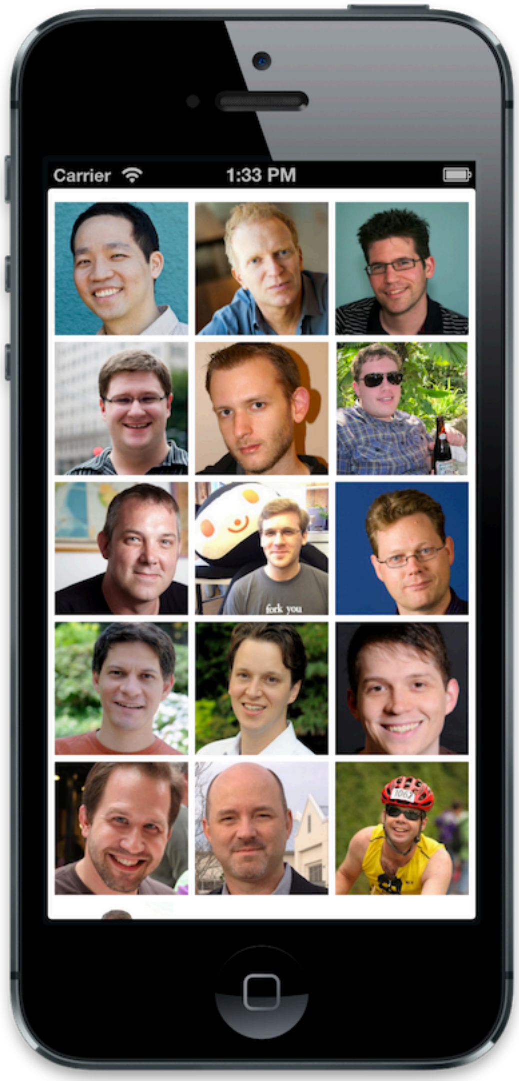
Collection Views in iOS

Evolve Advanced Track, Chapter 8

Xamarin Inc

Introduction

Collection Views, available in the `UICollectionView` class, are a new concept in iOS 6, and provide complex layout of multiple items. Out of the box, Collection Views allow developers new ways to display data out of the box, such as the grid layout shown below, from the EvolveCollectionViewDemo that accompanies this chapter.



The techniques for providing data to a `UICollectionView` to create items and for interacting with those items follow the same delegate and data source patterns commonly used in iOS development and are much like working with tables.

However, since Collection Views work with a layout subsystem that is independent of the `UICollectionView` itself, simply providing a different layout can easily change the presentation of a Collection View.

iOS provides a layout class called `UICollectionViewFlowLayout` that allows line-based layouts such as a grid to be created with no additional work. But custom layouts can be used to create a wide spectrum of presentation types.

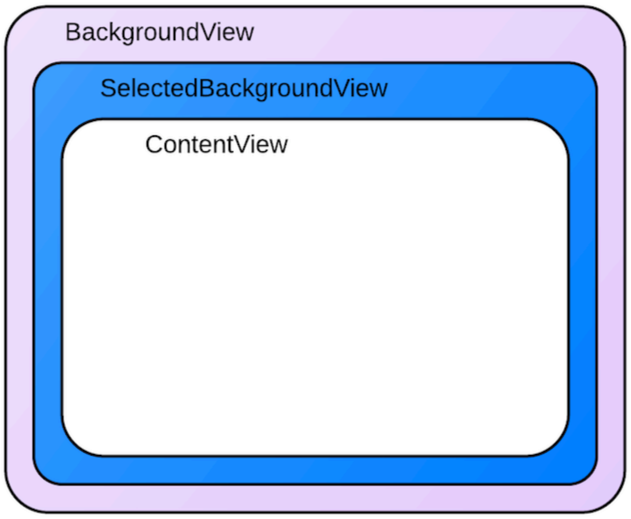
UICollectionView Basics

The `UICollectionView` class is made up of three different items:

- **Cells** – Data-driven Views that are used to render each item.
- **Supplementary Views** – Data-driven (meaning the views present content from some backing data collection) Views associated with a section.
- **Decoration Views** – Non-data-driven Views created by a layout.

Cells

Cells are objects that represent a single item in the data set that is being presented by the Collection View. Each cell is an instance of the `UICollectionViewCell` class, which is composed of three different Views, as shown in the figure below:

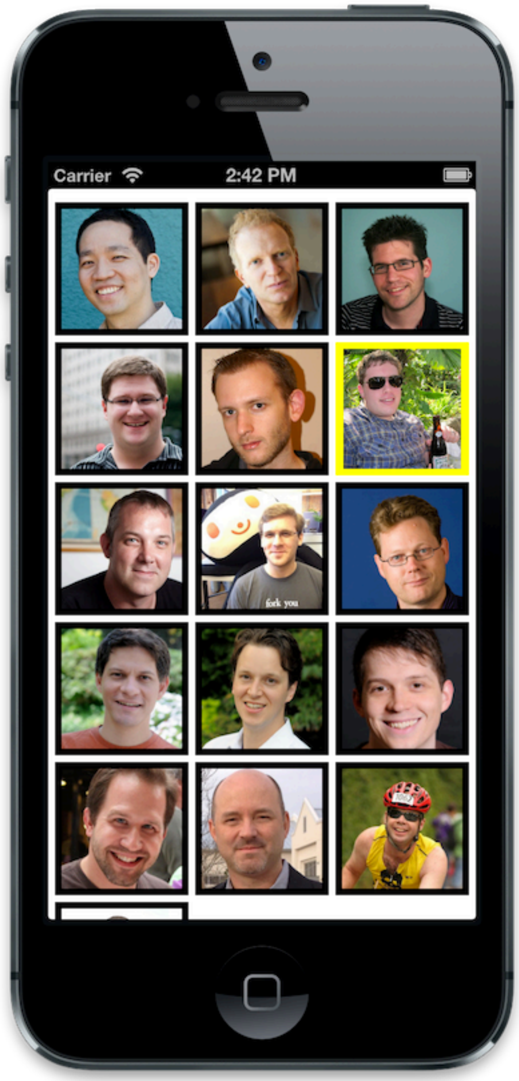


The `UICollectionViewCell` class has the following properties for each of these Views:

- `ContentView` – This View contains the content that the cell presents. It is rendered in the topmost z-order on the screen.
- `SelectedBackgroundView` – Cells have built-in support for selection. This View is used to visually denote that a cell is selected. It is rendered just below the `ContentView` when a cell is selected.

- `BackgroundView` – Cells can also display a background, which is presented by the `BackgroundView`. This View is rendered beneath the `SelectedBackgroundView`.

By setting the `ContentView` such that it is smaller than the `BackgroundView` and `SelectedBackgroundView`, the `BackgroundView` can be used to visually frame the content, while the `SelectedBackgroundView` will be displayed when a cell is selected, as shown below:



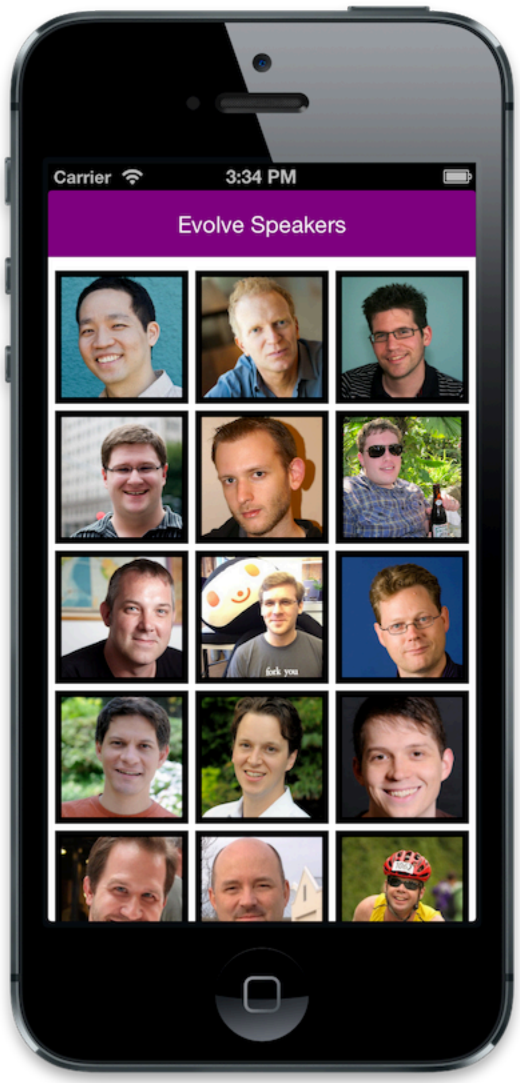
The cells in the screenshot above are created by inheriting from `UICollectionViewCell` and by setting the `ContentView`, `SelectedBackgroundView`, and `BackgroundView` properties, respectively, as shown in the following code:

```
class ImageCell : UICollectionViewCell
{
        UIImageView imageView;

        [Export ("initWithFrame:")]
        ImageCell (RectangleF frame) : base (frame)
        {
            // create an image view to use in the cell
            imageView = new UIImageView (
                new RectangleF (0, 0, 100, 100));
            imageView.ContentMode = UIViewContentMode.ScaleAspectFit;

            // populate the content view
            ContentView.AddSubview (imageView);

            // scale the content view down so that the background view
            // is visible, effecively as a border
            ContentView.Transform =
                CGAffineTransform.MakeScale (0.9f, 0.9f);

            // background view displays behind content view and
            // selected background view
            BackgroundView = new UIView{
                BackgroundColor = UIColor.Black};

            // selected background view displays over background view
            // when cell is selected
            SelectedBackgroundView = new UIView{
                BackgroundColor = UIColor.Yellow};
        }

        internal void UpdateImage (string path)
        {
            imageView.Image = UIImage.FromFile (path);
        }
    }
```

Supplementary Views

Supplementary Views are Views that present information associated with each section of a `UICollectionView`. Like cells, Supplementary Views are data-driven. Where cells present the item data from a data source, Supplementary Views present the section data, such as the categories of books in a bookshelf or the genre of music in a music library.

For example, a Supplementary View could be used to present a header for a particular section, as shown in the figure below:



However, Supplementary Views are more generic than just headers and footers. They can be positioned anywhere in the Collection View and can be composed of any views, making their appearance fully customizable.

Decoration Views

Decoration Views are purely visual Views that can be displayed in a `UICollectionView`. Unlike cells and Supplementary Views, they are not data-driven. They are created within the layout system that works with the `UICollectionView` and subsequently can change as the content's layout changes. For example, a Decoration View could be used to present a background View containing a map, as shown below:

Data Source

As with other parts of iOS, such as `UITableView` and `MKMapView`, `UICollectionView` gets its data from a data source, which is exposed in Xamarin.iOS via the `UICollectionViewDataSource` class. This class is responsible for providing the content to the `UICollectionView` including:

- Cells – Returned from the `GetCell` method.
- Supplementary Views – Returned from the `GetViewForSupplementaryElement` method.
- Number of sections – Returned from the `NumberOfSections` method. Defaults to one if not implemented.
- Number of items per section – Returned from the `GetItemsCount` method.

Also, as a convenience, the `UICollectionViewController` class is available, which is automatically configured to be both the delegate (which is discussed in the next section) and data source for its View, which is a `UICollectionView`.

As with `UITableView`, the `UICollectionView` class will only call its data source to get cells for items that are on the screen. Cells that scroll off the screen are placed into a queue for reuse.

Cell Creation and Reuse

In iOS 6 the cell reuse pattern has been simplified for both `UICollectionView` and `UITableView`. You no longer need to create a cell directly in the data source if a cell isn't available in the reuse queue. Instead, in iOS 6, cells are registered with the system. Then, when making the call to de-queue the cell from the reuse queue, (if a cell is not available) iOS 6 will create it automatically based upon the type or nib that was registered. The same technique is available for Supplementary Views as well.

For example, consider the following code that registers the `ImageCell` class shown earlier:

```
static readonly NSString cellId = new NSString ("ImageCell");
…
public override void ViewDidLoad ()
{
        base.ViewDidLoad ();
        CollectionView.RegisterClassForCell (typeof(ImageCell), cellId);
}
```

When a `UICollectionView` needs a cell because its item is on the screen, the `UICollectionView` calls its data source's `GetCell` method. Similar to how this works with `UITableView`, this method is responsible for configuring a cell from the backing data, which would be a `Speaker` class in this case.

The following code shows an implementation of `GetCell` that returns an `ImageCell` instance:

```
public override UICollectionViewCell GetCell (UICollectionView
collectionView, MonoTouch.Foundation.NSIndexPath indexPath)
{
        // get an ImageCell from the pool. DequeueReusableCell will create
        // one if necessary
        ImageCell imageCell = (ImageCell)collectionView.DequeueReusableCell
                (cellId, indexPath);

        // update the image for the speaker
        imageCell.UpdateImage (Speakers [indexPath.Row].ImageFile);

        return imageCell;
}
```

When the call is made to `DequeReusableCell`, the cell will be either de-queued from the reuse queue or—if a cell is not available in the queue—created based upon the type registered in the call to `CollectionView.RegisterClassForCell`.

In this case, by registering the `ImageCell` class, iOS will create a new `ImageCell` internally and return it when a call to de-queue a cell is made, after which it is configured with the image contained in the animal class and returned for display to the `UICollectionView`.

Delegate

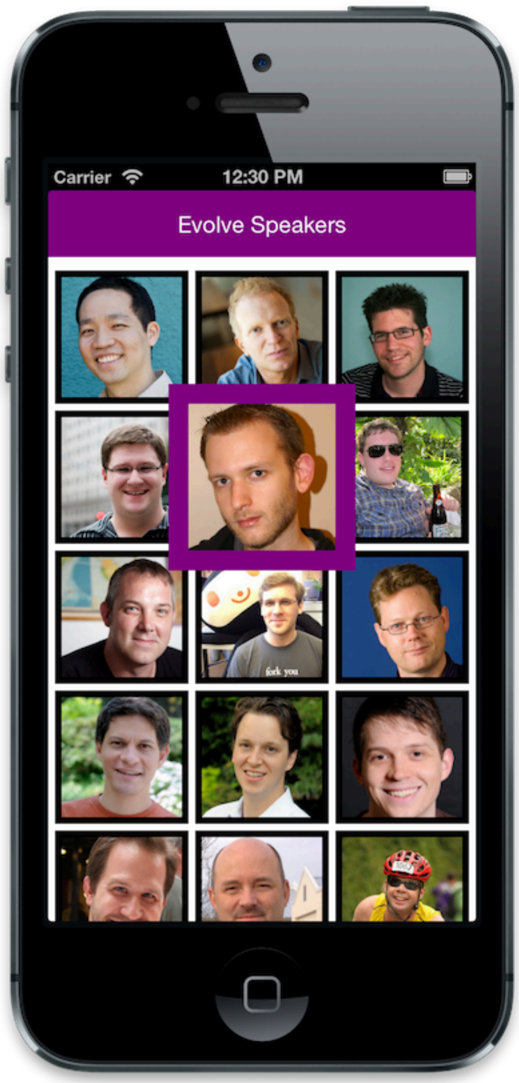
The `UICollectionView` class uses a delegate of type `UICollectionViewDelegate` to support interaction with content in the `UICollectionView`. This allows control of features such as:

- Cell Selection – Determines if a cell is selected.
- Cell Highlighting – Determines if a cell is currently being touched.

As with the data source, the `UICollectionViewController` is configured by default to be the delegate for the `UICollectionView`.

Cell Highlighting

When a cell is pressed, the cell transitions into a highlighted state, although it is not selected until the user lifts his or her finger from the cell. This allows a temporary change in the appearance of the cell before it is actually selected. The figure below shows the highlighted state just before the selection occurs, where in this case the cell is scaled to be a bit larger while it is highlighted:



To implement highlighting, the `ItemHighlighted` and `ItemUnhighlighted` methods of the `UICollectionViewDelegate` are used. For example, the following code will animate the cell to scale up when highlighted:

```
public override void ItemHighlighted (UICollectionView collectionView,
NSIndexPath indexPath)
{
        UICollectionViewCell cell = collectionView.CellForItem (indexPath);

        // animate the cell to scale up when highlighted
        UIView.Animate (
                duration: 0.2,
                animation: () => {
                        cell.ContentView.Transform =
                                CGAffineTransform.MakeScale (1.1f, 1.1f);
                        cell.BackgroundView.Transform =
                                CGAffineTransform.MakeScale (1.4f, 1.4f);
                        cell.BackgroundView.BackgroundColor =
                                UIColor.Purple;
                        cell.Layer.ZPosition = ++cellZIndex;
                }
        );
}

public override void ItemUnhighlighted (UICollectionView collectionView,
NSIndexPath indexPath)
{
        // restore the cell to its original scale when unhighlighted
        UICollectionViewCell cell = collectionView.CellForItem (indexPath);
        cell.BackgroundView.BackgroundColor = UIColor.Black;
        cell.ContentView.Transform = CGAffineTransform.MakeScale (0.9f,
                0.9f);
        cell.BackgroundView.Transform = CGAffineTransform.MakeScale (1.0f,
                1.0f);
}
```

Disabling Selection

Selection is enabled by default in `UICollectionView`. To disable selection, override `ShouldHighlightItem` and return `false` as shown below:

```
public override bool ShouldHighlightItem (UICollectionView collectionView,
NSIndexPath indexPath)
{
        return false;
}
```

When highlighting is disabled, the process of selecting a cell is disabled as well. Additionally, there is also a `ShouldSelectItem` method that controls the selection directly, although if `ShouldHighlightItem` is implemented and returns `false`, `ShouldSelectItem` is not called.

`ShouldSelectItem` allows selection to be turned on or off on an item-by-item basis, when `ShouldHighlightItem` is not implemented. It also allows highlighting without selection, if `ShouldHighlightItem` is implemented and returns `true`, while `ShouldSelectItem` returns `false`.

Layout

`UICollectionView` supports a layout system that allows the positioning of all its elements, cells, Supplementary Views, and Decoration Views, to be managed independent of the `UICollectionView` itself. Using the layout system, an application can support layouts such as the grid-like one we've seen in this article, as well as provide custom layouts.

Layout Basics

Layouts in a `UICollectionView` are defined in a class that inherits from `UICollectionViewLayout`. The layout implementation is responsible for creating the layout attributes for every item in the `UICollectionView`. There are two ways to create a layout:

- Use or subclass the built-in `UICollectionViewFlowLayout`

- Provide a custom layout by inheriting from `UICollectionViewLayout`.

Flow Layout

The `UICollectionViewFlowLayout` class provides a line-based layout that is suitable for arranging content in a grid of cells such as we've seen.

To use a flow layout:

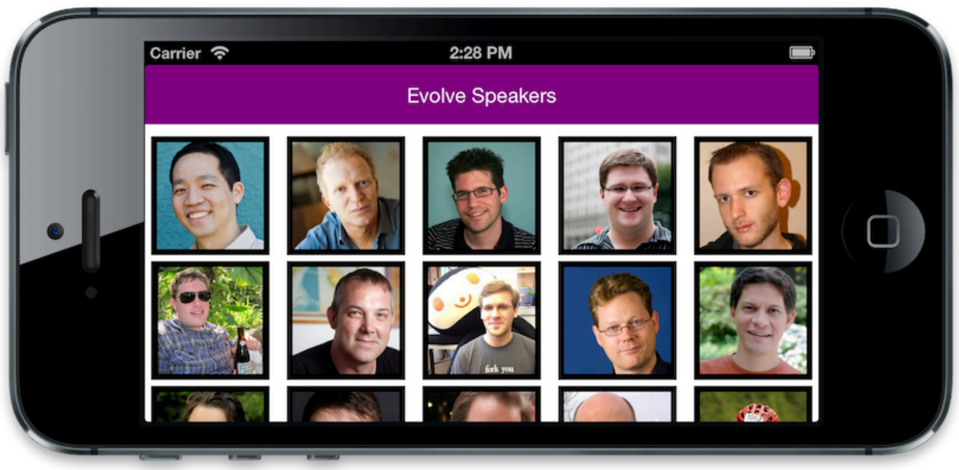
- Create and initialize an instance of `UICollectionViewFlowLayout`:

```
// create and initialize a UICollectionViewFlowLayout
layout = new UICollectionViewFlowLayout (){
        HeaderReferenceSize = new SizeF (
                UIScreen.MainScreen.Bounds.Width, 50),
        SectionInset = new UIEdgeInsets (10,5,10,5),
        MinimumInteritemSpacing = 5,
        MinimumLineSpacing = 5,
        ItemSize = new SizeF (100, 100)
};
```

- Pass the instance to the constructor of the `UICollectionView`:

```
// create an EvolveCollectionViewController (which is a
// UICollectionViewController) with a layout
viewController = new EvolveCollectionViewController (layout);
```

This is all that is needed to layout content in a grid. Also, when the orientation changes, the `UICollectionViewFlowLayout` handles rearranging the content appropriately, as shown:

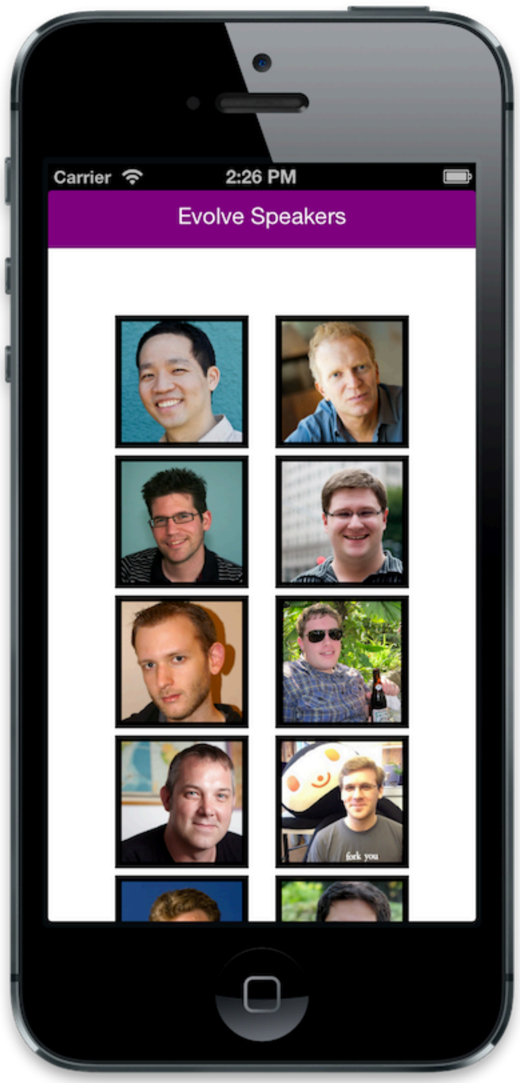


SectionInset

In order to provide some space around the `UIContentView`, layouts have a `SectionInset` property of type `UIEdgeInsets`. For example, the following code provides a 50-pixel buffer around each section of the `UIContentView` when laid out by a `UICollectionViewFlowLayout`:

```
SectionInset = new UIEdgeInsets (50,50,50,50);
```

This results in spacing around the section as shown below:



Also, notice the `SectionInset` has no affect on the position of the Supplementary View.

Custom Layout

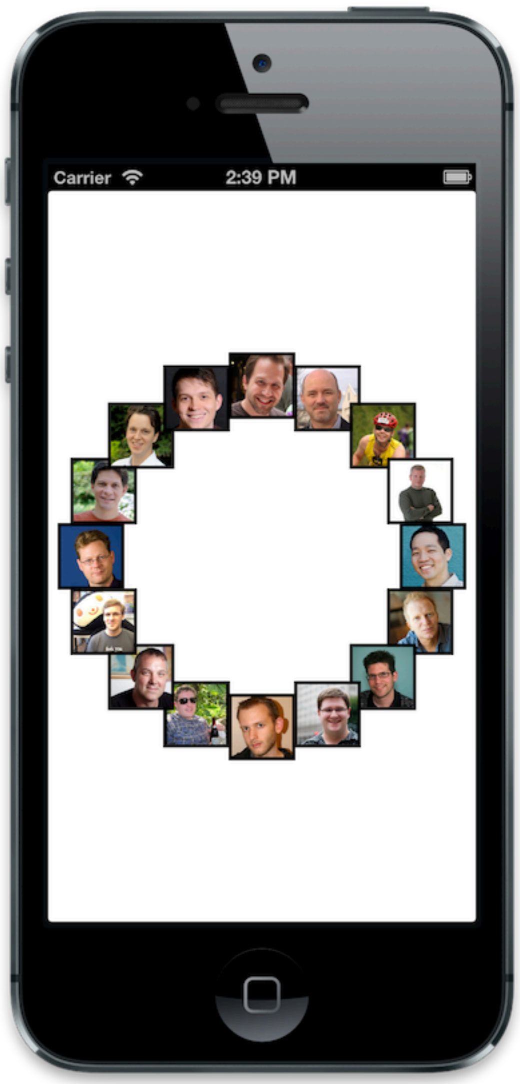
Layouts can also be fully customized by inheriting directly from `UICollectionViewLayout`.

The key methods to override are:

- `PrepareLayout` – Used for performing initial geometric calculations that will be used throughout the layout process.
- `CollectionViewContentSize` – Returns the size of the area used to display content.
- `LayoutAttributesForElementsInRect` – As with the `UICollectionViewFlowLayout` example shown earlier, this method is used to provide information to the `UICollectionView` regarding how to lay out

each item. However, unlike the `UICollectionViewFlowLayout`, when creating a custom layout, you can position items however you choose.

For example, the same content could be presented in a circular layout as shown below:



Changing the Layout at Runtime

To change from the grid-like layout to a horizontal scrolling layout—and subsequently to this circular layout—only the layout class provided by the `UICollectionView` must be changed.

For example, the following code adds `UISwipeGestureRecognizer` to the Collection View, which toggles between the flow layout and the custom layout when a swipe to the left occurs:

```
// toggle the layout in response to a swipe left
swipeLeft = new UITapGestureRecognizer (g => {

        if (customLayout == null) {
                // create and initialize a CustomLayout
                customLayout = new CustomLayout (
                        viewController.Speakers.Count){
                        ItemSize = new SizeF (100, 100)
                };
        }

        if (viewController.CollectionView.CollectionViewLayout is
                UICollectionViewFlowLayout) {

                // switch to a custom layout
                viewController.CollectionView.SetCollectionViewLayout (
                        customLayout, true);
        } else {

                // invalidate the flow layout in case the orientation
                // changed
                layout.InvalidateLayout();

                // switch to a flow layout
                viewController.CollectionView.SetCollectionViewLayout (
                        layout, true);

                // scroll to the top
                viewController.CollectionView.SetContentOffset(
                        new PointF(0,0), false);
        }
}){
        Direction = UISwipeGestureRecognizerDirection.Left
};
```

This is the powerful thing about layouts. Nothing at all changes in the `UICollectionView`, its delegate, or the data source code to achieve an animated transition to a different layout.

Summary

This chapter introduced Collection Views. It discussed the basic concepts of working with Collection Views and examined the primary classes that make up a Collection View. It then covered the mechanisms for providing data to a Collection View and handling item selection. Finally, it discussed the layout system used with Collection Views.