Touch

Evolve Advanced Track, Chapter 2

Overview

Touch screens on many of today's devices allow users to quickly and efficiently interact with devices in a natural and intuitive way. This interaction is not limited just to touch – it is possible to use gestures as well. The pinch-to-zoom is a very common of this – by pinching a part of the screen with two fingers the user can zoom in or zoom out on part of the screen.

This chapter will examine touch and gestures in both iOS and Android. Both operating systems can support multi-touch (many points of contact on the screen) and complex gestures. Both operating systems handle screen contact in the same way – with a class that is responsible for capture data about user contact with the screen, i.e. location, pressure, velocity, how many points of contact, etc. iOS relies heavily on events and the `UITouch` object and overrideable methods on the `UIViewController`, while Android uses a `MotionEvent` object and callback methods on the `View` object.

Both operating systems also provide API's on how to interpret these touch objects into gestures that may be used for application specific commands. iOS provides a rich collection of classes that can be used to handle many of the common types of gestures with very little extra code. Android, on the other hand, provides tools and API's that make adding complex custom gestures very easy.

It can be a bit confusing which API should be used to handle touches – should an application use the touch events or should it use the gesture recognizers? In general, preference should be given to gesture recognizers. Gesture recognizers are implemented as discrete classes. This provides for greater separation of concerns and better encapsulation. This in turn makes it easy to share the logic between different views.

This chapter will follow a similar format for each operating system. First, the touch API's will be introduced and explained as they are the foundation on which touch interactions are built. Then we will move on to learn about the iOS and Android API's for implementing gestures in an application – first by exploring some of the more common gestures and finishing up with how to create custom gestures.

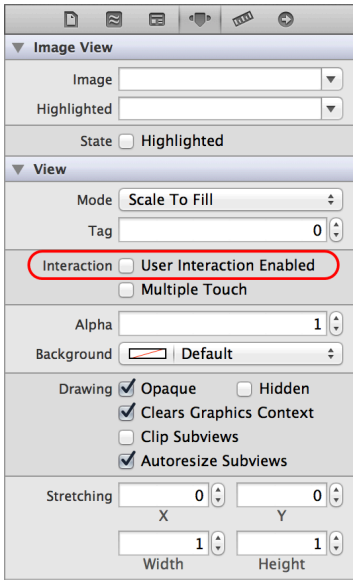
iOS Touch

It is import to understand the touch events and touch APIs in an iOS application as they are central to all physical interactions with the device. All touch interactions involve a `UITouch` object. In this section we will see how to use the `UITouch` class and it's API's to support touch, and move on from that to see how to support gestures.

Enabling Touch

Many of the controls in `UIKit` do not have touch enabled by default. There are two ways to enable touch on a control. The first way is to check the **User**

Interaction Enabled checkbox in interface, as shown in the following screenshot:



If the UI is created in code, then a controller should set the `UserInteractionEnabled` property to true on a `UIView` class. The following line of code is an example:

```
imgTouchMe.UserInteractionEnabled = true;
```

Touch Events

There are three phases of touch that occur when the user touches the screen, moves their finger, or removes their finger. iOS will call the associated methods on the `UIView` and the `UIViewController`:

- **TouchesBegin** – This is called when the screen is first touched.
- **TouchesMoved** – This is called when the location of the touch changes as the user is sliding their finger around the screen.
- **TouchesEnded** – This is called when the user's finger is lifted off the screen.
- **TouchesCancelled** – This is called when iOS cancels the touch.

Touch events bubble down through the stack. They will first be called on the topmost `UIView` or `UIViewController` and then will be called on the `UIView` and `UIViewController`s below them in the view hierarchy.

A `UITouch` object will be created each time the user touches the screen. The `UITouch` object includes data about the touch, such as when the touch occurred, where it occurred, if the touch was a swipe, etc.

Classes that override one of the touch events should first call the base implementation and then get the `UITouch` object associated with the event. To obtain a reference to the first touch, call the `AnyObject` property and cast it as a `UITouch` as show in the following example:

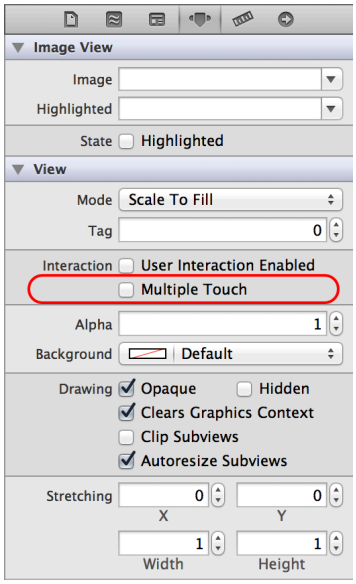
```
public override void TouchesBegan (NSSet touches, UIEvent evt)
{
    base.TouchesBegan (touches, evt);
    UITouch touch = touches.AnyObject as UITouch;
    if (touch != null)
    {
        //code here to handle touch
    }
}
```

iOS automatically recognizes successive quick touches on the screen and will collect them all as one tap in a single `UITouch` object. This makes checking for a double tap as easy as checking the `TapCount` property as shown in the following code snippet:

```
public override void TouchesBegan (NSSet touches, UIEvent evt)
{
    base.TouchesBegan (touches, evt);
    UITouch touch = touches.AnyObject as UITouch;
    if (touch != null)
    {
        if (touch.TapCount == 2)
        {
            // do something with the double touch.
        }
    }
}
```

Multi-Touch

Multi-touch is not enabled by default on controls. Multi-touch can be enabled in Interface Builder as shown in the following screenshot:



It is also possible to set multi-touch programmatically by setting the `MultipleTouchEnabled` property as shown in the following line of code:

```
imgTouchMe.MultipleTouchEnabled = true;
```

It is possible to find out how many fingers were used in the touch by `Count` property on the `UITouch` property

```
public override void TouchesBegan (NSSet touches, UIEvent evt)
{
    base.TouchesBegan (touches, evt);
    lblNumberOfFingers.Text = "Number of fingers: " +
touches.Count.ToString();
}
```

Determining Touch Location

The method `UITouch.LocationInView` will return a `PointF` object that holds the coordinates of the touch within a given view. Additionally you can test to see if that location is within a control by calling the method `Frame.Contains`. The following code snippet shows an example of this:

```
if (this.imgTouchMe.Frame.Contains (touch.LocationInView (this.View)))
{
    // the touch event happened inside the UIView imgTouchMe.
}
```

Now that we have an understanding of the touch events in iOS, lets learn about gesture recognizers.

Gesture Recognizers

Gesture recognizers can greatly simplify and reduce the programming effort to support touch in an application. iOS gesture recognizers will aggregate a series of touch events into a single touch event. Xamarin.iOS provides the class `UIGestureRecognizer` as a base class for the following built-in gesture recognizers:

- **UITapGesturesRecognizer** – This is for one or more taps.
- **UIPinchGestureRecognizer** – Pinching and spreading apart fingers.
- **UIPanGestureRecognizer** – Panning or dragging
- **UISwipeGestureRecognizer** – Swiping in any direction
- **UIRotationGestureRecognizer** – Rotating two fingers in a clockwise or counter-clockwise motion.
- **UILongPressGestureRecognizer** – Press and hold, sometimes referred to as a long-press or long-click.

The basic pattern to using a gesture recognizer is as follows:

1. **Instantiate the gesture recognizer** – This is as simple as instantiating a `UIGestureRecognizer` subclass. The object that is instantiated will be associated by a view and will garbage collected when the view is disposed of. It is not necessary to create this view as a class level variable.
2. **Configure any gesture settings** – Next it will be necessary to configure the gesture recognizer. Consult Xamarin's documentation on `UIGestureRecognizer` and it's subclasses for a list of properties that can be set to control the behaviour of a `UIGestureRecognizer` instance.

3. **Configure the target** – Because of its Objective-C heritage, Xamarin.iOS doesn't raise events when a gesture recognizer matches a gesture. `UIGestureRecognizer` has a method, `AddTarget` that can accept an anonymous delegate or an Objective-C selector with the code to execute when the gesture recognizer makes a match.
4. **Enable gesture recognizer** – Just like with touch events, gestures will only be recognized if touch interactions are enabled.
5. **Add the gesture recognizer to the view** – The final step is to add the gesture to a view by call `View.AddGestureRecognizer` and passing it a gesture recognizer object.

When the gesture target is called, it will be passed a reference to the gesture that occurred. This allows the gesture target to obtain information about the gesture that occurred. What information is available depends on the type of gesture recognizer that was used. Please see Xamarin's documentation for each `UIGestureRecognizer` subclass for information the data available to each subclass.

Once a gesture recognizer has been added to a view, the view (and any views below it) will not receive any touch events. In order to allow touch events simultaneously with gestures, the CancelsTouchesInView property must be set to false as shown in this code snippet:

```
_tapGesture.Recognizer.CancelsTouchesInView = false;
```

Each `UIGestureRecognizer` has a `State` property that provides important information about the status of the gesture recognizer. Every time the value of this property changes, iOS will call the subscribing method giving it an update. If the `State` property is never updated by a custom gesture recognizer, the subscriber is never called, making the gesture recognizer useless.

Gestures can be summarized as one of two types:

- **discrete** – These gestures only fire once – the first time it is recognized.
- **continuous** – These gestures continue to fire as long as they are recognized.

Gesture recognizers exists in one of the following states:

- **Possible** – This is the initial state of all gesture recognizers. This is the default value the State property.
- **Began** – When a continuous gesture is first recognized, the state is set to Began. This allows subscribes to differentiate between when gesture recognition starts and when it is changed.
- **Changed** – After a continuous gesture has begun, but hasn't finished, the state will be set to Changed every time a touch moves or changes, as long as it's still within the expected parameters of the gesture.
- **Cancelled** – This state will be set if the recognizer went from Began to Changed, and then the touches changed in such a way as to no longer fit the pattern of the gesture.

- Recognized – The state will be set when the gesture recognizer matches a set of touches and will inform the subscriber that the gesture has finished.
- Ended – This is an alias for the Recognized state.
- Failed – When the gesture recognizer can no longer match the touches it is listening for, the state will changed to Failed.

Xamarin.iOS represents these values in the `UIGestureRecognizerState` enumeration.

Working with Multiple Gestures

By default iOS does not allow default gestures to run simultaneously. Instead, each gesture recognizer will receive touch events in a non-deterministic order. The following code snippet shows how to make a gesture recognizer run simultaneously:

```
gesture.ShouldRecognizeSimultaneously += (UIGestureRecognizer r) =>
{ return true; };
```

It is also possible to disable a gesture in iOS. There are two delegate properties that allow a gesture recognizer to examine the state of an application and the current touch events in order to make decisions on how and if a gesture should be recognized. The two events are:

- ShouldReceiveTouch – This delegate is called right before the gesture recognizer is passed a touch event and provides an opportunity to examine the touches and decide which touches will be handled by the gesture recognizer.
- ShouldBegin – This is called when a recognizer attempts to change state from `Possible` to some other state. Returning `false` will force the state of the gesture recognizer to be changed to `Failed`.

You can override these methods with a strongly typed `UIGestureRecognizerDelegate`, a weak delegate, or bind via the event handler syntax as shown in the following code snippet:

```
gesture.ShouldReceiveTouch += (UIGestureRecognizer r, UITouch t) =>
{ return true; };
```

Finally, it is possible to queue up a gesture recognizer so that it will only succeed if another gesture recognizer fails. For example, a single tap gesture recognizer should only succeed when a double tap gesture recognizer fails. The following code snippet provides an example of this:

```
singleTapGesture.RequireGestureRecognizerToFail(doubleTapGesture);
```

Creating a Custom Gesture

Although iOS provides some default gesture recognizers, it may be necessary to create custom gesture recognizers in certain cases. Creating a custom gesture recognizer involves the following steps:

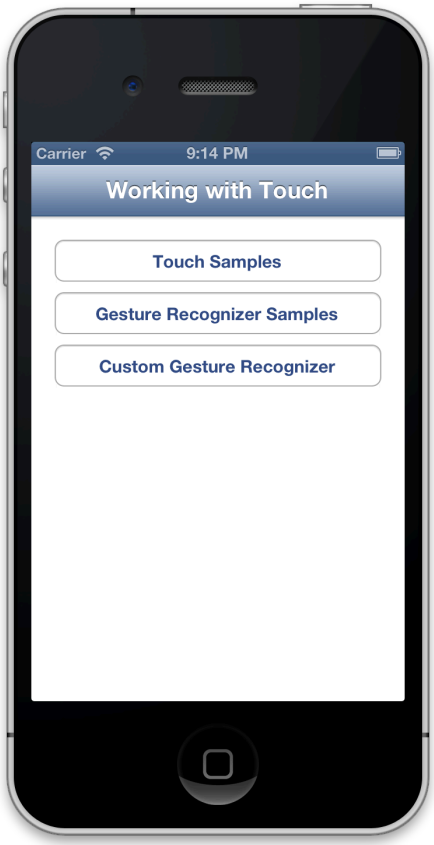
1. Subclass `UIGestureRecognizer`.
2. Override the appropriate touch event methods.

3. Bubble up recognition status via the base class' `State` property.

A practical example of this will be covered in the next section as part of the walkthrough.

iOS Touch Walkthrough

In this walkthrough we will create an iOS application with four separate screens, each screen demonstrating a different concept from this chapter. There will be one screen that acts as a switchboard and launches the other three. The following screenshot shows this main screen:

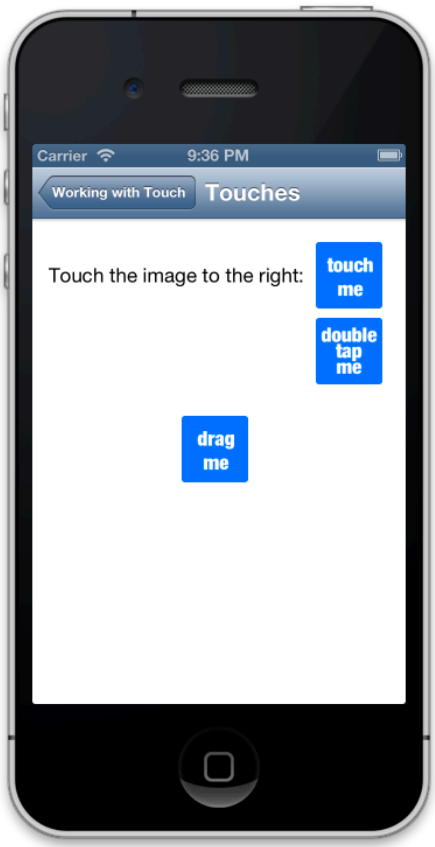


The first screen, Touch Samples, will show how to handle the touch events. The next screen, Gesture Recognizer Samples, will show how to use some of the built in gesture recognizers. The final screen, Custom Gesture Recognizer, will demonstrate how to create a custom gesture recognizer. We've broken this walkthrough up into separate sections, with each section focusing on one screen. Lets get started.

Touch Samples

In this sample, we will demonstrate some of the touch APIs.

1. Open the project Touch_Start. Before we get started, lets run the project to make sure everything is okay, and touch the Touch Samples button. You should see a screen similar to the following:



2. The first thing we need to do is to edit the file `Screens/iPhone/SimpleTouch/Touches_iPhone.xib.cs` and add the following two instance variables to the class `Touches_iPhone`:

```
protected bool imageHighlighted = false;
protected bool touchStartedInside;
```

3. Next, let's implement the TouchesBegan method as shown in the code below:

```
public override void TouchesBegan(NSSet touches, UIEvent evt)
{
        base.TouchesBegan(touches, evt);

        // we can get the number of fingers from the touch count, but
Multitouch must be enabled
        lblNumberOfFingers.Text = "Number of fingers: " +
touches.Count.ToString();

        // get the touch
        UITouch touch = touches.AnyObject as UITouch;
        if (touch != null)
        {
                Console.WriteLine("screen touched");
```

```
                //==== IMAGE TOUCH
                if (imgTouchMe.Frame.Contains(touch.LocationInView(View)))
                {
                        lblTouchStatus.Text = "TouchesBegan";
                }

                //==== IMAGE DOUBLE TAP
                if (touch.TapCount == 2 &&
        imgTapMe.Frame.Contains(touch.LocationInView(View)))
                {
                        if (imageHighlighted)
                        {
                                imgTapMe.Image =
                        UIImage.FromBundle("Images/DoubleTapMe.png");
                        }
                        else
                        {
                                imgTapMe.Image =
                        UIImage.FromBundle("Images/DoubleTapMe_Highlighted.p
                        ng");
                        }
                        imageHighlighted = !imageHighlighted;
                }

                //==== IMAGE DRAG
                // check to see if the touch started in the dragme image
                if (imgDragMe.Frame.Contains(touch.LocationInView(View)))
                {
                        touchStartedInside = true;
                }
        }
}
```

This code should be fairly straightforward. First we get our `UITouch` object and perfrom the null check. Then what we do depends on where the touch occurred:

- Inside `imgTouchMe` – display the text TouchesBegan in a label
- Inside `imgTapMe` – change the image displayed if the gesture was a double-tap.
- Inside `imgDragMe` – set a flag indicating that the touch has started. The method `TouchesMoved` will use this flag to determine if `imgDragMe` should be moved around the screen or not, as we shall see in the next step.

4. Now it's time to do something with the images when the user moves their finger on the screen. Implement `TouchesMoved` as shown in the code below:

```
public override void TouchesMoved(NSSet touches, UIEvent evt)
{
        base.TouchesMoved(touches, evt);
        // get the touch
        UITouch touch = touches.AnyObject as UITouch;
        if (touch != null)
```

```
        {
                //==== IMAGE TOUCH
                if (imgTouchMe.Frame.Contains(touch.LocationInView(View)))
                {
                        lblTouchStatus.Text = "TouchesMoved";
                }

                //==== IMAGE DRAG
                // check to see if the touch started in the dragme image
                if (touchStartedInside)
                {
                        // move the shape
                        float offsetX = touch.PreviousLocationInView(View).X
                - touch.LocationInView(View).X;
                        float offsetY = touch.PreviousLocationInView(View).Y
                - touch.LocationInView(View).Y;
                        imgDragMe.Frame = new RectangleF(new
                PointF(imgDragMe.Frame.X - offsetX, imgDragMe.Frame.Y -
                offsetY), imgDragMe.Frame.Size);
                }
        }
}
```

This method gets a `UITouch` object, and then checks to see where the touch occurred. If the touch occurred in `imgTouchMe`, then the text **TouchesMoved** is displayed on the screen.

If `touchStartedInside` is true, then we know that the user has their finger on `imgDragMe` and is moving it around. The code will move `imgDragMe` as the user moves their finger around the screen.

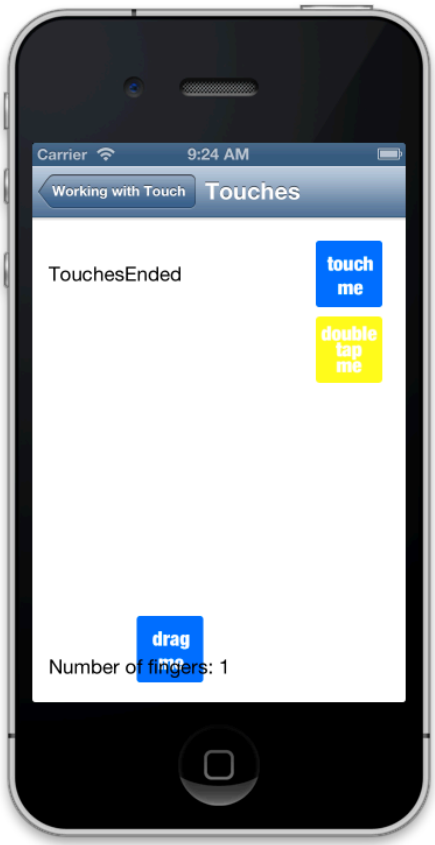
5. We need to hand the case when then user lifts their finger off the screen, or iOS cancels the touch event. For this, we will implement `TouchesEnded` and `TouchesCancelled` as shown below:

```
public override void TouchesCancelled(NSSet touches, UIEvent evt)
{
        base.TouchesCancelled(touches, evt);
        // reset our tracking flags
        touchStartedInside = false;
}

public override void TouchesEnded(NSSet touches, UIEvent evt)
{
        base.TouchesEnded(touches, evt);
        // get the touch
        UITouch touch = touches.AnyObject as UITouch;
        if (touch != null)
        {
                //==== IMAGE TOUCH
                if (imgTouchMe.Frame.Contains(touch.LocationInView(View)))
                {
                        lblTouchStatus.Text = "TouchesEnded";
                }
        }
        // reset our tracking flags
```

```
        touchStartedInside = false;
    }
```

Both of these methods will reset the `touchStartedInside` flag to `false`. `TouchesEnded` will also display **TouchesEnded** on the screen.

6. At this point the Touch Samples screen is finished. Notice how the screen changes with as you interact with each of the images. The following screenshot shows an example of this:



With this screen finished, lets move on to the next one – the **Gesture Recognizer Samples**.

Gesture Recognizer Samples

The previous section demonstrated how to drag an object around the screen by using touch events. In this next screen we will get rid of the touch events and show how to use the following gesture recognizers:

- the `UIPanGestureRecognizer` for dragging an image around the screen
- the `UITabpGestureRecognizer` which will respond to double taps on the screen

If you run the project, and click on the **Gesture Recognizer Samples** button, you should see the following screen:

Lets get started and implement the functionality for this screen.

1. Edit the file `Screens/iPhone/GestureRecognizers/GestureRecognizers_iPhone.xib.cs` and add the following instance variable:

```
private RectangleF originalImageFrame = RectangleF.Empty;
```

We need this instance variable to keep track of the previous location of the image. It will be used by the pan gesture recognizer to calculate the offset required to redraw the image on the screen.

2. Next, add the following method to the controller:

```
protected void WireUpDragGestureRecognizer()
{
        // create a new tap gesture
        UIPanGestureRecognizer gesture = new UIPanGestureRecognizer();
        // wire up the event handler (have to use a selector)
        gesture.AddTarget(() => { HandleDrag(gesture); });
        // add the gesture recognizer to the view
        imgDragMe.AddGestureRecognizer(gesture);
}
```

This code instantiates a `UIPanGestureRecognizer` instance, and adds it to a view. Notice that we assign a target to the gesture in the form of the method `HandleDrag`. Let's implement that method next.

3. To implement `HandleDrag`, add the following code to the controller:

```
protected void HandleDrag(UIPanGestureRecognizer recognizer)
{
      // if it's just began, cache the location of the image
      if (recognizer.State == UIGestureRecognizerState.Began)
      {
            originalImageFrame = imgDragMe.Frame;
      }

      if (recognizer.State != (UIGestureRecognizerState.Cancelled |
UIGestureRecognizerState.Failed | UIGestureRecognizerState.Possible))
      {
            // move the shape by adding the offset to the object's
       frame
            PointF offset = recognizer.TranslationInView(imgDragMe);
            RectangleF newFrame = originalImageFrame;
            newFrame.Offset(offset.X, offset.Y);
            imgDragMe.Frame = newFrame;
      }
}
```

The code above will first check the state of the gesture recognizer and then move the image around the screen. With this code in place, the controller can now support dragging the one image around the screen. Lets move on and add a gesture recognizer to display some text when the user double taps on the image.

4. Lets add a `UITabGestureRecognizer` that will change the image being displayed in `imgTapMe`. Add the following method to the `GestureRecognizer_iPhone` controller:

```
protected void WireUpTapGestureRecognizer()
{
      // create a new tap gesture
      UITapGestureRecognizer tapGesture = null;

      NSAction action = () => { lblGestureStatus.Text = "tap me image
tapped @" + tapGesture.LocationOfTouch(0, imgTapMe).ToString(); };

      tapGesture = new UITapGestureRecognizer(action);
      // configure it
      tapGesture.NumberOfTapsRequired = 2;
      // add the gesture recognizer to the view
      imgTapMe.AddGestureRecognizer(tapGesture);
}
```

This code is very similar to the code for the `UIPanGestureRecognizer` but instead of using a delegate for a target we are using an `NSAction`.

5. The final thing we need to do is modify `ViewDidLoad` so that it calls the methods we just added. Change `ViewDidLoad` so it resembles the following code:

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

        Title = "Gesture Recognizers";

        imgDragMe.Image = UIImage.FromBundle("Images/DragMe.png");
        imgTapMe.Image = UIImage.FromBundle("Images/DoubleTapMe.png");

        originalImageFrame = imgDragMe.Frame;

        WireUpTapGestureRecognizer();
        WireUpDragGestureRecognizer();
}
```

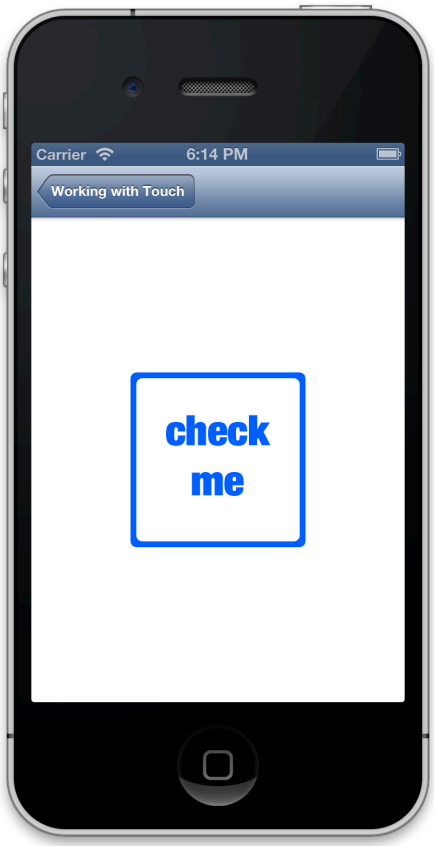
Notice as well that we initialize the value of `originalImageFrame` as well.

6. Run the application, and interact with the two images. The following screenshot is one example of these interactions:

At this point we've implemented two of the three screens in our application, and should have gained a good understanding of touches and gestures. Lets move on to the final screen and see how to implement a custom gesture.

Custom Gesture Recognizer

In this final screen we'll take everything we've learned so far, and make our own custom gesture recognizer. This will be a class that subclasses `UIGestureRecognizer`. It will recognize when the user draws a "V" on the screen, and toggle the bitmap displayed by the screen when it does so. The following screenshot is an example of this screen:



1. The first thing we need to do is to create our custom gesture recognizer. Add a new class to the project named `CheckmarkGestureRecognizer`, and update it with the following code:

```
public class CheckmarkGestureRecognizer : UIGestureRecognizer
{
        // declarations
        protected PointF midpoint = PointF.Empty;
        protected bool strokeUp = false;

        /// <summary>
        ///   Called when the touches end or the recognizer state fails
        /// </summary>
        public override void Reset()
        {
                base.Reset();

                strokeUp = false;
```

```
                midpoint = PointF.Empty;
        }

        /// <summary>
        ///   Is called when the fingers touch the screen.
        /// </summary>
        public override void TouchesBegan(NSSet touches, UIEvent evt)
        {
                base.TouchesBegan(touches, evt);

                // we want one and only one finger
                if (touches.Count != 1)
                {
                        base.State = UIGestureRecognizerState.Failed;
                }

                Console.WriteLine(base.State.ToString());
        }

        /// <summary>
        ///   Called when the touches are cancelled due to a phone call,
etc.
        /// </summary>
        public override void TouchesCancelled(NSSet touches, UIEvent evt)
        {
                base.TouchesCancelled(touches, evt);
                // we fail the recognizer so that there isn't unexpected
        behavior
                // if the application comes back into view
                base.State = UIGestureRecognizerState.Failed;
        }

        /// <summary>
        ///   Called when the fingers lift off the screen
        /// </summary>
        public override void TouchesEnded(NSSet touches, UIEvent evt)
        {
                base.TouchesEnded(touches, evt);

                if (base.State == UIGestureRecognizerState.Possible &&
        strokeUp)
                {
                        base.State = UIGestureRecognizerState.Recognized;
                }

                Console.WriteLine(base.State.ToString());
        }

        /// <summary>
        ///   Called when the fingers move
        /// </summary>
        public override void TouchesMoved(NSSet touches, UIEvent evt)
        {
                base.TouchesMoved(touches, evt);
```

```
                // if we haven't already failed
                if (base.State != UIGestureRecognizerState.Failed)
                {
                        // get the current and previous touch point
                        PointF newPoint = (touches.AnyObject as
                UITouch).LocationInView(View);
                        PointF previousPoint = (touches.AnyObject as
                UITouch).PreviousLocationInView(View);

                        // if we're not already on the upstroke
                        if (!strokeUp)
                        {
                                // if we're moving down, just continue to
                        set the midpoint at
                                // whatever point we're at. when we start to
                        stroke up, it'll stick
                                // as the last point before we upticked
                                if (newPoint.X >= previousPoint.X &&
                        newPoint.Y >= previousPoint.Y)
                                {
                                        midpoint = newPoint;
                                }
                                // if we're stroking up (moving right x and
                        up y [y axis is flipped])
                                else if (newPoint.X >= previousPoint.X &&
                        newPoint.Y <= previousPoint.Y)
                                {
                                        strokeUp = true;
                                }
                                // otherwise, we fail the recognizer
                        else
                        {
                                base.State =
                        UIGestureRecognizerState.Failed;
                        }
                }
                }
        Console.WriteLine(base.State.ToString());
        }
}
```

The code in this class shouldn't have any surprises in it, but it does have one overridden method, `Reset`, that we didn't discuss earlier in this chapter. The `Reset` method is called when the `State` property changes to either `Recognized` or `Ended`, the `Reset` method is called, giving you a chance to reset any internal state that you've set in your custom gesture recognizer. That way your class can start fresh when the user interacts with the application again and it can be ready to re-attempt at recognizing the gesture.

2. Now we need to use `CheckmarkGestureRecognizer`. Edit the file and add the following two instance variables:

```
protected bool isChecked = false;
private CheckmarkGestureRecognizer checkmarkGesture;
```

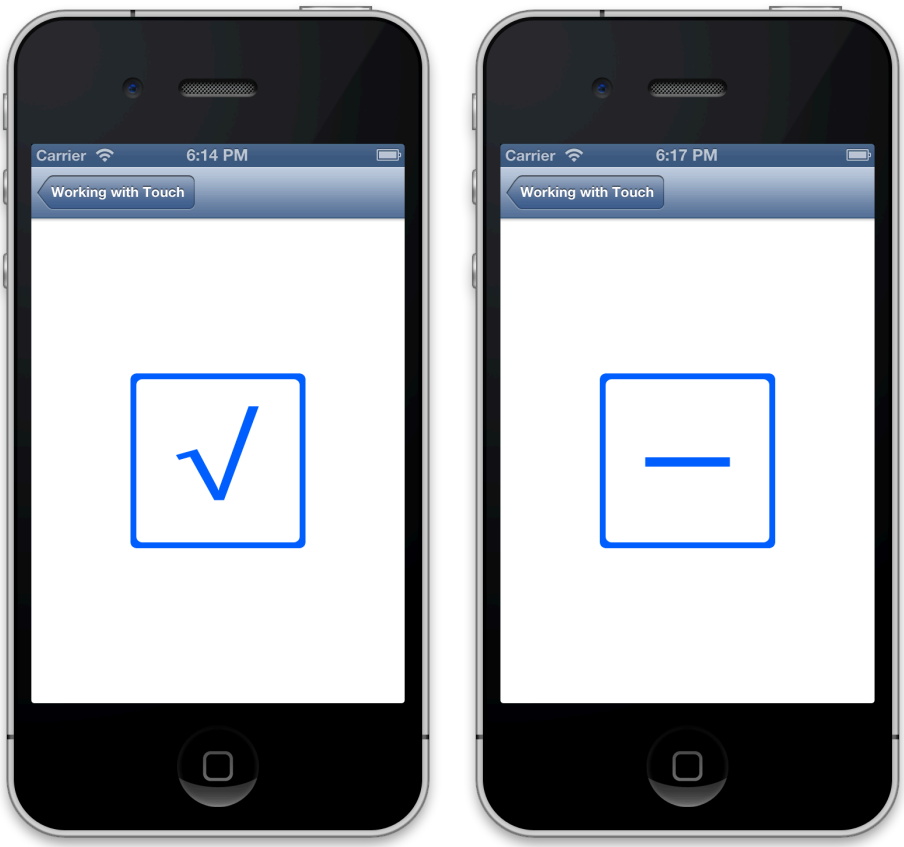
3. To instantiate and configure our gesture recognizer add the following method to the controller:

```
protected void WireUpCheckmarkGestureRecognizer()
{
        // create the recognizer
        checkmarkGesture = new CheckmarkGestureRecognizer();
        // wire up the event handler
        checkmarkGesture.AddTarget(() =>
        {
                if (checkmarkGesture.State ==
        (UIGestureRecognizerState.Recognized |
        UIGestureRecognizerState.Ended))
                {
                        if (isChecked)
                        {
                                imgCheckmark.Image =
                        UIImage.FromBundle("Images/CheckBox_Unchecked.png");
                        }
                        else
                        {
                                imgCheckmark.Image =
                        UIImage.FromBundle("Images/CheckBox_Checked.png");
                        }
                        isChecked = !isChecked;
                }
                });
        // add the gesture recognizer to the view
        View.AddGestureRecognizer(checkmarkGesture);
}
```

4. Finally, edit `ViewDidLoad` so that it calls `WireUpCheckmarkGestureRecognizer`, as shown in the following code snippet:

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

        imgCheckmark.Image =
UIImage.FromBundle("Images/CheckBox_Start.png");

        WireUpCheckmarkGestureRecognizer();
}
```

5. Run the application, and try drawing a “V” on the screen. You should see the image being displayed change, as shown in the following screenshots:



Congratulations! At this point you've completed the iOS touch portion of this chapter.

Android Touch

Much like iOS, Android creates an object that holds data about the user's physical interaction with the screen – an `Android.View.MotionEvent` object. This object holds data such as what action is performed, where the touch took place, how much pressure was applied, etc. A `MotionEvent` object breaks down the movement into to the following values:

- An action code that describes the type of motion, such as the initial touch, the touch moving across the screen, or the touch ending.
- A set of axis values that describe the position of the `MotionEvent` and other movement properties such as where the touch is taking place, when the touch took place, and how much pressure was used. The axis values may different depending on the device so the previous list does not describe all axis values.

The `MotionEvent` object will be passed to an appropriate method in an application. There are three ways for a Xamarin.Android application to respond a touch event:

- Assign an event handler to `View.Touch` – the `Android.Views.View` class has an `EventHandler<View.TouchEventArgs>` which applications can assign a handler to. This is typical .NET behaviour.
- Implementing `View.IOnTouchListener` – instances of this interface may be assigned to a view object using the `View.SetOnListener` method. This is functionally equivalent to assigning an event handler to the `View.Touch` event. If there is some common or shared logic that many different views may need when they are touched, it will be more efficient to create a class and implement this method than to assign each view their own event handler.
- Override `View.OnTouchEvent` – All views in Android subclass `Android.Views.View`. When a View is touched, Android will call the `OnTouchEvent` and pass it a `MotionEvent` object as a parameter.

Note: Not all Android devices support touch screens. It might be a good idea is to add the following tag to your manifest file so Google Play will only display your app to those devices that are touch enabled: `<uses-configuration android:reqTouchScreen="finger" />`.

Gestures

A *gesture* is a hand-drawn shape on the touch screen. A gesture can have one or multiple strokes to it, each stroke consisting of a sequence of points created by a different point of contact with the screen. Android can support many different types of gestures – from a simple fling across the screen to complex gestures that involve multi-touch.

Android provides the `Android.Gestures` namespace specifically for managing and responding to gestures. At the heart of all gestures is a special class name `Android.Gestures.GestureDetector`. As the name implies, this class will listen for gestures and events based on `MotionEvent`s supplied by the operating system.

To implement a gesture detector, an Activity must instantiate a `GestureDetector` class and provide an instance of `IOnGestureListener` as shown in the following code snippet:

```
GestureOverlayView.IOnGestureListener myListener = new
MyGestureListener();
_gestureDetector = new GestureDetector(this, myListener);
```

The other important thing that an Activity must do is to implement the `OnTouchEvent` and pass the `MotionEvent` on to the gesture detector. The following code snippet shows and example of this:

```
public override bool OnTouchEvent(MotionEvent e)
{
        // This method is in an Activity
        return _gestureDetector.OnTouchEvent(e);
}
```

When an instance of `GestureDetector` identifies a gesture of interest it will notify the activity or application either by raising an event or through a callback provided

by `GestureDetector.IOnGestureListener`. This interface provides six methods for the various gestures:

Method	Description
`OnDown`	Called when a tap occurs but is not released.
`OnFling`	Called when a fling occurs and provides data on the start and end touch that triggered the event.
`OnLongPress`	Called when a long press occurs.
`OnScroll`	Called when a scroll event occurs.
`OnShowPress`	Called after an `OnDown` has occurred and a move or up event has not been performed.
`OnSingleTapUp`	Called when a single tap occurs.

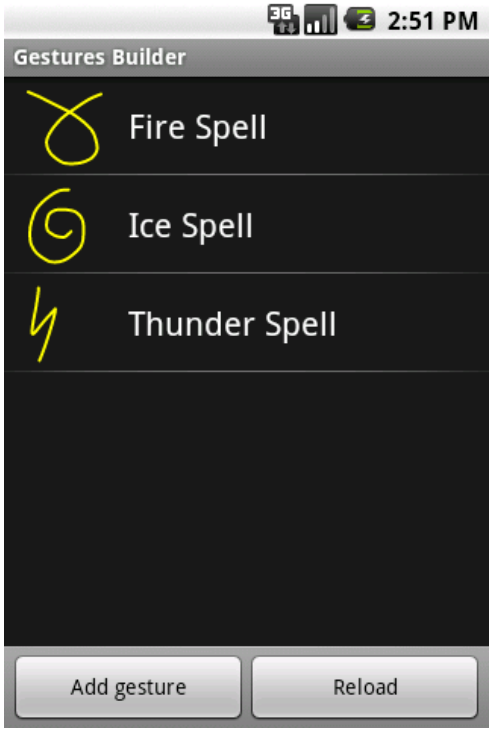
In many cases applications may only be interested in a subset of gestures. In this case, applications should extend the class `GestureDetector.SimpleOnGestureListener` and override the methods that correspond to the events that that they are interested in.

Custom Gestures

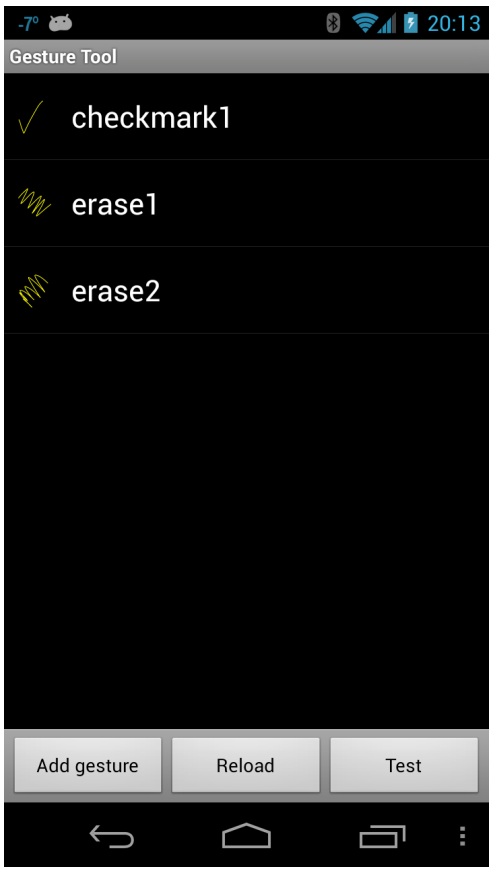
Gestures are a great way for users to interact with an application. The API's we have seen so far would suffice for simple gestures, but might prove a bit onerous for more complicated gestures. To help with more complicated gestures, Android provides another set of API's in the `Android.Gestures` namespace that will ease some of the burden associated with custom gestures.

Creating Custom Gestures

Since Android 1.6, the Android SDK comes with an application pre-installed on the emulator called *Gestures Builder*. This application allows a developer to create pre-defined gestures that can be embedded in an application. The following screen shot shows an example of Gestures Builder:



An improved version of this application called *Gesture Tool* can be found Google Play. Gesture Tool is very much like Gestures Builder except that it allows you to test gestures after they have been created. This next screenshot shows Gestures Builder:



Gesture Tool is a bit more useful for creating custom gestures as it allows the gestures to be tested as they are being created and is easily available through Google Play.

Gesture Tool allows you create a gesture by drawing on the screen and assigning a name. After the gestures are created they are saved in a binary file on the SD card of your device. This file needs to be retrieved from the device, and then packaged with an application in the folder `/Resources/raw`. This file can be retrieved from the emulator using the *Android Debug Bridge*. The following example shows copying the file from a Galaxy Nexus to the Resource directory of an application:

```
$ adb pull /storage/sdcard0/gestures <projectdirectory>/Resources/raw
```

Once you have retrieved the file it must be packaged with your application inside the directory `/Resources/raw`. The easiest way to use this gesture file is to load the file into a `GestureLibrary`, as shown in the following snippet:

```
GestureLibary myGestures = GestureLibraries.FromRawResources(this,
Resource.Raw.gestures);
if (!myGestures.Load())
{
        // The library didn't load, so close the activity.
        Finish();
}
```

Using Custom Gestures

To recognize custom gestures in an Activity, it must have an `Android.Gesture.GestureOverlay` object added to its layout. The following code snippet shows how to programmatically add a `GestureOverlayView` to an Activity:

```
GestureOverlayView gestureOverlayView = new GestureOverlayView(this);
gestureOverlayView.AddOnGesturePerformedListener(this);
SetContentView(gestureOverlayView);
```

The following XML snippet shows how to add a `GestureOverlayView` declaratively:

```
<android.gesture.GestureOverlayView
    android:id="@+id/gestures"
    android:layout_width="match_parent "
    android:layout_height="match_parent" />
```

The `GestureOverlayView` has several events that will be raised during the process of drawing a gesture. The most interesting event is `GesturePeformed`. This event is raised when the user has completed drawing their gesture.

When this event is raised, the Activity asks a `GestureLibrary` to try and match the gesture that the user with one of the gestures created by Gesture Tool. `GestureLibrary` will return a list of `Prediction` objects.

Each `Prediction` object holds a score and name of one of the gestures in the `GestureLibrary`. The higher the score, the more likely the gesture named in the `Prediction` matches the gesture drawn by the user. Generally speaking, those scores that are lower than 1.0 are considered to be poor matches.

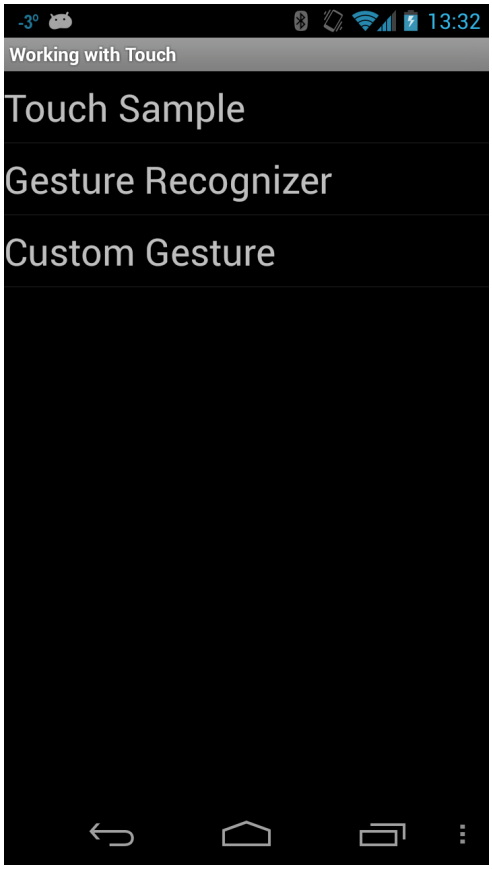
The following code shows an example of matching a gesture:

```
    private void GestureOverlayViewOnGesturePerformed(object sender,
    GestureOverlayView.GesturePerformedEventArgs gesturePerformedEventArgs)
    {
            // In this example _gestureLibrary was instantiated in OnCreate
            IEnumerable<Prediction> predictions = from p in
    _gestureLibrary.Recognize(gesturePerformedEventArgs.Gesture)
                    orderby p.Score descending
                    where p.Score > 1.0
                    select p;
            Prediction prediction = predictions.FirstOrDefault();

            if (prediction == null)
            {
                    Log.Debug(GetType().FullName, "Nothing matched the user's
            gesture.");
                    return;
            }

            Toast.MakeText(this, prediction.Name, ToastLength.Short).Show();
    }
```

With this done, you should have an understanding of how to use touch and gestures in a Xamarin.Android application. Let us now move on to a walkthrough and see all of the concepts in a working sample application.

Android Touch Walkthrough

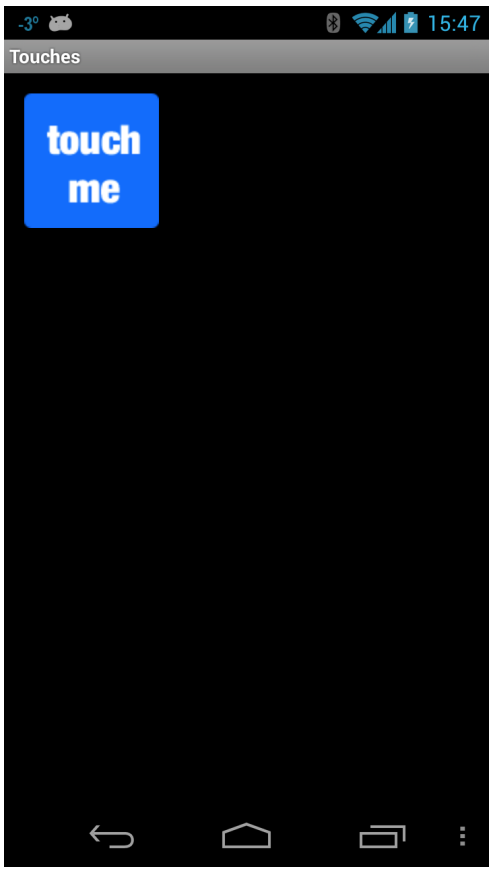
Let us see how to use the concepts from the previous section in a working application. We will create an application with four activities. The first activity will be a menu or a switchboard that will launch the other activities that will demonstrate the various APIs. The following screenshot shows the main activity:



The first Activity, **Touch Sample**, will show how use event handlers for touching the Views. The **Gesture Recognizer** activity will demonstrate how to subclass `Android.View.Views` and handle events as well as showing how to handle pinch gestures. The third and final activity, **Custom Gesture**, will show how use custom gestures. To make things easier to follow and absorb, we'll break this walkthrough up into sections, with each section focusing on one of the Activities. Lets get started.

Touch Sample Activity

1. Open the project **TouchWalkthrough_Start**. The `MainActivity` is all set to go – it is up to us to implement the touch behaviour in the activity. If you run the application and click on **Touch Sample**, the following activity should start up:



2. Now that we have confirmed that the Activity starts up, open up the file `TouchActivity.cs`, and add a handler for the Touch event of the ImageView as shown in the following screenshot:

```
_touchMeImageView.Touch += TouchMeImageViewOnTouch;
```

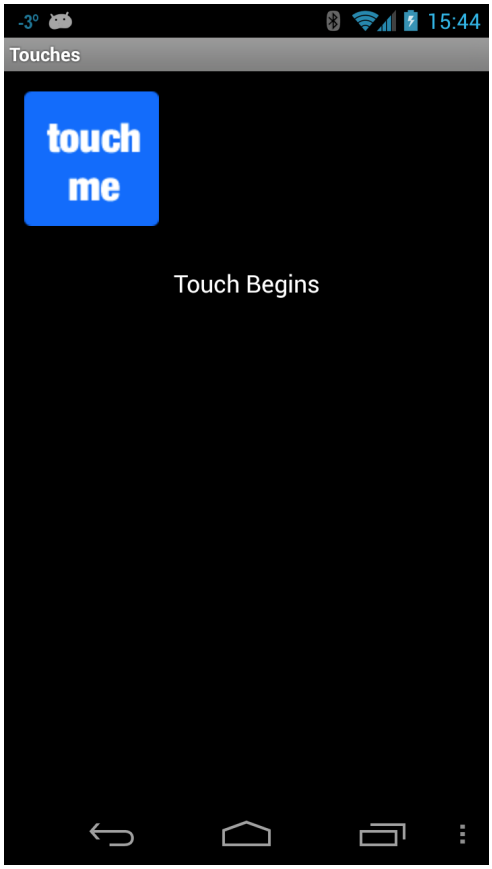
3. Next, add the following method to `TouchActivity.cs`:

```
private void TouchMeImageViewOnTouch(object sender, View.TouchEventArgs
touchEventArgs)
{
        string message;
        switch (touchEventArgs.Event.Action & MotionEventArgs.Mask)
        {
                case MotionEventActions.Down:
                case MotionEventActions.Move:
                        message = "Touch Begins";
                        break;

                case MotionEventActions.Up:
                        message = "Touch Ends";
                        break;

                default:
                        message = string.Empty;
                        break;
        }
```

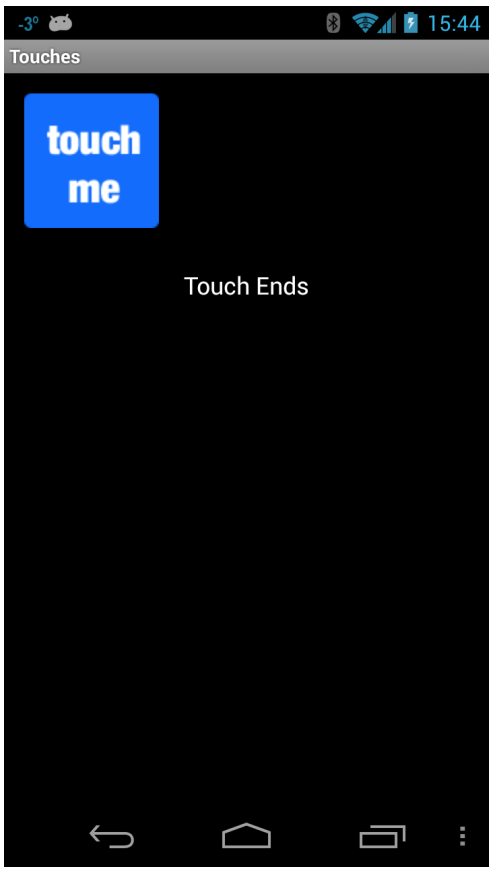
```
            _touchInfoTextView.Text = message;
        }
```

Notice in the code above that we treat the `Move` and `Down` action as the same. This is because even though the user may not lift their finger off the ImageView, it may move around or the pressure exerted by the user may change. These types of changes will generate a `Move` action.

Each time the user touches the ImageView, the Touch event will be raised and our handler will display the message **Touch Begins** on the screen, as shown in the following screenshot:



When the user is no longer touching the ImageView . As long as the user is touching the ImageView, the message Touch Ends will be displayed in a TextView, as shown in the following screenshot:



Gesture Recognizer Activity

Now lets go on an implement the Gesture Recognizer activity. This activity will demonstrate how to drag a view around the screen and one way to implement pinch-to-zoom.

1. Add a new Activity to the application called `GestureRecognizer`. Edit the code for this activity to resemble the following code:

```
public class GestureRecognizerActivity : Activity
{
        protected override void OnCreate(Bundle bundle)
        {
                base.OnCreate(bundle);
                View v = new GestureRecognizerView(this);
                SetContentView(v);
        }
}
```

2. Add a new Android view to the project, and name it `GestureRecognizerView`. Add the following variables to this class:

```
private static readonly int InvalidPointerId = -1;

private readonly Drawable _icon;
private readonly ScaleGestureDetector _scaleDetector;
```

```
private int _activePointerId = InvalidPointerId;
private float _lastTouchX;
private float _lastTouchY;
private float _posX;
private float _posY;
private float _scaleFactor = 1.0f;
```

3. Add the following constructor to `GestureRecognizerView`:

```
public GestureRecognizerView(Context context): base(context, null, 0)
{
        _icon = context.Resources.GetDrawable(Resource.Drawable.Icon);
        _icon.SetBounds(0, 0, _icon.IntrinsicWidth, _icon.IntrinsicHeight);
        _scaleDetector = new ScaleGestureDetector(context, new
MyScaleListener(this));
}
```

This constructor will add an ImageView to our activity. At this point the code still will not compile – we need to create the class `MyScaleListener` that will help with resizing the ImageView when the user pinches it.

4. In order to draw the image on our activity, we need to override the OnDraw method of the View class as shown in the following snippet:

```
protected override void OnDraw(Canvas canvas)
{
        base.OnDraw(canvas);
        canvas.Save();
        canvas.Translate(_posX, _posY);
        canvas.Scale(_scaleFactor, _scaleFactor);
        _icon.Draw(canvas);
        canvas.Restore();
}
```

This code will move the ImageView to the position specified by `_posX` and `_posY` as well as resize the image according to the scaling factor.

5. Next we need to update the instance variable `_scaleFactor` as the user pinches the ImageView. We will add a class called `MyScaleListener`. This class will listen for the scale events that will be raised by Android when the user pinches the ImageView. Add the following inner class to `GestureRecognizerView`:

```
private class MyScaleListener :
ScaleGestureDetector.SimpleOnScaleGestureListener
{
        private readonly GestureRecognizerView _view;

        public MyScaleListener(GestureRecognizerView view)
        {
                _view = view;
        }

        public override bool OnScale(ScaleGestureDetector detector)
        {
                _view._scaleFactor *= detector.ScaleFactor;
```

```
                // put a limit on how small or big the image can get.
                if (_view._scaleFactor > 5.0f)
                {
                        _view._scaleFactor = 5.0f;
                }
                if (_view._scaleFactor < 0.1f)
                {
                        _view._scaleFactor = 0.1f;
                }

                _view.Invalidate();
                return true;
        }
    }
```

This class is `ScaleGesture.SimpleOnScaleGestureListener`. `ScaleGesture.SimpleOnScaleGestureListener` is a convenience class that listeners can subclass when you are interested in a subset of gestures.

6. The next method we need to override in GestureRecognizerView is `OnTouchEvent`. The following code is this method fully implemented:

```
public override bool OnTouchEvent(MotionEvent ev)
{
        _scaleDetector.OnTouchEvent(ev);

        MotionEventActions action = ev.Action & MotionEventActions.Mask;
        int pointerIndex;

        switch (action)
        {
                case MotionEventActions.Down:
                        _lastTouchX = ev.GetX();
                        _lastTouchY = ev.GetY();
                        _activePointerId = ev.GetPointerId(0);
                        break;

                case MotionEventActions.Move:
                        pointerIndex =
                ev.FindPointerIndex(_activePointerId);
                        float x = ev.GetX(pointerIndex);
                        float y = ev.GetY(pointerIndex);
                        if (!_scaleDetector.IsInProgress)
                        {
                                // Only move the ScaleGestureDetector isn't
                        already processing a gesture.
                                float deltaX = x - _lastTouchX;
                                float deltaY = y - _lastTouchY;
                                _posX += deltaX;
                                _posY += deltaY;
                                Invalidate();
                        }

                        _lastTouchX = x;
                        _lastTouchY = y;
```

```
                        break;

                    case MotionEventActions.Up:
                    case MotionEventActions.Cancel:
                        // We no longer need to keep track of the active
                    pointer.
                        _activePointerId = InvalidPointerId;
                        break;

                    case MotionEventActions.PointerUp:
                        // check to make sure that the pointer that went up
                    is for the gesture we're tracking.
                        pointerIndex = (int) (ev.Action &
                    MotionEventActions.PointerIndexMask) >> (int)
                    MotionEventActions.PointerIndexShift;
                        int pointerId = ev.GetPointerId(pointerIndex);
                        if (pointerId == _activePointerId)
                        {
                            // This was our active pointer going up.
                        Choose a new
                            // action pointer and adjust accordingly
                            int newPointerIndex = pointerIndex == 0 ?
                        1 : 0;
                            _lastTouchX = ev.GetX(newPointerIndex);
                            _lastTouchY = ev.GetY(newPointerIndex);
                            _activePointerId =
                        ev.GetPointerId(newPointerIndex);
                        }
                        break;
                }
                return true;
            }
```

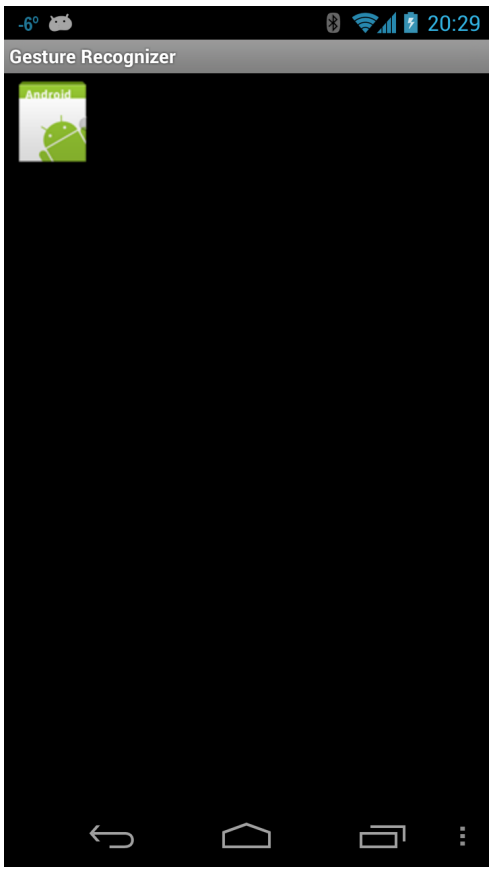
There is a lot of code here, so lets take a minute and look what is going on here.

The first thing this method does is scale the icon if necessary – this is handled by calling `_scaleDetector.OnTouchEvent`.

Next we try to figure out what action called this method:

- If the user touched the screen with, we record the X and Y positions and the ID of the first pointer that touched the screen.
- If the user moved their touch on the screen, then we figure out how far the user moved the pointer.
- If the user has lifted his finger off the screen, then we will stop tracking the gestures.

7. Now run the application, and start the Gesture Recognizer activity. When it starts the screen should look something like the screenshot below:



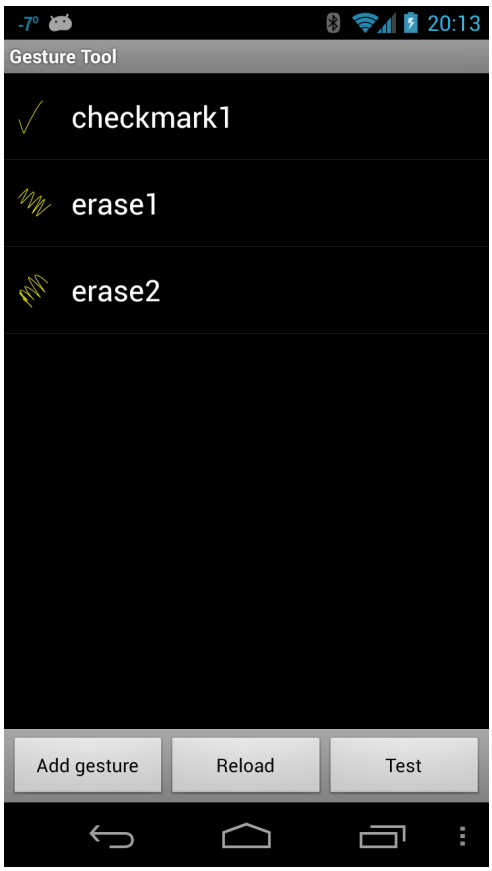
8. Now touch the icon, and drag it around the screen. Try the pinch-to-zoom gesture. At some point your screen may look something like the following screen shot:

At this point you should give yourself a pat on the back – you have just implemented pinch-to-zoom in an Android application. Take a quick break and lets move on to the third and final Activity in this walkthrough – using custom gestures.

Custom Gesture Activity

The final screen in this walkthrough will use custom gestures.

For the purposes of this Walkthrough, the gestures library has already been created using Gesture Tool and added to the project in the file `Resources/raw/gestures`. The following screenshot shows the gestures that exist in the library:



With this bit of housekeeping out of the way, lets get on with the final Activity in the walkthrough.

1. Add a layout file named `custom_gesture_layout.axml` to the project with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="0dp"
                android:layout_weight="1" />
        <ImageView
                android:src="@drawable/check_me"
                android:layout_width="match_parent"
                android:layout_height="0dp"
                android:layout_weight="3"
                android:id="@+id/imageView1"
                android:layout_gravity="center_vertical" />
        <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="0dp"
                android:layout_weight="1" />
</LinearLayout>
```

The project already has all the images in the Resources folder, so don't worry about having to add any of your own.

2. Next add a new Activity to the project and name it `CustomGestureRecognizerActivity.cs`. Add two instance variables to the class, as showing in the following two lines of code:

```
private GestureLibrary _gestureLibrary;
private ImageView _imageView;
```

3. Edit the OnCreate method of the this Activity so that it resembles the following code:

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

        GestureOverlayView gestureOverlayView = new
GestureOverlayView(this);
        SetContentView(gestureOverlayView);
        gestureOverlayView.GesturePerformed +=
GestureOverlayViewOnGesturePerformed;

        View view =
LayoutInflater.Inflate(Resource.Layout.custom_gesture_layout, null);
        _imageView = view.FindViewById<ImageView>(Resource.Id.imageView1);
        gestureOverlayView.AddView(view);

        _gestureLibrary = GestureLibraries.FromRawResource(this,
Resource.Raw.gestures);
        if (!_gestureLibrary.Load())
        {
                Log.Wtf(GetType().FullName, "There was a problem loading
        the gesture library.");
                Finish();
        }
}
```

Lets take a minute to explain what is going on in this code. The first thing we do is instantiate a `GestureOverlayView` and set that as the root view of the Activity. We also assign an event handler to the `GesturePerformed` event of `GestureOverlayView`.

Next we inflate the layout file that was created earlier, and add that as a child view of the `GestureOverlayView`.

The final step is to initialize the variable `_gestureLibrary` and load the gestures file from the application resources. If the gestures file cannot be loaded for some reason, there is not much this Activity can do so it is shutdown.

4. The final thing we need to do implement the method `GestureOverlayViewOnGesturePerformed` as shown in the following code snippet:

```
private void GestureOverlayViewOnGesturePerformed(object sender,
GestureOverlayView.GesturePerformedEventArgs gesturePerformedEventArgs)
{
```

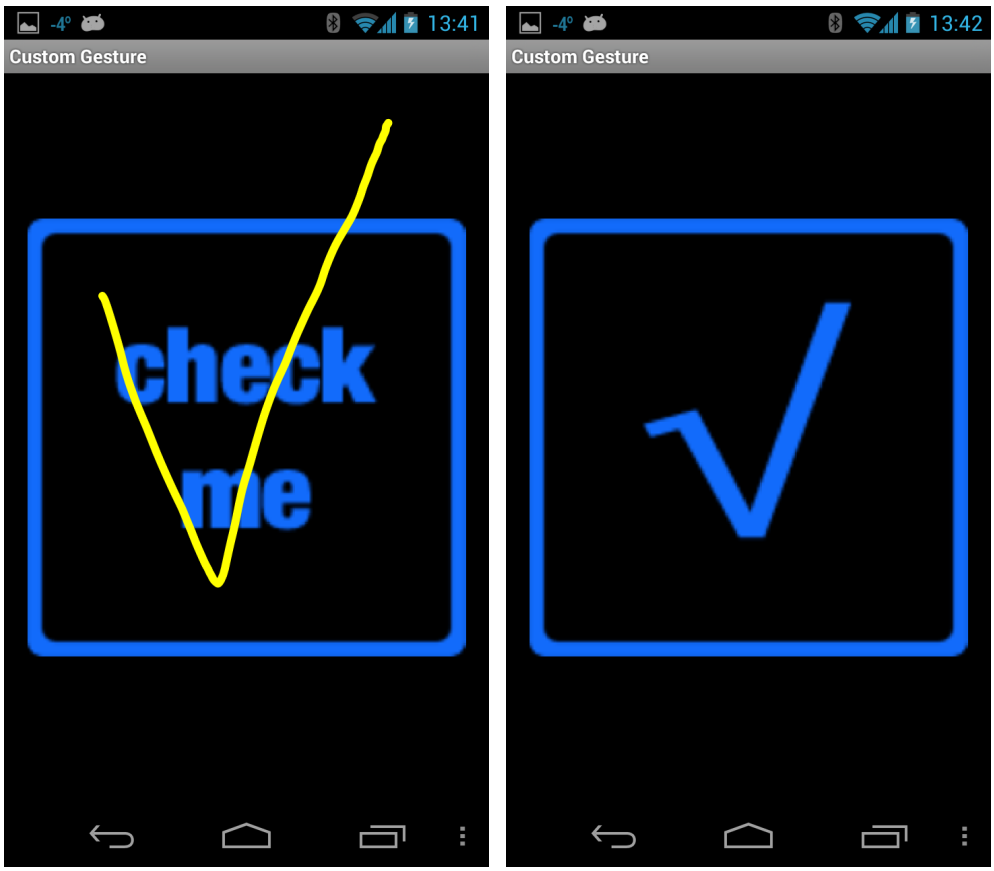
```
        IEnumerable<Prediction> predictions = from p in
_gestureLibrary.Recognize(gesturePerformedEventArgs.Gesture)
                orderby p.Score descending
                where p.Score > 1.0
                select p;
        Prediction prediction = predictions.FirstOrDefault();

        if (prediction == null)
        {
                Log.Debug(GetType().FullName, "Nothing seemed to match the
        user's gesture, so don't do anything.");
                return;
        }

        Log.Debug(GetType().FullName, "Using the prediction named {0} with
a score of {1}.", prediction.Name, prediction.Score);

        if (prediction.Name.StartsWith("checkmark"))
        {
                _imageView.SetImageResource(Resource.Drawable.checked_me);
        }
        else if (prediction.Name.StartsWith("erase",
StringComparison.OrdinalIgnoreCase))
        {
                // Match one of our "erase" gestures
                _imageView.SetImageResource(Resource.Drawable.check_me);
        }
}
```

When the `GestureOverlayView` detects a gesture, it calls back to this method. The first thing we try to get an `IList<Prediction>` objects that match the gesture by calling `_gestureLibrary.Recognize()`. We use a bit of LINQ to get the Prediction that has the highest score for the gesture.

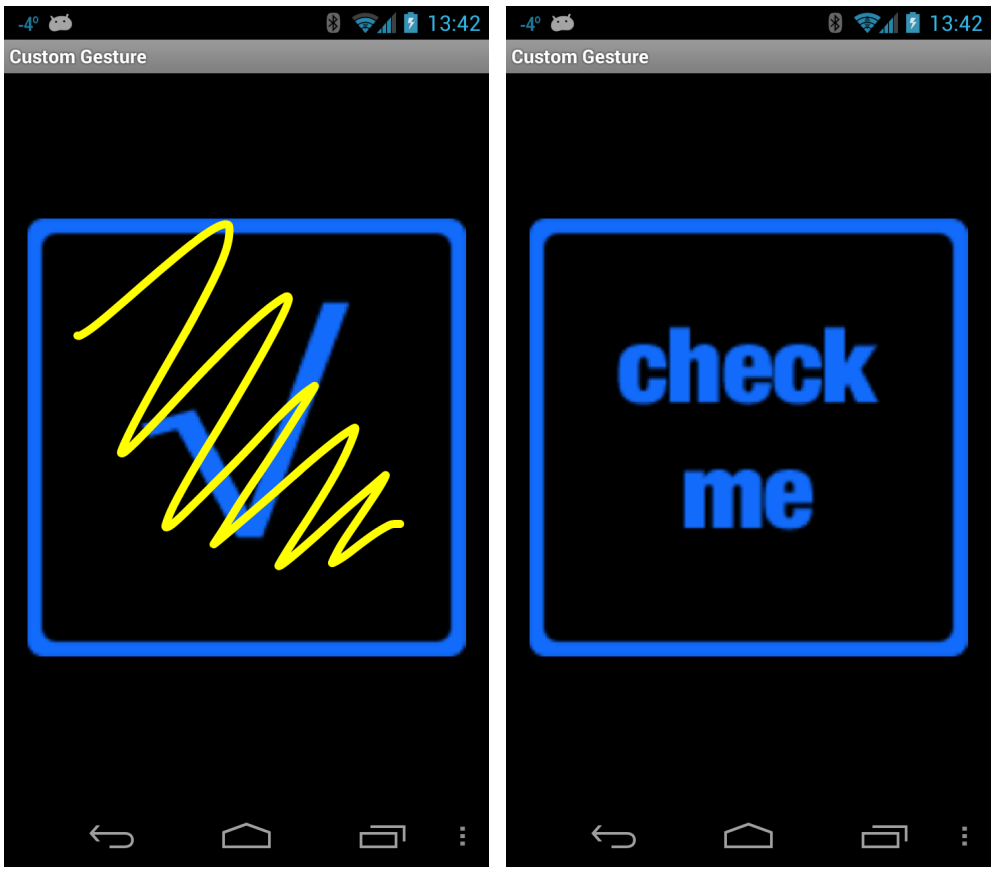
If there was no matching gesture with a high enough score, then the event handler exits without doing anything. Otherwise we check the name of the prediction and change the image being displayed based on the name of the gesture.

5. At this point, run the application and start up the **Custom Gesture Recognizer** activity. It should first look something like the following screenshot:

Now draw a checkmark on the screen, and the bitmap being displayed should look something the next screenshot:



Finally, draw a scribble on the screen, and the checkbox should change back to its original image as shown in this screenshot:



We've covered a lot of ground in this walkthrough, but now it's finally over and you now have an understanding of how to integrate touch and gestures in an Android application using Xamarin.Android. Give yourself a pat on the back, and move on to the next chapter.

Summary

In this chapter we examined touch in iOS and Android. For both operating systems, we learned how to enable touch and how to respond to the touch events. Then we learned about gestures and some of the gesture recognizers that both Android and iOS provide to handle some of the more common scenarios. After that we saw how to create custom gestures and implement them in applications. Finally, each section finished up with a walkthrough that demonstrated the concepts and APIs for each operating system in action.