Xaml`.

Figure 4-11 shows the full screen.

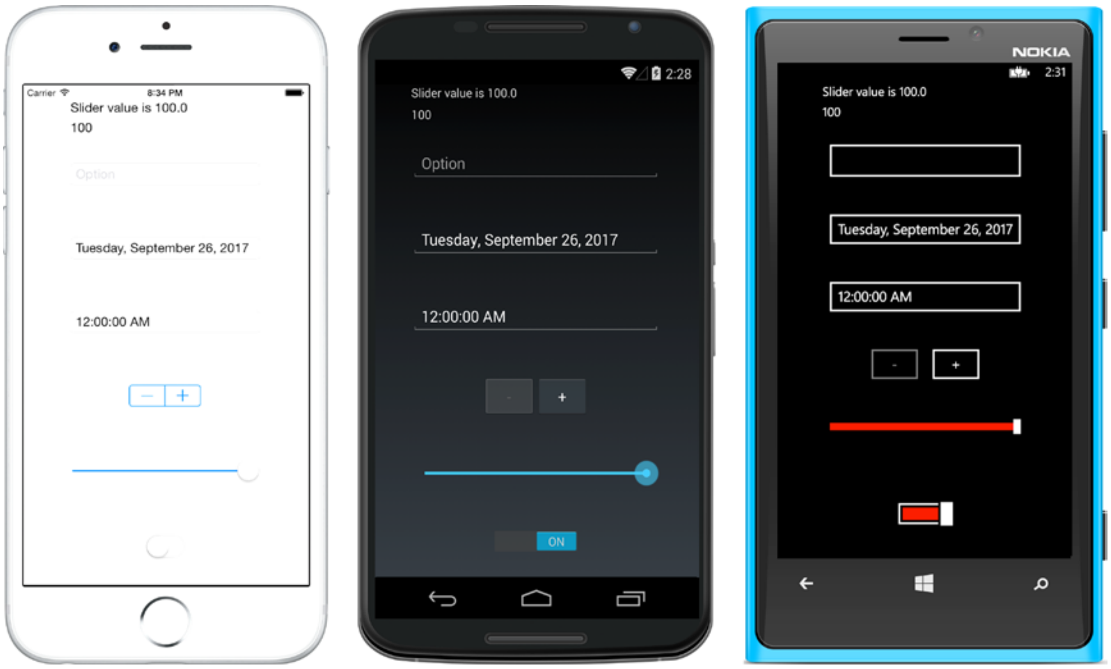


Figure 4-11. *Xamarin.Forms selection views*

Listing 4-1. Controls.cs in the ControlExamples Project of the ControlExamples Solution

```
public partial class Controls : ContentPage
   {
    Label eventValue;
    Label pageValue;

    public Controls()
    {
        eventValue= new Label();
        eventValue.Text = "Label";
        pageValue = new Label();
        pageValue.Text = "PageValue";

        Picker picker = new Picker
        {
            Title = "Option",
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        var options = new List<string> { "First", "Second", "Third", "Fourth" };
```

```
foreach (string optionName in options)
{
    picker.Items.Add(optionName);
}

picker.SelectedIndexChanged += (sender, args) =>
{
    pageValue.Text = picker.Items[picker.SelectedIndex];
};

DatePicker datePicker = new DatePicker
{
    Format = "D",
    VerticalOptions = LayoutOptions.CenterAndExpand
};

datePicker.DateSelected += (object sender, DateChangedEventArgs e) =>
{
    eventValue.Text = e.NewDate.ToString();
    pageValue.Text = datePicker.Date.ToString();
};

TimePicker timePicker = new TimePicker
{
    Format = "T",
    VerticalOptions = LayoutOptions.CenterAndExpand
};

timePicker.PropertyChanged += (sender, e) =>
{
    if (e.PropertyName == TimePicker.TimeProperty.PropertyName)
    {
        pageValue.Text = timePicker.Time.ToString();
    }
};

Stepper stepper = new Stepper
{
    Minimum = 0,
    Maximum = 10,
    Increment = 1,
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.CenterAndExpand
};
```

```
        stepper.ValueChanged += (sender, e) =>
        {
            eventValue.Text = String.Format("Stepper value is {0:F1}", e.NewValue);
            pageValue.Text = stepper.Value.ToString();
        };

        Slider slider = new Slider
        {
            Minimum = 0,
            Maximum = 100,
            Value = 50,
            VerticalOptions = LayoutOptions.CenterAndExpand,
            WidthRequest = 300
        };

        slider.ValueChanged += (sender, e) =>
        {
            eventValue.Text = String.Format("Slider value is {0:F1}", e.NewValue);
            pageValue.Text = slider.Value.ToString();
        };


        Switch switcher = new Switch
        {
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        switcher.Toggled += (sender, e) =>
        {
            eventValue.Text = String.Format("Switch is now {0}", e.Value);
            pageValue.Text = switcher.IsToggled.ToString();
        };

        this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 5);

        this.Content = new StackLayout {
            HorizontalOptions = LayoutOptions.Center,
            Children = {
                eventValue,
                pageValue,
                picker,
                datePicker,
                timePicker,
                stepper,
                slider,
                switcher
            }
        };
    }
}
```

■ **Note** Again, the two labels used in this example reflect the two ways in which selection values can be retrieved: in a handler event property (for example, `e.NewValue`), which provides the most recent value, or in a general-use property on the view, which provides the selected value for use throughout the page.

That completes our tour of Xamarin.Forms views!

It's time for your choice:

- If you're reading this book for *Xamarin.Forms only* and aren't yet interested in a platform-specific UI, you may want to jump to the beginning of the next chapter to continue reading about Xamarin.Forms.
- If you are ready to delve deeper into the platform-specific approach using Xamarin.Android and Xamarin.iOS, then read on!

Let's start with the Xamarin.Android controls before moving on to iOS. You can use the following controls when building Xamarin.Android platform-specific solutions or when creating Android custom renderers inside Xamarin.Forms solutions, as described in Chapter 8.

Android Controls

ANDROID

Some of the selection widgets used most often on Android are `Spinner`, `DatePicker`, `TimePicker`, `SeekBar`, `CheckBox`, `Switch`, and `RadioButton`. The `Spinner` is a simple drop-down selection picker, the `SeekBar` is a slider, and the rest of these controls are exactly what they sound like.

Unlike many Xamarin.Forms controls, Android views don't produce a modal dialog box by default, but produce only an inline dialog. Use the techniques described in the following sections or roll your own modals by using `DialogFragments` (see Chapter 6).

Create a new Android solution of type Blank App (Android) called `ControlExamplesAndroid`.

Spinner

A `Spinner` is an Android widget that provides a simple drop-down list of items to choose from.

Making a `Spinner` requires a few steps. You place a `Spinner` in your layout XML, and then create another layout that contains a `TextView` for binding to an `Adapter` to create a list to display. Next you instantiate the spinner in an activity, populate the list, and then bind the list to the `Spinner.Adapter` property and handle selection using the `Spinner.ItemSelected` event.

First, place a `Spinner` on a `LinearLayout` either using a designer or coded by hand in XML and call the layout `Spinner.axml` in the `Resources/layout` folder:

```
<Spinner
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:id="@+id/spinner" />
```

A `Spinner` is populated from an `Adapter`. In this example, an `ArrayAdapter` based on a hand-coded string array. An `ArrayAdapter` populates a list with strings, using a `TextView` for each list item.

Create a new layout to contain a TextView (also in the Resources/layout folder) that is used as a cell in the Spinner's list. Name this layout TextViewForSpinner.axml:

```
<?xml version="1.0" encoding="UTF-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/textItem"
    android:textSize="44sp"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
```

Create a new activity called SpinnerActivity.cs. In the OnCreate method, point to the spinner layout by using SetContentView, and find the spinner in the layout by using FindViewById:

```
SetContentView (Resource.Layout.Spinner);
Spinner spinner = FindViewById<Spinner> (Resource.Id.spinner);
```

Populate a string array with selectable options and construct the ArrayAdapter:

```
string[] options = {"one", "two", "three", "four", "five"} ;
ArrayAdapter adapter = new ArrayAdapter (this, Resource.Layout.TextViewForSpinner,
options);
spinner.Adapter = adapter;
```

Note how the ArrayAdapter constructor uses the TextView and the string array as parameters. The ArrayAdapter is assigned to the Adapter property of the spinner.

■ **Tip** Use the default resource, SimpleSpinnerItem, instead of manually creating TextViewForSpinner.axml in the Adapter declaration:

```
ArrayAdapter adapter = new ArrayAdapter (this, Resource.Layout.SimpleSpinnerItem, options);
```

■ **Tip** You can also set a default drop-down style by using SetDropDownViewResource:

```
adapter.SetDropDownViewResource (Android.Resource.Layout.SimpleSpinnerDropDownItem);
```

The spinner looks like an Entry view at first (Figure 4-12).

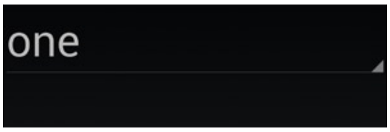


Figure 4-12. *Spinner anxiously awaiting a tap*

Tap it to display the drop-down list (Figure 4-13).

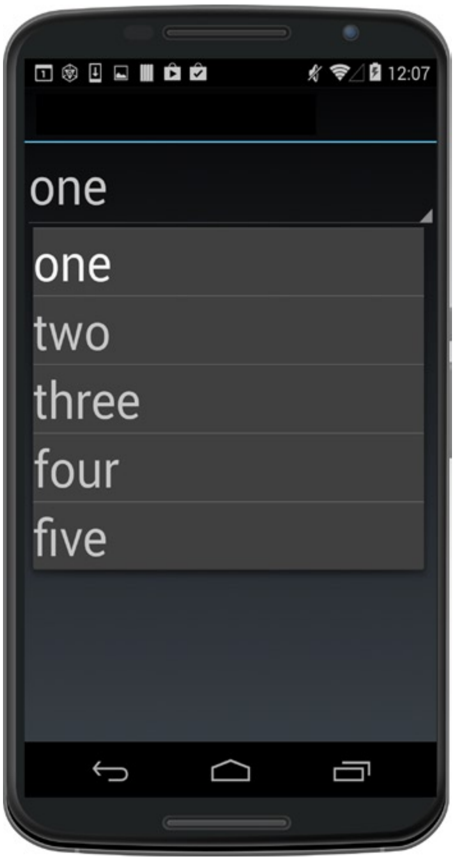


Figure 4-13. *Spinner is a dialog box*

Selecting an item fires the `ItemSelected` event, which can be handled inline or in a handler method like this:

```
spinner.ItemSelected += new EventHandler<AdapterView.ItemSelectedEventArgs>
(spinner_ItemSelected);
```

Here is the handler method that creates a toast to display the selected item:

```
        private void spinner_ItemSelected (object sender, AdapterView.ItemSelectedEventArgs e)
        {
            Spinner spinner = (Spinner)sender;

            string toast = string.Format ("Selection: {0}", spinner.GetItemAtPosition
            (e.Position));
            Toast.MakeText (this, toast, ToastLength.Long).Show ();
        }
```

Selecting an item pops up the toast, displaying the selected item shown in Figure 4-14.

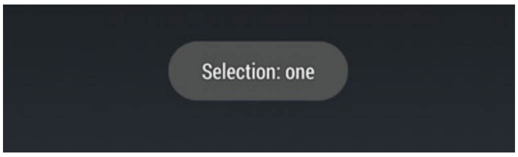


Figure 4-14. *A toast is Android's way of raising a glass for an important occasion*

■ **Note** Two other useful techniques for populating a spinner include an XML `<string-array>` and a data-bound adapter.

CODE COMPLETE: Spinner

ANDROID

The full `SpinnerActivity.cs` is shown in Listing 4-2. Refer to the downloadable source code for the layout XMLs.

Listing 4-2. SpinnerActivity.cs in the ControlExamplesAndroid Solution

```
[Activity (Label = "AndroidSelectionExamples", MainLauncher = true, Icon = "@drawable/icon")]
public class SpinnerActivity : Activity
{
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);
        SetContentView (Resource.Layout.Spinner);
        Spinner spinner = FindViewById<Spinner> (Resource.Id.spinner);
        spinner.ItemSelected += new EventHandler<AdapterView.ItemSelectedEventArgs>
        (spinner_ItemSelected);

        string[] options = {"one", "two", "three", "four", "five"} ;
        ArrayAdapter adapter = new ArrayAdapter (this,
            Resource.Layout.TextViewForSpinner, options);

        spinner.Adapter = adapter;
    }

    private void spinner_ItemSelected (object sender, AdapterView.ItemSelectedEventArgs e)
    {
        Spinner spinner = (Spinner)sender;

        string toast = string.Format ("Selection: {0}", spinner.GetItemAtPosition
        (e.Position));
        Toast.MakeText (this, toast, ToastLength.Long).Show ();
    }
}
```

DatePicker

ANDROID

The DatePicker is a spinner and/or calendar used for selecting the month, date, and year.

■ **Note** Modal dialogs are not a built-in function of Android pickers such as the DatePicker. Android pickers are inline dialogs by default. Modals must be coded by hand, in this case by using DatePickerDialog. You'll do this in the next section.

Add a DatePicker to your main layout by either using a designer or coding by hand in XML:

```
<DatePicker
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/datePicker" />
```

Instantiate a DatePicker in your activity. Match the widget ID name between the layout and activity:

```
DatePicker datePicker = FindViewById<DatePicker> (Resource.Id.datePicker);
```

The default DatePicker looks like Figure 4-15.

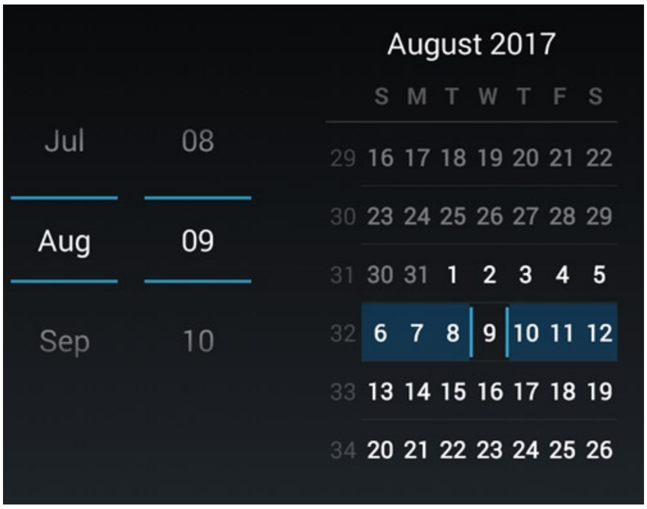


Figure 4-15. *The default DatePicker is a spinner and calendar*

Month and day columns can be scrolled and can accept a typed entry, enabling the user to search for a month or to enter the date. The calendar at right is scrollable, and the day is selectable.

This example creates an inline DatePicker, which consumes real estate and remains on the page even after the user is done with it. Typically, DatePickers are implemented as modal dialogs that allow the user to select a date value to fill a particular text field, then the dialog disappears after the selection is made.

Creating a Modal DatePicker by Using DatePickerDialog

The following example demonstrates a clickable text view that invokes the `DatePicker` as a modal dialog.

In a new layout called Picker.axml, add the `TextView` to a `LinearLayout` and name it `textView`, with the text `"Pick Date"`:

```
<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="Pick Date"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/textView" />
</LinearLayout>
```

In the activity, declare a `date` variable. Reference the `textView` and create a listener for the `Click` event that calls the `ShowDialog` method:

```
public class DatePickerActivity : Activity
{
        DateTime date;

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

            var textView = FindViewById<TextView>(Resource.Id.textView);

            textView.Click += delegate {
                ShowDialog (0);
            };

            date = DateTime.Today;
        }
```

Add a method to the activity that fires when `ShowDialog` is called and that instantiates `DatePickerDialog`:

```
protected override Dialog OnCreateDialog (int id)
{
    return new DatePickerDialog (this, HandleDateSet, date.Year, date.Month - 1,
    date.Day);
}
```

DatePickerDialog is populated with fields from the date variable. Last, handle changes to the date in DatePickerDialog by defining HandleDateSet:

```
void HandleDateSet (object sender, DatePickerDialog.DateSetEventArgs e)
{
    var textView = FindViewById<TextView>(Resource.Id.textView);
    date = e.Date;
    textView.Text = date.ToString("d");
}
```

This assigns the entered Date to the date variable and formats this to be placed into the textView's Text property. Run the solution to see the TextView in Figure 4-16.

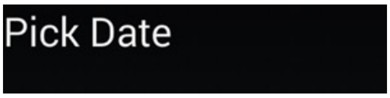


Figure 4-16. *textView patiently awaits a click*

Tap/click the TextView to pop up DatePickerDialog (Figure 4-17).

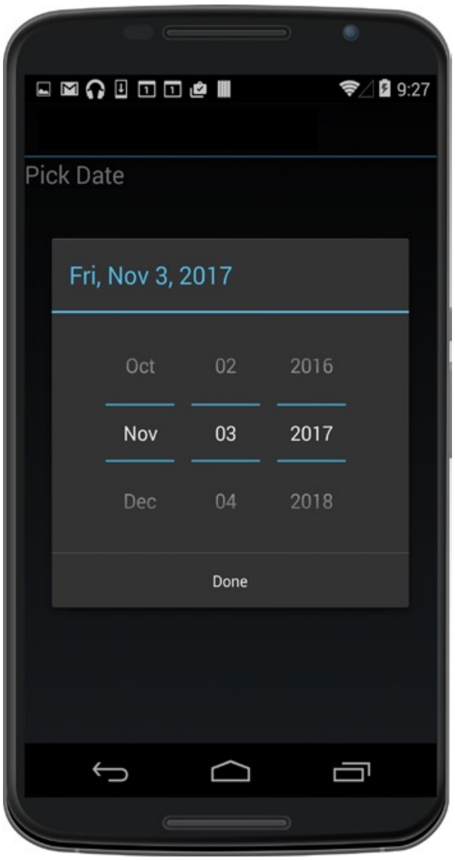
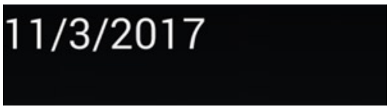


Figure 4-17. *DataPickerDialog modal dialog*

Spin the spinners, choose the date, and click Done. `textView` is updated with the selected date (Figure 4-18).



11/3/2017

Figure 4-18. *The TextView displays the selected date*

CODE COMPLETE: DatePickerExample

ANDROID

Listing 4-3 contains the activity code for the modal `DatePicker` dialog `DatePickerDialogExample` solution. Remember to add the `TextView` named `textView` to the main screen by using a designer.

Listing 4-3. DatePickerActivity.cs in the ControlExamplesAndroid Solution

```
[Activity (Label = "DatePickerDialogExample", MainLauncher = true)]
public class DatePickerActivity : Activity
    {
        DateTime date;

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

            SetContentView (Resource.Layout.Picker);

            var textView = FindViewById<TextView>(Resource.Id.textView);

            textView.Click += delegate {
                ShowDialog (0);
            };

            date = DateTime.Today;
        }

        protected override Dialog OnCreateDialog (int id)
        {
            return new DatePickerDialog (this, HandleDateSet, date.Year, date.Month - 1,
            date.Day);
        }

        void HandleDateSet (object sender, DatePickerDialog.DateSetEventArgs e)
        {
            var textView = FindViewById<TextView>(Resource.Id.textView);
            date = e.Date;
            textView.Text = date.ToString("d");
        }

    }
```

TimePicker

ANDROID

The TimePicker is a spinner for selecting hour, minute, and AM/PM.

■ **Note** Modal dialog boxes are not a built-in function of pickers such as the TimePicker and must be coded manually, in this case using the TimePickerDialog class. Use the modal technique described for DatePickerDialog in the previous section. See the full code in TimePickerActivity.cs in the ControlExamplesAndroid solution.

Add a TimePicker called timePicker to your layout:

```
<TimePicker
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/timePicker" />
```

Create a TimePicker class in your activity. Match the widget ID name between the layout and activity:

```
   TimePicker timePicker = FindViewById<TimePicker> (Resource.Id.timePicker);
```

Default time values can be set in the properties CurrentHour and CurrentMinute:

```
timePicker.CurrentHour = (Java.Lang.Integer) 17;
timePicker.CurrentMinute = (Java.Lang.Integer) 30;
```

The TimePicker looks like Figure 4-19.

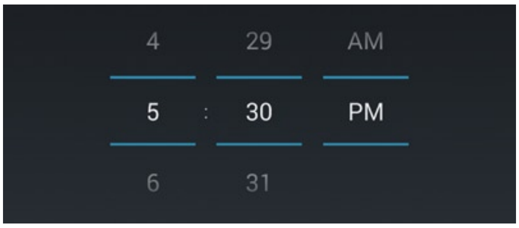
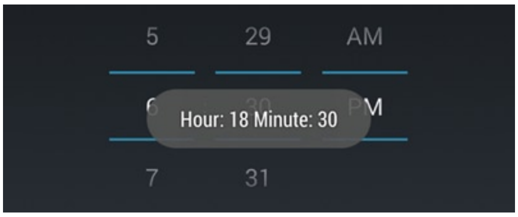


Figure 4-19. *TimePicker has columns of spinners*

All rows are scrollable. Changes to the control fire the TimeChanged event, which can be handled inline like this:

```
timePicker.TimeChanged += delegate(object sender, TimePicker.TimeChangedEventArgs e)
{
Toast.MakeText (this, "Hour: " + e.HourOfDay + " Minute: " + e.Minute, ToastLength.
Short).Show();
};
```

e.HourOfDay and e.Minute contain the selected time values. Figure 4-20 shows the toast.



Hour: 18 Minute: 30

Figure 4-20. *Cheers! It's that time*

SeekBar

ANDROID

A SeekBar is a sliding input lever for continuous values.

Add a SeekBar to your layout:

```
<SeekBar
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/seekBar"/>
```

Create a SeekBar class in your activity:

```
SeekBar seekBar = FindViewById<SeekBar> (Resource.Id.seekBar);
```

The SeekBar looks like Figure 4-21.

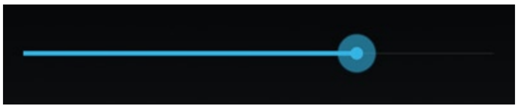


Figure 4-21. *SeekBar is a slider.*

■ **Tip** The minimum value of the SeekBar is zero.

When the SeekBar value is changed, the ProgressChanged event is fired, which can be handled inline like this:

```
seekBar.ProgressChanged += (object sender, SeekBar.ProgressChangedEventArgs e) => {
    if (e.FromUser)
    {
      Toast.MakeText (this, "Value: " + e.Progress, ToastLength.Short).Show ();
     }
};
```

The selection value is found in the `e.Progress` property. The `e.FromUser` property is `true` if the change was initiated by the user (not programmatically). Figure 4-22 shows the toast.

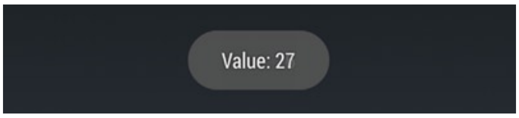


Figure 4-22. *The toast memorializes the value*

■ **Tip** The `SeekBar` can also be tracked with listeners by implementing the `SeekBar.IOnSeekBarChangeListener` interface. See `http://developer.xamarin.com/recipes/android/controls/seekbar/` for details.

■ **Tip** The default `SeekBar` range can be changed using `SeekBar.Max`. The default value is 100.

CheckBox

ANDROID

CheckBox is a standard Boolean check-box control, often deployed in groups.

Add CheckBoxes to your layout:

```
<CheckBox
    android:text="Option1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/checkBox1" />
<CheckBox
    android:text="Option2"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/checkBox2" />
```

These have the IDs checkBox1 and checkBox2. The Text property contains the displayed text string.
`Create CheckBox classes in your activity (usually in OnCreate()):`

```
CheckBox checkBox1 = FindViewById<CheckBox> (Resource.Id.checkBox1);
CheckBox checkBox2 = FindViewById<CheckBox> (Resource.Id.checkBox2);
```

Check boxes look like Figure 4-23.

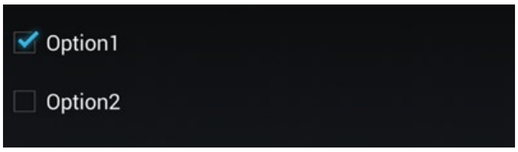


Figure 4-23. *Two separate, unconnected check boxes*

The Boolean selection value is stored in the CheckBox.Checked property of each CheckBox.

A Click event fires when a check box is tapped, which can be handled inline like this in the OnCreate() method:

```
checkBox1.Click += (o, e) => {
    if (checkBox1.Checked)
        Toast.MakeText (this, "Checked", ToastLength.Short).Show ();
    else
        Toast.MakeText (this, "Not checked", ToastLength.Short).Show ();
};
```

This handler displays a toast reflecting the value of checkBox1. See the rest of the code in Listing 4-4.

■ **Tip** For CheckBoxes to behave as a group (so only one can be selected at a time, for example) they must be coded by hand, by using each CheckBox's Click event and setting the value of each manually. For these situations, you should probably be considering RadioButtons, covered shortly.

Switch

ANDROID

A Switch is a Boolean on/off widget, often used to turn features on or off.

Add a Switch to your layout:

```
<Switch
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/switch1"
        android:layout_gravity="center_horizontal" />
```

Instantiate a Switch class in your activity:

```
Switch switch1 = FindViewById<Switch> (Resource.Id.switch1);
```

A Switch looks like Figure 4-24.

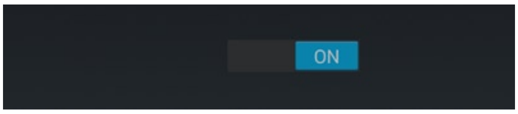


Figure 4-24. *Switch is on, waiting for a tap*

The Boolean value is stored in the `Switch.IsChecked` property. Tapping the control fires the CheckedChange event, which can be handled like this:

```
switch1.CheckedChange += delegate(object sender, CompoundButton.
CheckedChangeEventArgs e)
{
    var toast = Toast.MakeText (this, "Selection:" + (e.IsChecked ?  "On" : "Off"),
    ToastLength.Short);
    toast.Show ();
};
```

This pops up a toast with the switch value (Figure 4-25).

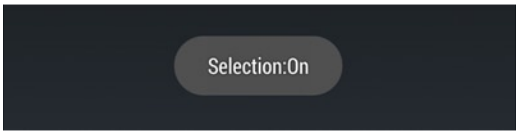

Selection:On

Figure 4-25. *Toast displaying Switch state*

Customizing with a Title, Switch Text, and State

A text title that appears before the switch, usually a feature or question, can be added in the `text` property. Default text on the switch itself can be changed by using the `textOn` and `textOff` properties (for example, to indicate Yes/No instead of On/Off). The default switch on/off state can be changed with the checked property:

```
<Switch android:text="Feature activated?"
        android:id="@+id/switch1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:textOn="YES"
        android:textOff="NO" />
```

This results in a customized switch (Figure 4-26).

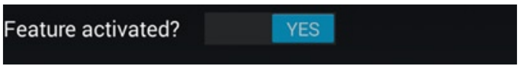


Figure 4-26. *The text property shows a label, and On/Off is now Yes/No*

RadioButton

ANDROID

A `RadioButton` is a selectable button widget that can be grouped with other `RadioButtons` by using a `RadioGroup`. This grouping makes selection mutually exclusive, so only one button can be selected at a time.

Create a RadioGroup containing RadioButtons in your layout:

```
<RadioGroup
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">
  <RadioButton android:id="@+id/radio1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="First" />
  <RadioButton android:id="@+id/radio2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Second" />
</RadioGroup>
```

Here we have two RadioButtons with the IDs radio1 and radio2. Their text properties are set to First and Second, respectively, and that is the text that will appear next to the buttons.

Create RadioButtons at the top of your activity outside OnCreate:

```
  RadioButton radio1;
  RadioButton radio2;
```

Inside OnCreate, find the buttons:

```
radio1 = FindViewById<RadioButton>(Resource.Id.radio1);
radio2 = FindViewById<RadioButton>(Resource.Id.radio2);
```

Figure 4-27 shows the result.

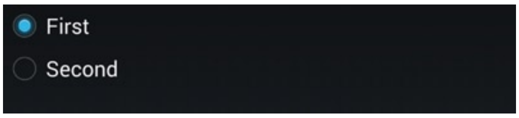


***Figure 4-27.** RadioGroup of two RadioButtons*

Tapping a RadioButton fires a Click event that can be handled like this:

```
radio1.Click += RadioButtonClick;
radio2.Click += RadioButtonClick;
```

Place the handler method outside the OnCreate method:

```
private void RadioButtonClick (object sender, EventArgs e)
{
    RadioButton rb = (RadioButton)sender;
    Toast.MakeText (this, rb.Text, ToastLength.Short).Show ();
}
```

This will toast the `text` property of the `RadioButton` when tapped (Figure 4-28).

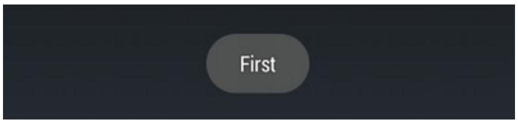


Figure 4-28. *Cheers, First!*

CODE COMPLETE: Android Controls

ANDROID

Listing 4-4 contains the activity code for the `SeekBar`, `CheckBox`, and `Switch` controls. For the `DatePicker` and `TimePicker` examples, see Listing 4-3 and the downloadable code solution, `ControlExamplesAndroid`.

Figure 4-29 displays all the selection controls covered here.

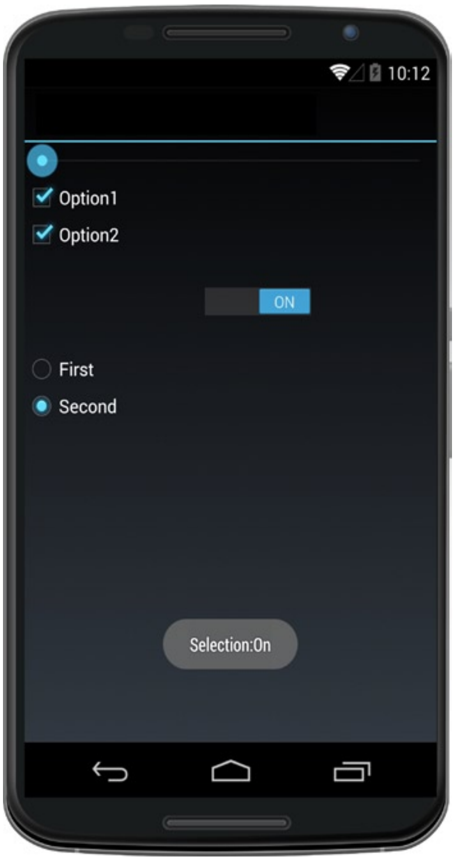


Figure 4-29. *Android selection controls*

Listing 4-4. SelectionActivity.cs in the ControlExamplesAndroid Solution

```
public class SelectionActivity : Activity
{

    RadioButton radio1;
    RadioButton radio2;

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

        SetContentView (Resource.Layout.Selection);

        SeekBar seekBar = FindViewById<SeekBar> (Resource.Id.seekBar);
        seekBar.ProgressChanged += (object sender, SeekBar.ProgressChangedEventArgs e) =>
        {
            if (e.FromUser)
            {
                Toast.MakeText(this, "Value: " + e.Progress, ToastLength.Short).Show();
            }
        };

        CheckBox checkBox1 = FindViewById<CheckBox> (Resource.Id.checkBox1);
        CheckBox checkBox2 = FindViewById<CheckBox> (Resource.Id.checkBox2);
        checkBox1.Click += (o, e) =>
        {
            if (checkBox1.Checked)
                Toast.MakeText(this, "Box 1 Checked", ToastLength.Short).Show();
            else
                Toast.MakeText(this, "Box 1 Not checked", ToastLength.Short).Show();
        };
        checkBox2.Click += (o, e) =>
        {
            if (checkBox2.Checked)
                Toast.MakeText(this, "Box 2 Checked", ToastLength.Short).Show();
            else
                Toast.MakeText(this, "Box 2 Not checked", ToastLength.Short).Show();
        };

        Switch switch1 = FindViewById<Switch> (Resource.Id.switch1);
        switch1.CheckedChange += delegate(object sender, CompoundButton.
        CheckedChangeEventArgs e)
        {
            var toast = Toast.MakeText(this, "Selection:" + (e.IsChecked ? "On" : "Off"),
            ToastLength.Short);
            toast.Show();
        };
```

```
            radio1 = FindViewById<RadioButton>(Resource.Id.radio1);
            radio2 = FindViewById<RadioButton>(Resource.Id.radio2);
            radio1.Click += RadioButtonClick;
            radio2.Click += RadioButtonClick;
        }

        private void RadioButtonClick(object sender, EventArgs e)
        {
            RadioButton rb = (RadioButton)sender;
            Toast.MakeText(this, rb.Text, ToastLength.Short).Show();
        }

    }
```

If you're ready to learn more about Android, then turn to Chapter 5 where you'll learn about lists.

For iOS controls, read on. Next let's explore Xamarin.iOS platform-specific controls. You can use these controls when building Xamarin.iOS platform-specific solutions or when creating iOS custom renderers inside Xamarin.Forms solutions, as described in Chapter 8.

iOS Controls

iOS

Some of most common iOS selection controls are `UIPickerView`, `UIDatePicker`, `UIStepper`, `UISlider`, and `UISwitch`. `UIPickerView` supplies a drop-down list, `UIStepper` gives a plus/minus button for changing numeric values, and the rest are self-explanatory.

Similar to Android, iOS controls don't produce a modal dialog box by default, but produce only an inline dialog. Use the `UITextField.InputView` modal dialog technique described in the following sections.

UIPickerView

The `UIPickerView` control provides a drop-down list for selection of a single item, typically from a short list of items obtained from a data model.

The `UIPickerView` is generally linked to a `UITextField` which, when tapped, pops up the picker. The picker is data-bound to a specialized view-model class called `UIPickerViewModel`, which returns rows from its contained data model and has a `ValueChanged` method and `SelectedItem` property that returns the selected value.

Let's build a drop-down that allows the user to pick a color.

Using the designer of your choice, add a `UITextField` to your layout (storyboard) and name it `color`. Then prepare the data model as a class `PickerModel` (Listing 4-5). Place it below the `ViewDidLoad` method in a `UIViewController` called `DatePickerViewController`.

Listing 4-5. PickerModel Provides Data for UIPickerView, ValueChangedEvent, and SelectedItem

```
public class PickerModel : UIPickerViewModel
{
    private readonly IList<string> items = new List<string>
    {
        "Red",
        "Blue",
        "Green",
        "Yellow",
        "Black"
    } ;

    public event EventHandler<EventArgs> ValueChanged;

    protected int selectedIndex = 0;

    public PickerModel()
    {
    }

    public string SelectedItem
    {
        get { return items[selectedIndex]; }
    }

    public override nint GetComponentCount (UIPickerView picker)
    {
        return 1;
    }

    public override nint GetRowsInComponent (UIPickerView picker, nint component)
    {
        return items.Count;
    }

    public override string GetTitle (UIPickerView picker, nint row, nint component)
    {
        return items[(int)row];
    }

    public override nfloat GetRowHeight (UIPickerView picker, nint component)
    {
        return 40f;
    }
    public override void Selected (UIPickerView picker, nint row, nint component)
    {
        selectedIndex = (int)row;
        if (this.ValueChanged != null)
        {
            this.ValueChanged (this, new EventArgs ());
        }
    }

}
```

Components are columns in the picker. This example has only one component/column. Making a selection fires the `Selected` event, which stores the selected value in `selectedIndex`, and then fires the `ValueChanged` event, which later handles the chosen value.

■ **Tip** `Nint` and `nfloat` are native iOS data types that were added in the Xamarin.iOS Unified API for 32- and 64-bit support.

In your `ViewDidLoad` method, create a `selectedColor` string to store the chosen value. Then instantiate the `PickerModel` and wire up the `model.ValueChanged` event, where `selectedColor` is populated by the selected item:

```
        public override void ViewDidLoad ()
        {
                string selectedColor = "";

                PickerModel model = new PickerModel();
                model.ValueChanged += (sender, e) => {
                        selectedColor = model.SelectedItem;
                } ;
```

Code the `UIPickerView` and set its properties, including assigning the `Model` and setting the default value of `color.Text`:

```
        UIPickerView picker = new UIPickerView();
        picker.ShowSelectionIndicator = true;
        picker.BackgroundColor = UIColor.White;
        picker.Model = model;

        this.color.Text = model.SelectedItem;
```

Now let's make that picker into a pop-up.

Making a UIPickerView into a Pop-up

iOS

A modal dialog box can be created several ways, the most common using `UITextField.InputView`. The `InputView` property of `UITextField` supports the assignment of an alternative "keyboard," which is a `UIPickerView` here. This keyboard can be decorated with additional controls, such as a toolbar with a Done button.

Still in the `ViewDidLoad` method, create a `UIToolbar`. Then add the Done button that populates the `textField` with the color when tapped:

```
UIToolbar toolbar = new UIToolbar();
toolbar.BarStyle = UIBarStyle.Default;
toolbar.Translucent = true;
toolbar.SizeToFit();

UIBarButtonItem doneButton = new UIBarButtonItem("Done", UIBarButtonItemStyle.Done,
    (s, e) => {
        this.color.Text = selectedColor;
        this.color.ResignFirstResponder();
    } );
toolbar.SetItems(new UIBarButtonItem[]{doneButton}, true);
```

The `ResignFirstResponder` method dismisses the keyboard from `textField`.

Assign the toolbar to the `textField`'s `InputAccessoryView` property.

```
this.color.InputAccessoryView = toolbar;
```

Last, associate your picker with the `UITextField` by using the `InputView` property. This causes the `UIPickerView` control to become a modal pop-up when the text control is tapped.

```
this.color.InputView = picker;
```

Clicking the text view pops up the picker (Figure 4-30).

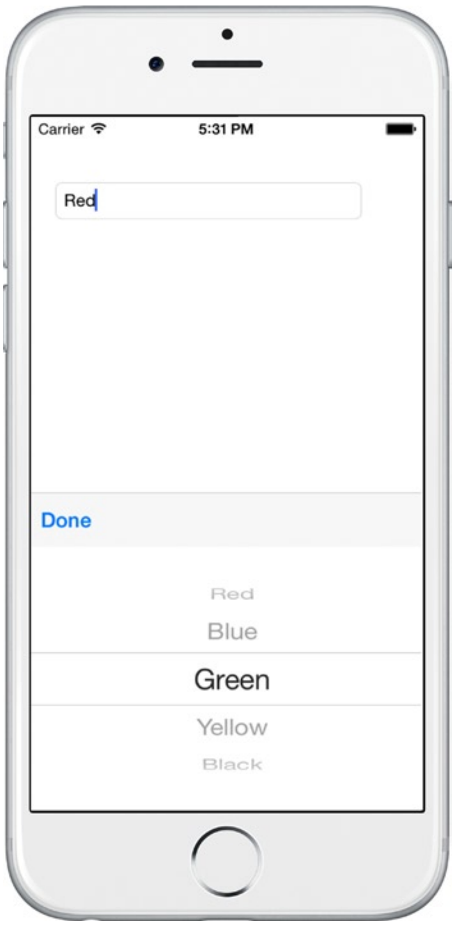


Figure 4-30. *Color text field with picker pop-up*

Tapping Done executes the action, which populates the text box with the selected color (Figure 4-31), and then closes the picker by using the `ResignFirstResponder()` method, dismissing the keyboard from `textField`.

Figure 4-31. *The color text field populated with the selected color*

■ **Note** See Chapter 6 for more on modals.

CODE COMPLETE: UIPickerView

iOS

Listing 4-6 shows the complete code example for `UIPickerView`. Remember to add the `UITextField` called `color` to your layout by using the designer tool.

Listing 4-6. PickerViewController.cs from the PickerExample solution

```
public partial class PickerViewController : UIViewController
    {
        public PickerViewController (IntPtr handle) : base (handle)
        {
        }

        public override void ViewDidLoad ()
        {
            base.ViewDidLoad ();

            string selectedColor = "";

            PickerModel model = new PickerModel();
            model.ValueChanged += (sender, e) => {
                selectedColor = model.SelectedItem;
            } ;

            UIPickerView picker = new UIPickerView();
            picker.ShowSelectionIndicator = true;
            picker.BackgroundColor = UIColor.White;
            picker.Model = model;

            this.color.Text = model.SelectedItem;

            UIToolbar toolbar = new UIToolbar();
            toolbar.BarStyle = UIBarStyle.Default;
            toolbar.Translucent = true;
            toolbar.SizeToFit();

            UIBarButtonItem doneButton = new UIBarButtonItem("Done",
            UIBarButtonItemStyle.Done,
                (s, e) => {
                    this.color.Text = selectedColor;
                    this.color.ResignFirstResponder();
                } );
            toolbar.SetItems(new UIBarButtonItem[]{doneButton}, true);

            this.color.InputAccessoryView = toolbar;
            this.color.InputView = picker;

        }
```

```
public class PickerModel : UIPickerViewModel
{
    private readonly IList<string> items = new List<string>
    {
        "Red",
        "Blue",
        "Green",
        "Yellow",
        "Black"
    } ;

    public event EventHandler<EventArgs> ValueChanged;

    protected int selectedIndex = 0;

    public PickerModel()
    {

    }

    public string SelectedItem
    {
        get { return items[selectedIndex]; }
    }

    public override nint GetComponentCount (UIPickerView picker)
    {
        return 1;
    }

    public override nint GetRowsInComponent (UIPickerView picker, nint component)
    {
        return items.Count;
    }

    public override string GetTitle (UIPickerView picker, nint row, nint component)
    {
        return items[(int)row];
    }

    public override nfloat GetRowHeight (UIPickerView picker, nint component)
    {
        return 40f;
    }
```

```
            public override void Selected (UIPickerView picker, nint row, nint component)
            {
                selectedIndex = (int)row;
                if (this.ValueChanged != null)
                {
                    this.ValueChanged (this, new EventArgs ());
                }
            }

        }
    }
```

■ **Tip** Modal pop-ups for iOS pickers can be produced by other methods. These require manual coding using `UIViewController` or `UIPopover`. Create a custom `UIViewController` class containing the picker, title label, and a Done button. Instantiate the modal `ViewController` and present the view with `await PresentViewControllerAsync(viewController, true);`.

UIDatePicker

iOS

The `UIDatePicker` is a spinner-style control used to select the date and time (Figure 4-32).

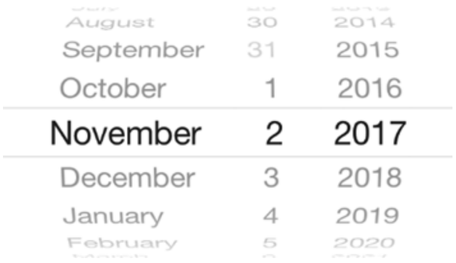


Figure 4-32. *UIDatePicker has columns of spinners*

The `UIDatePicker` is typically linked to a `UITextField` which, when tapped, pops up the picker.

Using the designer of your choice, create a `UITextField` on the layout and name it `textView`. Enter the name of the field in the `Text` property, in this case `Your Birthday`.

In the `ViewController's ViewDidLoad` method, code the `UIDatePicker` and set the `Mode` property to `Date`, to allow date-only selection:

```
        UIDatePicker datePicker = new UIDatePicker ();
        datePicker.Mode = UIDatePickerMode.Date;
        datePicker.BackgroundColor = UIColor.White;
```

■ **Tip** We'll cover other modes, such as `Time`, `DateAndTime`, and `CountDownTimer`, in the upcoming "Specify Which Fields to Display" subsection.

Restrict the range of entries by using the `MinimumDate` and `MaximumDate` properties:

```
        datePicker.MinimumDate = (NSDate)DateTime.Today.AddDays(-7);
        datePicker.MaximumDate = (NSDate)DateTime.Today.AddDays(7);
```

This example limits the date to a two-week range centered on today's date. Dates outside this range can be selected with the spinner, but the spinner will then spin back within the allowable range.

Next you need to make the date picker into a pop-up.

Making a UIDatePicker into a Pop-up

Like `UIPickerView`, a modal dialog box can be created several ways, the most common using `UITextField.InputView`. The `InputView` property of the `UITextField` supports the assignment of an alternative "keyboard" that is, in this case, a `UIDatePicker`. This keyboard can be decorated with additional controls, such as a toolbar with a Done button. Create a `UIToolbar` and add the Done button, which populates the `textField` with the date when tapped:

```
        UIToolbar toolbar = new UIToolbar();
        toolbar.BarStyle = UIBarStyle.Default;
        toolbar.Translucent = true;
        toolbar.SizeToFit();

        UIBarButtonItem doneButton = new UIBarButtonItem("Done", UIBarButtonItemStyle.Done,
            (s, e) => {
                DateTime dateTime = DateTime.SpecifyKind((DateTime)datePicker.Date,
                DateTimeKind.Unspecified);
                this.textField.Text = dateTime.ToString("MM-dd-yyyy");
                this.textField.ResignFirstResponder();
            } );
        toolbar.SetItems(new UIBarButtonItem[]{doneButton}, true);
```

`DateTime.SpecifyKind` returns a new date/time value in the proper format. The `ResignFirstResponder` method dismisses the keyboard from `textField`.

Assign the toolbar to the `textField`'s `InputAccessoryView` property:

```
        this.textField.InputAccessoryView = toolbar;
```

Last, associate your date picker with the `UITextField` by using the `InputView` property. This causes the `UIDatePicker` control to become a modal pop-up when the text control is tapped:

```
        this.textField.InputView = datePicker;
```

Clicking on the text view pops up the date picker (Figure 4-33).

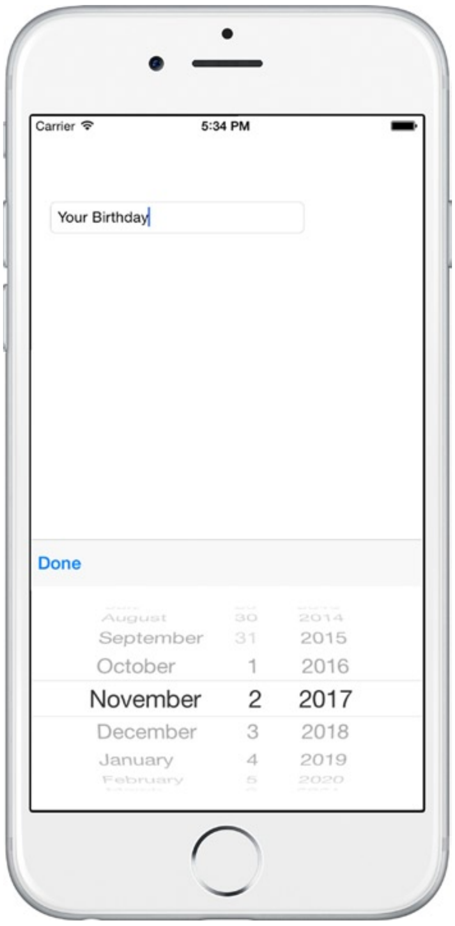


Figure 4-33. *textField with popped-up datePicker*

Tapping the Done option executes the action, which populates the text box with the formatted date (Figure 4-34), and closes the picker by using the `ResignFirstResponder()` method to dismiss the keyboard from `textField`.

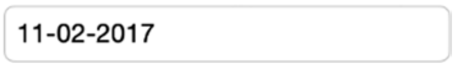


Figure 4-34. *The textField populated with a formatted date*

Various fields can appear in the `UIDatePicker` selection. Let's look at those next.

Specify Which Fields to Display

Specify the fields used in `UIDatePicker` with the `Mode` property, which uses the `UIDatePickerMode` enumerator. For example, `datePicker.Mode = UIDatePickerMode.`**`Date`**`;`

Use these `UIDatePickerModes` to specify the indicated fields:

- `Time`: Select the time only
- `Date`: Select the date only
- `DateAndTime`: Select both the date and time
- `CountDownTimer`: Select only hours and minutes

■ **Tip** When in `CountDownTimer` mode, the `CountDownDuration` property contains the total number of seconds of the selected time.

CODE COMPLETE: UIDatePicker

iOS

Listing 4-7 shows the complete `UIDatePicker` example. Remember to add `textField` to your layout by using the designer tool.

Listing 4-7. DatePickerViewController.cs from the DatePickerExample solution

```
public partial class DatePickerViewController : UIViewController {

public override void ViewDidLoad ()
{
    base.ViewDidLoad ();
    UIDatePicker datePicker = new UIDatePicker ();
    datePicker.Mode = UIDatePickerMode.Date;
    datePicker.BackgroundColor = UIColor.White;

    datePicker.MinimumDate = (NSDate)DateTime.Today.AddDays(-7);
    datePicker.MaximumDate = (NSDate)DateTime.Today.AddDays(7);

    UIToolbar toolbar = new UIToolbar();
    toolbar.BarStyle = UIBarStyle.Default;
    toolbar.Translucent = true;
    toolbar.SizeToFit();
    UIBarButtonItem doneButton = new UIBarButtonItem("Done", UIBarButtonItemStyle.Done,
        (s, e) => {
            DateTime dateTime = DateTime.SpecifyKind((DateTime)datePicker.Date,
            DateTimeKind.Unspecified);
            this.textField.Text = dateTime.ToString("MM-dd-yyyy");
            this.textField.ResignFirstResponder();
        } );
    toolbar.SetItems(new UIBarButtonItem[]{doneButton}, true);

    this.textField.InputAccessoryView = toolbar;
    this.textField.InputView = datePicker;
}
```

UIStepper

iOS

The UIStepper is an increment/decrement button for discrete values and is useful when slight changes to a value are needed. Add a UIStepper to the view by using your designer of choice.

Position the button and name it stepper in the Properties view. When run, it should look like Figure 4-35.

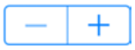


Figure 4-35. *Stepper*

Set the range of values the control will accept by using the MinimumValue and MaximumValue properties:

```
        stepper.MinimumValue = 0;
        stepper.MaximumValue = 11;
```

Add a UILabel named stepperLabel above the stepper to display the stepper value. Tapping the control fires the ValueChanged event, which can be handled like this:

```
        stepper.ValueChanged += (object sender, EventArgs e) => stepperLabel.Text =
        stepper.Value.ToString ();
```

The value of the stepper is contained in the Value property. Run it and click the plus button, clicked up from 0 to 8, to look like Figure 4-36.

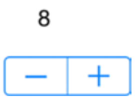


Figure 4-36. *stepperLabel and stepper*

Keep clicking plus, and the value increases to 11.

The StepValue property will increment the stepper by the specified value. Increment by two at a time (0, 2, 4, and so forth), like this:

```
stepper.StepValue = 2;
```

The experience of using the stepper can be tweaked by using these Boolean properties:

- AutoRepeat: If true, holding down the stepper button changes the value repeatedly. Default = true.

  ```
  stepper.AutoRepeat = true;
  ```

- Continuous: If true, all value changes fire an event. If false, the event fires only when user interaction has stopped. Default = true.

  ```
  stepper.Continuous = true;
  ```

- Wraps: If true, then when the value reaches the MinimumValue, it proceeds to MaximumValue. Conversely, when it reaches MaximumValue, further increments change it to the MinimumValue. Default = false.

  ```
  stepper.Wraps = false;
  ```

UISlider

iOS

The UISlider is a sliding input button for selection across a range of values. Add a UISlider to the view by using your designer of choice.

Position the slider and stretch it to the desired width. Name it slider in the Properties view. When run, it should look like Figure 4-37.

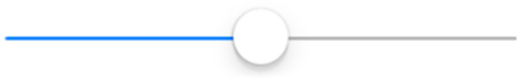


Figure 4-37. *Slider in its native habitat*

The properties can be set in the Properties view or in code:

```
slider.MinValue = -1;
slider.MaxValue = 2;
slider.Value = 0.5f;
```

MinValue and MaxValue properties determine the endpoint values of the slider. The Value property is the default value of the slider.

Add a UILabel named sliderLabel above the stepper to display the stepper value. When the slider is moved, the ValueChanged event fires, which can be handled like this:

```
slider.ValueChanged += (sender,e) => sliderLabel.Text = ((UISlider)sender).Value.
ToString ();
```

When run, the slider looks like Figure 4-38, which shows the slider set at `MinValue`.

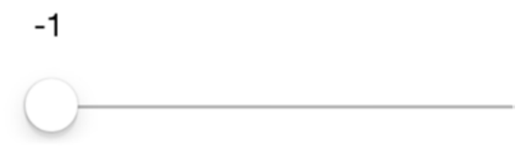


Figure 4-38. *Slider slid far left*

Figure 4-39 shows `MaxValue`.

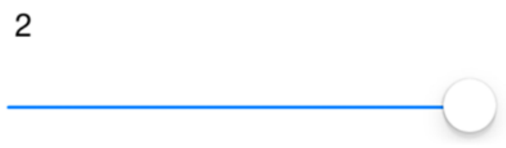


Figure 4-39. *Slider slid far right*

You can customize the look of the slider by using these properties:

- `MinimumTrackTintColor`: Color of the slider line to the left of the button.

  ```
  slider.MinimumTrackTintColor = UIColor.LightGray;
  ```

- `MaximumTrackTintColor`: Color of the slider line to the right of the button.

  ```
  slider.MaximumTrackTintColor = UIColor.Green;
  ```

- `ThumbTintColor`: Color of the thumb button. However, this option doesn't work reliably at the time of this writing because of an Apple bug. Use the following workaround before setting `ThumbTintColor` to make it work:

  ```
  slider.SetThumbImage(UIImage.FromBundle("thumb.png"),UIControlState.Normal);
  slider.ThumbTintColor = UIColor.Brown;
  ```

Place a small dummy image of some kind in the `Resources` folder corresponding to the image name.

CheckBox: Use UISwitch or MonoTouch.Dialog

iOS

iOS doesn't have an out-of-the-box check-box control, so many developers use `UISwitch`, explained next. You can also use `MonoTouch.Dialog`, a library that makes settings UIs easy to create for cross-platform apps. (See `https://github.com/migueldeicaza/MonoTouch.Dialog` for more details.)

UISwitch

iOS

`UISwitch` has become the standard for the Boolean on/off control. Add `UISwitch` to the view by using your designer of choice.

Position it and name it `thisSwitch` in the Properties view. When run, it should look like Figure 4-40.

Figure 4-40. *UISwitch turned on*

The `UISwitch.On` property contains the Boolean value of the switch, which can be set or gotten. Set it to `true` to create a default value or set it programmatically:

```
thisSwitch.On = true;
```

Get the state from the `On` property, like this:

```
bool state = thisSwitch.On;
```

Add a `UILabel` named `switchLabel` above the switch to display the switch value. When the switch is moved, the `ValueChanged` event fires, which can be handled like this:

```
thisSwitch.ValueChanged += (sender, e) => switchLabel.Text = thisSwitch.
On.ToString();
```

Flip the switch on (Figure 4-41) to see the label change to True.

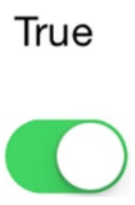


Figure 4-41. *It is true*

Flip the switch off (Figure 4-42) to see the label change to False.

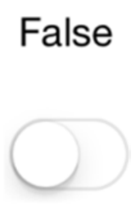


Figure 4-42. *Not true at all*

You can use these properties to customize the look of the switch:

- `ThumbTintColor`: Color of the thumb button

  ```
  thisSwitch.ThumbTintColor = UIColor.Blue;
  ```

- `TintColor`: Color around the edge of the switch

  ```
  thisSwitch.TintColor = UIColor.Blue;
  ```

- `OnTintColor` : Color of the switch around the button when button is on

  ```
  thisSwitch.OnTintColor = UIColor.Black;
  ```

- `OnImage`: Image used in place of the default thumb image when the switch is on

  ```
  switch.OnImage = onImage;
  ```

- `OffImage`: Image used in place of the default thumb image when the switch is off

  ```
  switch.OffImage = offImage;
  ```

■ **Note** The size of the thumb image used in `OnImage` and `OffImage` must be less than or equal to 77 points wide and 27 points tall. If you specify larger images, the edges may be clipped.

CODE COMPLETE: iOS Controls

iOS

Listing 4-8 shows the complete `UISlider`, `UIStepper`, and `UISwitch` example in the `ViewDidLoad` method.

Figure 4-43 shows all three selection controls.

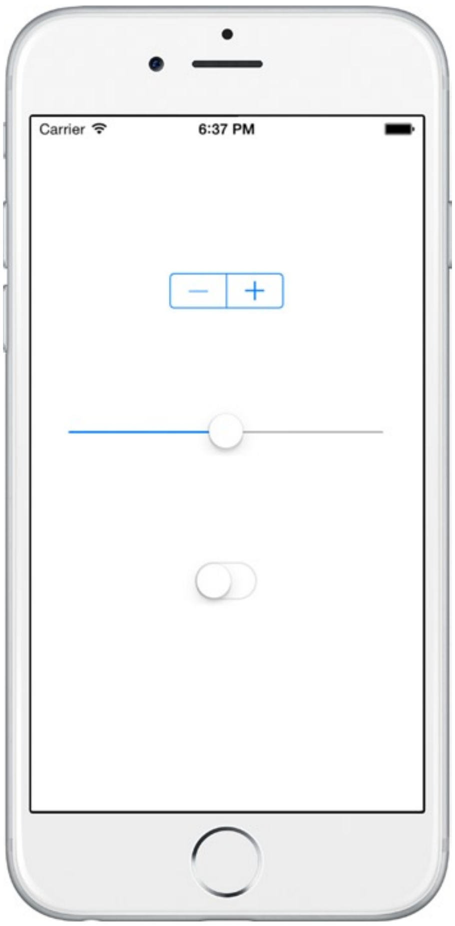


Figure 4-43. *UISlider, UIStepper, and UISwitch*

Listing 4-8. From SelectionViewController.cs in the StepperSliderSwitch solution

```
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();

    slider.MinValue = -1;
    slider.MaxValue = 2;
    slider.Value = 0.5f;
    slider.SetThumbImage(UIImage.FromBundle("thumb.png"),UIControlState.Normal);
    slider.ThumbTintColor = UIColor.Brown;
    slider.MinimumTrackTintColor = UIColor.LightGray;
    slider.MaximumTrackTintColor = UIColor.Green;
    slider.ValueChanged += (sender, e) => sliderLabel.Text = ((UISlider)sender).
    Value.ToString ();
    stepper.MinimumValue = 0;
    stepper.MaximumValue = 11;
    stepper.StepValue = 2;
    stepper.ValueChanged += (object sender, EventArgs e) => stepperLabel.Text =
    stepper.Value.ToString ();

    thisSwitch.On = false;
    thisSwitch.TintColor = UIColor.Blue;
    thisSwitch.OnTintColor = UIColor.Black;
    bool state = thisSwitch.On;
    thisSwitch.ValueChanged += (sender, e) => switchLabel.Text =
    thisSwitch.On.ToString();

}
```

Summary

Many controls share a common goal: allowing the user to pick a value. Simple selection controls require us to specify minimum and maximum values and set a default value. Pickers/spinners work best inside a modal dialog, and we use them to select from a list or choose dates and times. Xamarin.Forms handles the modal pop-ups for you, whereas Android and iOS require you to roll your own. Android provides some helpful out-of-the-box components such as `DatePickerDialog` and `TimePickerDialog`, but `DialogFragment` also makes a great modal. For modals in iOS, use the `UITextField.InputView` property (no more `ActionSheets`).

The selection controls in this chapter typically provide a *value changed* or click event of some kind to allow your code to respond to changes in values. While indispensable, the controls in this chapter are simple ones. In the next chapter, you'll dive deeper into the heart of mobile UI selection, where both the data and the selection can be richer and more complex when using lists and tables.