

Collection View Programming Guide for iOS

Contents

About iOS Collection Views 6

At a Glance 6

A Collection View Manages the Visual Presentation of Data-Driven Views 6

The Flow Layout Supports Grids and Other Line-Oriented Presentations 7

Gesture Recognizers Can Be Used for Cell and Layout Manipulations 7

Custom Layouts Let You Go Beyond Grids 7

Prerequisites 8

See Also 8

Collection View Basics 9

A Collection View Is a Collaboration of Objects 9

Reusable Views Improve Performance 12

The Layout Object Controls the Visual Presentation 13

Collection Views Initiate Animations Automatically 15

Designing Your Data Source and Delegate 16

The Data Source Manages Your Content 16

Designing Your Data Objects 18

Telling the Collection View About Your Content 19

Configuring Cells and Supplementary Views 20

Registering Your Cells and Supplementary Views 20

Dequeuing and Configuring Cells and Views 21

Inserting, Deleting, and Moving Sections and Items 23

Managing the Visual State for Selections and Highlights 24

Showing the Edit Menu for a Cell 27

Transitioning Between Layouts 28

Using the Flow Layout 30

Customizing the Flow Layout Attributes 31

Specifying the Size of Items in the Flow Layout 31

Specifying the Space Between Items and Lines 32

Using Section Insets to Tweak the Margins of Your Content 34

Knowing When to Subclass the Flow Layout 34

Incorporating Gesture Support 37

Using a Gesture Recognizer to Modify Layout Information 37

Working with Default Gesture Behaviors 39

Manipulating Cells and Views 39

Creating Custom Layouts 40

Subclassing UICollectionViewLayout 40

Understanding the Core Layout Process 41

Creating Layout Attributes 43

Preparing the Layout 44

Providing Layout Attributes for Items in a Given Rectangle 44

Providing Layout Attributes On Demand 46

Connecting Your Custom Layout for Use 47

Making Your Custom Layouts More Engaging 47

Elevating Content Through Supplementary Views 47

Including Decoration Views in Your Custom Layouts 48

Making Insertion and Deletion Animations More Interesting 49

Improving the Scrolling Experience of Your Layout 51

Tips for Implementing Your Custom Layouts 52

Custom Layouts: A Worked Example 54

The Concept 54

Initialization 56

Preparing the Layout 58

Creating the Layout Attributes 58

Storing the Layout Attributes 59

Providing the Content Size 62

Providing Layout Attributes 62

Providing Individual Attributes When Requested 63

Incorporating Supplementary Views 66

Recap 67

Document Revision History 69

Figures, Tables, and Listings

Collection View Basics 9

- Figure 1-1 Merging content and layout to create the final presentation 12
- Figure 1-2 The layout object provides layout metrics 14
- Table 1-1 The classes and protocols for implementing collection views 9

Designing Your Data Source and Delegate 16

- Figure 2-1 Sections arranged according to the arrangement of layout objects 17
- Figure 2-2 Arranging data objects using nested arrays 18
- Figure 2-3 Tracking touches in a cell 26
- Listing 2-1 Providing the section and item counts 19
- Listing 2-2 Configuring a custom cell 22
- Listing 2-3 Deleting the selected items 23
- Listing 2-4 Setting the background views to indicate changed states 24
- Listing 2-5 Applying a temporary highlight to a cell 25
- Listing 2-6 Selectively disabling actions in the Edit menu 27

Using the Flow Layout 30

- Figure 3-1 Laying out sections and cells using the flow layout 30
- Figure 3-2 Items of different sizes in the flow layout 31
- Figure 3-3 Actual spacing between items may be greater than the minimum 32
- Figure 3-4 Line spacing varies if items are of different sizes 33
- Figure 3-5 Section insets change the available space for laying out cells 34
- Table 3-1 Scenarios for subclassing UICollectionViewFlowLayout 35

Incorporating Gesture Support 37

- Listing 4-1 Using a gesture recognizer to change layout values 38
- Listing 4-2 Prioritizing your gesture recognizer 39

Creating Custom Layouts 40

- Figure 5-1 Laying out your custom content 42
- Figure 5-2 Laying out only the visible views 45
- Figure 5-3 Specifying the initial attributes for an item appearing onscreen 50
- Figure 5-4 Changing the proposed content offset to a more appropriate value 52
- Listing 5-1 Linking your custom layout 47

Listing 5-2 Specifying the initial attributes for an inserted cell 50

Custom Layouts: A Worked Example 54

Figure 6-1 Class hierarchy 55

Figure 6-2 Connecting parent and child index paths 58

Figure 6-3 The framing process 60

Figure 6-4 The layout so far 65

Listing 6-1 Connecting to the custom protocol 56

Listing 6-2 Initializing variables 56

Listing 6-3 Fulfilling requirements for subclassing layout attributes 57

Listing 6-4 Creating layout attributes 59

Listing 6-5 Storing layout attributes 60

Listing 6-6 Sizing the content area 62

Listing 6-7 Collecting and processing stored attributes 63

Listing 6-8 Providing attributes for specific items 64

Listing 6-9 Creating attributes objects for supplementary views 66

Listing 6-10 Providing supplementary view attributes on demand 67

About iOS Collection Views

A collection view is a way to present an ordered set of data items using a flexible and changeable layout. The most common use for collection views is to present items in a grid-like arrangement, but collection views in iOS are capable of more than just rows and columns. With collection views, the precise layout of visual elements is definable through subclassing and can be changed dynamically, so you can implement grids, stacks, circular layouts, dynamically changing layouts, or any type of arrangement you can imagine.

Collection views keep a strict separation between the data being presented and the visual elements used to present that data. In most cases, your app is solely responsible for managing the data. Your app also provides the view objects used to present that data. After that, the collection view takes your views and does all the work of positioning them onscreen. It does this work in conjunction with a layout object, which specifies the placement and visual attributes for your views and that can be subclassed to fit your app's exact needs. Thus, you provide the data, the layout object provides the placement information, and the collection view merges the two pieces together to achieve the final appearance.

At a Glance

The standard iOS collection view classes provide all of the behavior you need to implement simple grids. You can also extend the standard classes to support custom layouts and specific interactions with those layouts.

A Collection View Manages the Visual Presentation of Data-Driven Views

A collection view facilitates the presentation of data-driven views provided by your app. The collection view's only concern is taking your views and laying them out in a specific way. The collection view is all about the presentation and arrangement of your views and not about their content. Understanding the interactions between the collection view, its data source, the layout object, and your custom objects is crucial for using collection views in your app, particularly in smart and efficient ways.

Relevant chapters: [“Collection View Basics”](#) (page 9), [“Designing Your Data Source and Delegate”](#) (page 16)

The Flow Layout Supports Grids and Other Line-Oriented Presentations

A flow layout object is a concrete layout object provided by UIKit. You typically use the flow layout object to implement grids—that is, rows and columns of items—but the flow layout supports any type of linear flow. Because it is not just for grids, you can use the flow layout to create interesting and flexible arrangements of your content both with and without subclassing. The flow layout supports items of different sizes, variable spacing of items, custom headers and footers, and custom margins without subclassing. And subclassing allows you to tweak the behavior of the flow layout class even further.

Relevant chapter: [“Using the Flow Layout”](#) (page 30)

Gesture Recognizers Can Be Used for Cell and Layout Manipulations

Like all views, you can attach gesture recognizers to a collection view to manipulate the content of that view. Because a collection view involves the collaboration of multiple views, it helps to understand some basic techniques for incorporating gesture recognizers into your collection views. You can use gesture recognizers to tweak layout attributes or to manipulate items in a collection view.

Relevant chapter: [“Incorporating Gesture Support”](#) (page 37)

Custom Layouts Let You Go Beyond Grids

You can subclass the basic layout object to implement custom layouts for your app. Even though designing a custom layout does not typically require a large amount of code, the more you understand how layouts work, the better you can design your layout objects to be efficient. The guide’s final chapter focuses on an example project with a full implementation of a custom layout.

Relevant chapters: [“Creating Custom Layouts”](#) (page 40), [“Custom Layouts: A Worked Example”](#) (page 54)

Prerequisites

Before reading this document, you should have a solid understanding of the role views play in iOS apps. If you are new to iOS programming and not familiar with the iOS view architecture, read *View Programming Guide for iOS* before reading this book.

See Also

Collection views are somewhat related to table views, in that both present ordered data to the user. The implementation of a table view is similar to that of a standard collection view (one which uses the provided flow layout) in its use of index paths, cells, and view recycling. However, the visual presentation of table views is geared around a single-column layout, whereas collection views can support many different layouts. For more information about table views, see *Table View Programming Guide for iOS*.

Collection View Basics

To present its content onscreen, a collection view cooperates with many different objects. Some objects are custom and must be provided by your app. For example, your app must provide a data source object that tells the collection view how many items there are to display. Other objects are provided by UIKit and are part of the basic collection view design.

Like tables, collection views are data-oriented objects whose implementation involves a collaboration with your app's objects. Understanding what you have to do in your code requires a little background information about how a collection view does what it does.

A Collection View Is a Collaboration of Objects

The design of collection views separates the data being presented from the way that data is arranged and presented onscreen. Although your app is strictly responsible for managing the data to be presented, its visual presentation is managed by many different objects. Table 1-1 lists the collection view classes in UIKit and organizes them by the roles they play in implementing a collection view interface. Most of the classes are designed to be used as is without any need for subclassing, so you can usually implement a collection view with very little code. And when you want to go beyond the provided behavior, you can subclass and provide that behavior.

Table 1-1 The classes and protocols for implementing collection views

Purpose	Classes/Protocols	Description
Top-level containment and management	<code>UICollectionView</code> <code>UICollectionView-Controller</code>	<p>A <code>UICollectionView</code> object defines the visible area for your collection view's content. This class descends from <code>UIScrollView</code> and can contain a large scrollable area as needed. This class also facilitates the presentation of your data based on the layout information it receives from its layout object.</p> <p>A <code>UICollectionViewController</code> object provides view controller-level management support for a collection view. Its use is optional.</p>

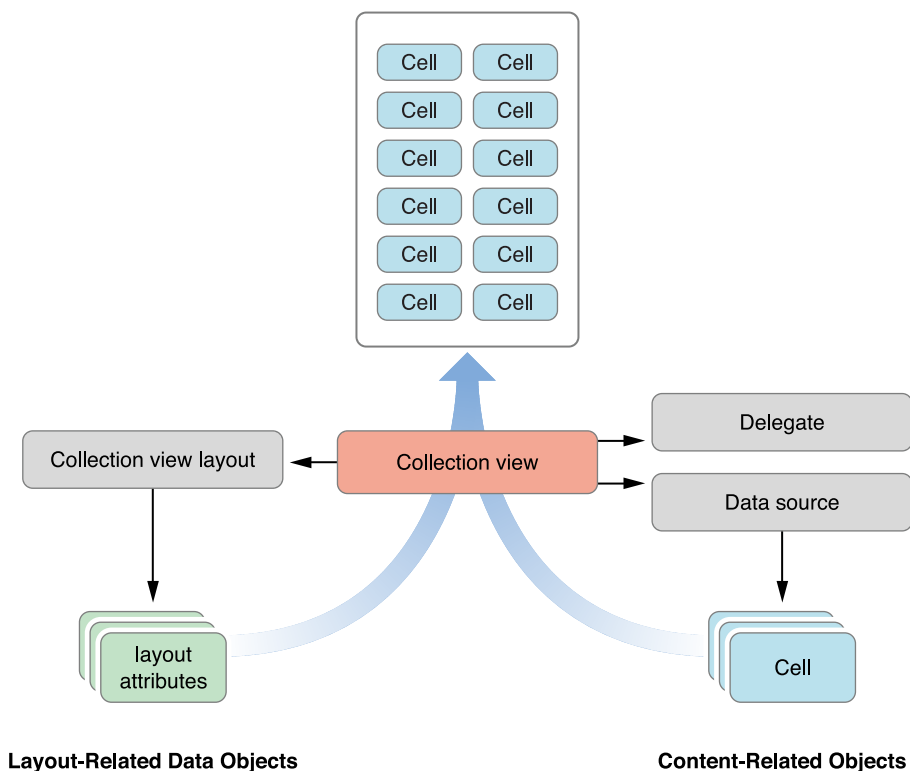
Purpose	Classes/Protocols	Description
Content management	UICollectionViewDataSource protocol UICollectionViewDelegate protocol	<p>The data source object is the most important object associated with the collection view and is one that you must provide. The data source manages the content of the collection view and creates the views needed to present that content. To implement a data source object, you must create an object that conforms to the <code>UICollectionViewDataSource</code> protocol.</p> <p>The collection view delegate object lets you intercept interesting messages from the collection view and customize the view's behavior. For example, you use a delegate object to track the selection and highlighting of items in the collection view. Unlike the data source object, the delegate object is optional.</p> <p>For information about how to implement the data source and delegate objects, see “Designing Your Data Source and Delegate” (page 16).</p>
Presentation	UICollectionViewReusable-View UICollectionViewCell	<p>All views displayed in a collection view must be instances of the <code>UICollectionViewReusable-View</code> class. This class supports a recycling mechanism in use by collection views. Recycling views (instead of creating new ones) improves performance in general and especially improves it during scrolling.</p> <p>A <code>UICollectionViewCell</code> object is a specific type of reusable view that you use for your main data items.</p>

Purpose	Classes/Protocols	Description
Layout	UICollectionViewLayout UICollectionViewLayoutAttributes UICollectionViewUpdateItem	<p>Subclasses of UICollectionViewLayout are referred to as layout objects and are responsible for defining the location, size, and visual attributes of the cells and reusable views inside a collection view.</p> <p>During the layout process, a layout object creates layout attribute objects (instances of the UICollectionViewLayoutAttributes class) that tell the collection view where and how to display cells and reusable views.</p> <p>The layout object receives instances of the UICollectionViewUpdateItem class whenever data items are inserted, deleted, or moved within the collection view. You never need to create instances of this class yourself.</p> <p>For more information about the layout object, see “The Layout Object Controls the Visual Presentation” (page 13).</p>
Flow layout	UICollectionViewFlowLayout UICollectionViewDelegateFlowLayout protocol	<p>The UICollectionViewFlowLayout class is a concrete layout object that you use to implement grids or other line-based layouts. You can use the class as-is or in conjunction with the flow delegate object, which allows you to customize the layout information dynamically.</p>

Figure 1-1 shows the relationship between the core objects associated with a collection view. The collection view gets information about the cells to display from its data source. The data source and delegate objects are custom objects provided by your app and used to manage the content, including the selection and highlighting

of cells. The layout object is responsible for deciding where those cells belong and for sending that information to the collection view in the form of one or more layout attribute objects. The collection view then merges the layout information with the actual cells (and other views) to create the final visual presentation.

Figure 1-1 Merging content and layout to create the final presentation



When creating a collection view interface, you first add a `UICollectionView` object to your storyboard or nib file. Think of the collection view as the central hub, from which all other objects emanate. After adding that object, you can begin to configure any related objects, such as the data source or delegate. All configurations are centered around the collection view itself. For example, you never create a layout object without also creating a collection view object.

Reusable Views Improve Performance

Collection views employ a view recycling program to improve efficiency. As views move offscreen, they are removed from view and placed in a reuse queue instead of being deleted. As new content is scrolled onscreen, views are removed from the queue and repurposed with new content. To facilitate this recycling and reuse, all views displayed by the collection view must descend from the `UICollectionViewReusableView` class.

Collection views support three distinct types of reusable views, each of which has a specific intended usage:

- **Cells** present the main content of your collection view. The job of a cell is to present the content for a single item from your data source object. Each cell must be an instance of the `UICollectionViewCell` class, which you may subclass as needed to present your content. Cell objects provide inherent support for managing their own selection and highlight state. To actually apply a highlight to a cell, you must write some custom code. For information on implementing cell highlighting/selecting, see [“Managing the Visual State for Selections and Highlights”](#) (page 24).
- **Supplementary views** display information about a section. Like cells, supplementary views are data driven. Unlike cells, supplementary views are not mandatory, and their usage and placement is controlled by the layout object being used. For example, the flow layout supports headers and footers as optional supplementary views.
- **Decoration views** are visual adornments that are wholly owned by the layout object and are not tied to any data in your data source object. For example, a layout object might use decoration views to implement a custom background appearance.

Unlike table views, collection views impose no specific style on the cells and supplementary views provided by your data source. Instead, the basic reusable view classes are blank canvases for you to modify. For example, you can use them to build small view hierarchies, to display images, or even to draw content dynamically.

Your data source object is responsible for providing the cells and supplementary views used by its associated collection view. However, the data source never creates views directly. When asked for a view, your data source dequeues a view of the desired type using the methods of the collection view. The dequeuing process always returns a valid view, either by retrieving one from a reuse queue or by using a class, nib file, or storyboard you provide to create a new view.

For information about how to create and configure views from your data source, see [“Configuring Cells and Supplementary Views”](#) (page 20).

The Layout Object Controls the Visual Presentation

The layout object is solely responsible for determining the placement and visual styling of items within the collection view. Although your data source object provides the views and the actual content, the layout object determines the size, location, and other appearance-related attributes of those views. This separation of responsibilities makes it possible to change layouts dynamically without changing any of the data objects managed by your app.

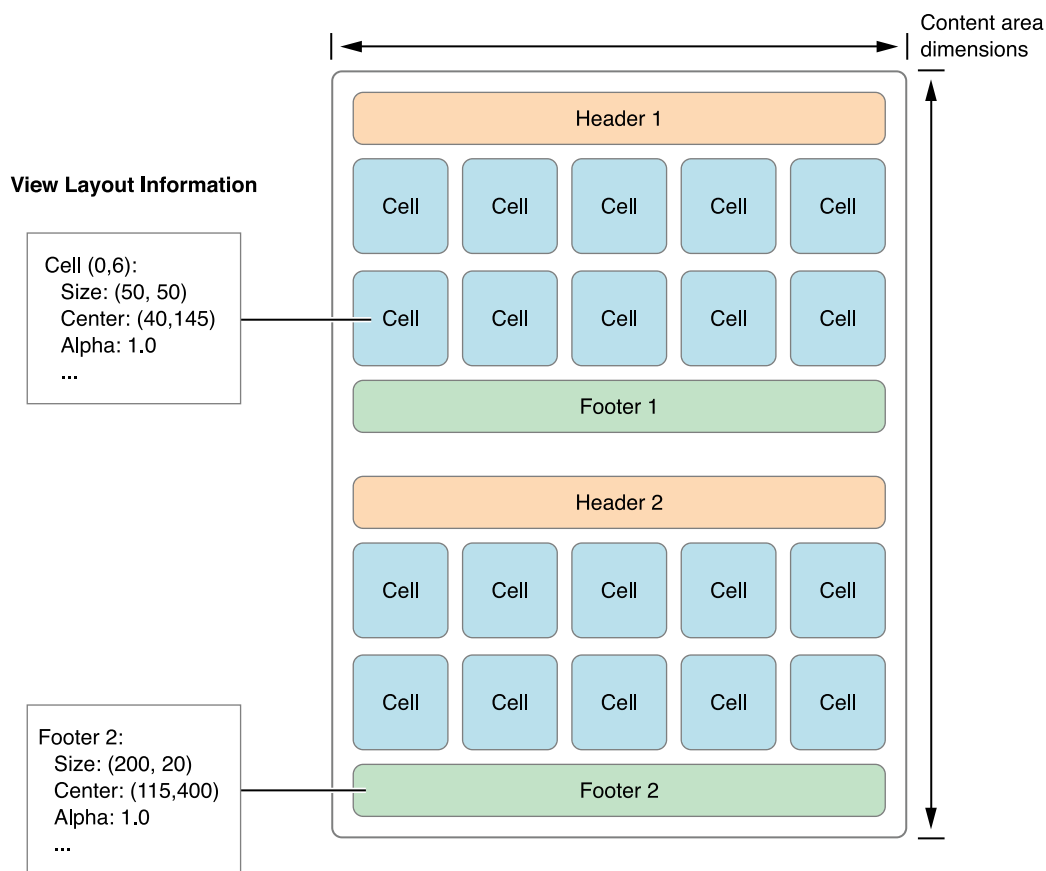
The layout process used by collection views is related to, but distinct from, the layout process used by the rest of your app’s views. In other words, do not confuse what a layout object does with the `layoutSubviews` method used to reposition child views inside a parent view. A layout object never touches the views it manages

directly because it does not actually own any of those views. Instead, it generates attributes that describe the location, size, and visual appearance of the cells, supplementary views, and decoration views in the collection view. It is then the job of the collection view to apply those attributes to the actual view objects.

There are no limits to how a layout object can affect the views in a collection view. A layout object can move some views but not others. It can move views only a little bit, or it can move them randomly around the screen. It can even reposition views without any regard for the surrounding views. For example, a layout object can stack views on top of each other if it wants. The only real limitation is how the layout object affects the visual style you want your app to have.

Figure 1-2 shows how a vertically scrolling flow layout arranges its cells and supplementary views. In a vertically scrolling flow layout, the width of the content area remains fixed and the height grows to accommodate the content. To compute the area, the layout object places views and cells one at a time, choosing the most appropriate location for each. In the case of the flow layout, the size of the cells and supplementary views are specified as properties, either on the layout object or by using a delegate. Computing the layout is just a matter of using those properties to place each view.

Figure 1-2 The layout object provides layout metrics



Layout objects control more than just the size and position of their views. The layout object can specify other view-related attributes, such as its transparency, its transform in 3D space, and its visibility (if any) above or below other views. These attributes let you create more interesting layouts. For example, you might create stacks of cells by placing the views on top of one another and changing their z-ordering, or you might use a transform to rotate them on any axis.

For detailed information about how a layout object fulfills its responsibilities to the collection view, see [“Creating Custom Layouts”](#) (page 40).

Collection Views Initiate Animations Automatically

Collection views build in support for animations at a fundamental level. When you insert (or delete) items or sections, the collection view automatically animates any views impacted by the change. For example, when you insert an item, items after the insertion point are usually shifted to make room for the new item. The collection view can create these animations because it detects the current position of items and can calculate their final positions after the insertion takes place. Thus, it can animate each item from its initial position to its final position.

In addition to animating insertions, deletions, and move operations, you can invalidate the layout at any time and force it to recalculate its layout attributes. Invalidating the layout does not animate items directly; when you invalidate the layout, the collection view displays the items in their newly calculated positions without animating them. Instead in a custom layout, you might use this behavior to position cells at regular intervals and create an animated effect.

Designing Your Data Source and Delegate

Every collection view must have a data source object. The **data source object** is the content that your app displays. It could be an object from your app's data model, or it could be the view controller that manages the collection view. The only requirement of the data source is that it must be able to provide information that the collection view needs, such as how many items there are and which views to use when displaying those items.

The **delegate object** is an optional (but recommended) object that manages aspects related to the presentation of and interaction with your content. Although the delegate's main job is to manage cell highlighting and selection, it can be extended to provide additional information. For example, the flow layout extends the basic delegate behavior to customize layout metrics, such as the size of cells and the spacing between them.

The Data Source Manages Your Content

The data source object is the object responsible for managing the content you are presenting using a collection view. The data source object must conform to the `UICollectionViewDataSource` protocol, which defines the basic behavior and methods that you must support. The job of the data source is to provide the collection view with answers to the following questions:

- How many sections does the collection view contain?
- For a given section, how many items does a section contain?
- For a given section or item, what views should be used to display the corresponding content?

Sections and **items** are the fundamental organizing principle for collection view content. A collection view typically has at least one section and may contain more. Each section, in turn, contains zero or more items. Items represent the main content you want to present, whereas sections organize those items into logical groups. For example, a photo app might use sections to represent a single album of photos or a set of photos taken on the same day.

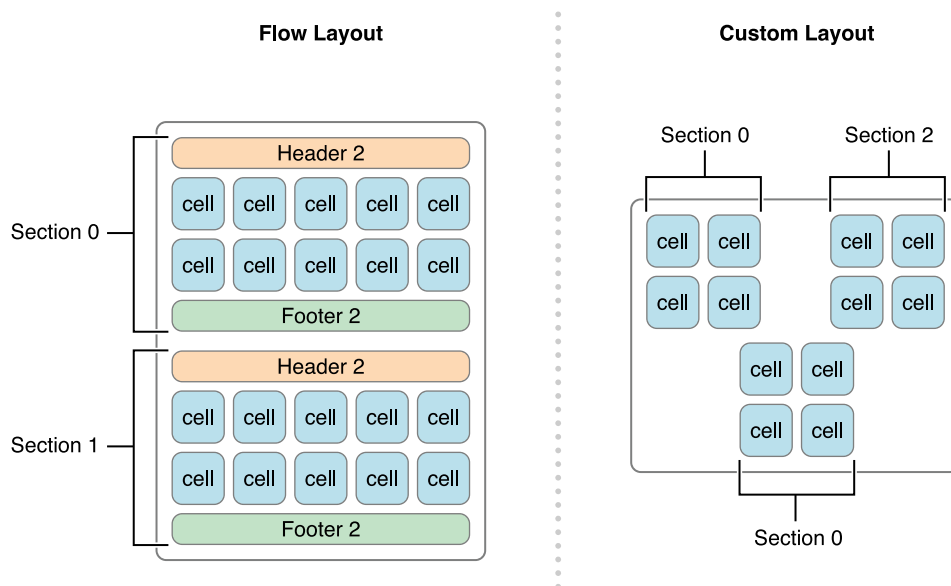
The collection view refers to the data it contains using `NSIndexPath` objects. When trying to locate an item, the collection view uses the index path information provided to it by the layout object. For items, the index path contains a section number and an item number. For supplementary and decoration views, the index path contains whichever values were provided by the layout object. The meaning of the index paths attached to supplementary and decoration views is dependent on your app, though the first index corresponds to a specific

section in the data source. These views' index paths are more about identification than meaning, identifying which view of what kind is currently being considered. So, if for example you have supplementary views that create headers and footers for your sections as seen in the flow layout, the relevant information provided by the index path is the section referenced.

Note: Although standard index paths support multiple levels, the collection view's cells only supports index paths that are 2-levels deep with "section" and "item" parameters, much like the index paths for the `UITableView` class. Supplementary views and decoration views can have more complex index paths if necessary. Elements whose index paths are > 1 is interpreted to correspond to the section designated by the first index in the path. Traditionally, only a second index is necessary, but supplementary and decoration views are not restricted to just two. Keep this in mind when designing your data source.

No matter how you arrange the sections and items in your data object, the visual presentation of those sections and items is still determined by the layout object. Different layout objects could present section and item data very differently, as shown in Figure 2-1. In this figure, the flow layout object arranges the sections vertically with each successive section below the previous one. A custom layout could position the sections in a nonlinear arrangement, demonstrating again the separation of the layout from the actual data.

Figure 2-1 Sections arranged according to the arrangement of layout objects

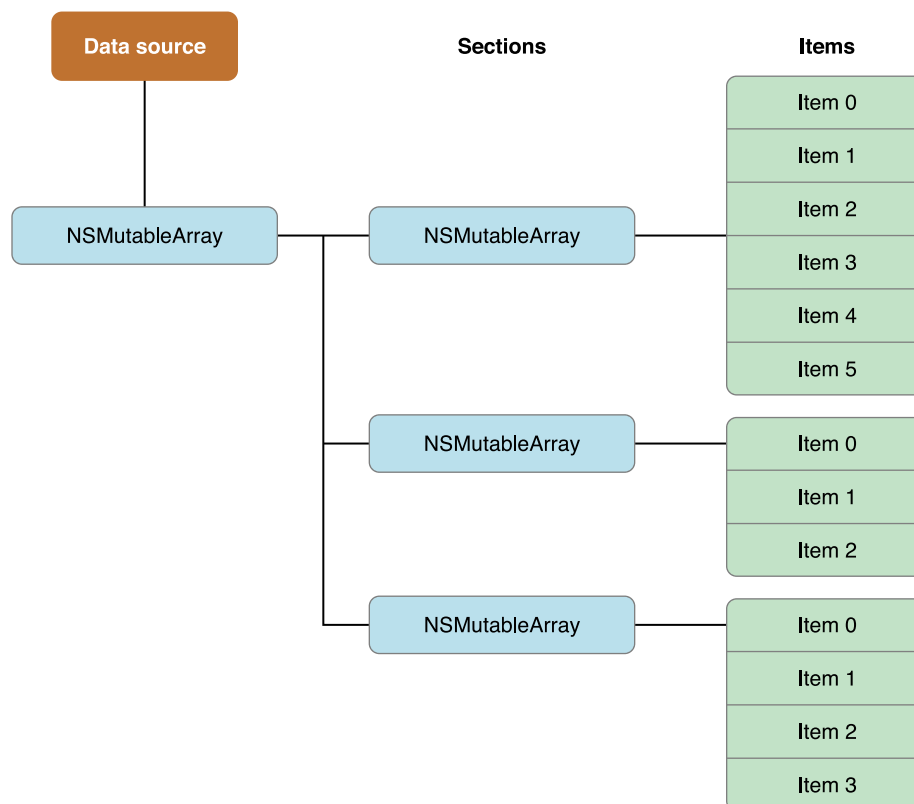


Designing Your Data Objects

An efficient data source uses sections and items to help organize its underlying data objects. Organizing your data into sections and items makes it much easier to implement your data source methods later. And because your data source methods are called frequently, you want to make sure that your implementations of those methods are able to retrieve data as quickly as possible.

One simple solution (but certainly not the only solution) is for your data source to use a set of nested arrays, as shown in Figure 2-2. In this configuration, a top-level array contains one or more arrays representing the sections of your data source. Each section array then contains the data items for that section. Finding an item in a section is a matter of retrieving its section array and then retrieving an item from that array. This type of arrangement makes it easy to manage moderately sized collections of items and retrieve individual items on demand.

Figure 2-2 Arranging data objects using nested arrays



When designing your data structures, you can always start with a simple set of arrays and move to a more efficient structure as needed. In general, your data objects should never be a performance bottleneck. The collection view usually accesses your data source only to calculate how many objects there are in total and to obtain views for elements that are currently onscreen. If the layout object relies only on data from your data objects, performance could be severely impacted when the data source contains thousands of objects.

Telling the Collection View About Your Content

Among the questions asked of your data source by the collection view are how many sections it contains and how many items each section contains. The collection view asks your data source to provide this information when any of the following actions occur:

- The collection view is displayed for the first time.
- You assign a different data source object to the collection view.
- You explicitly call the collection view's `reloadData` method.
- The collection view delegate executes a block using `performBatchUpdates:completion:` or any of the `move`, `insert`, or `delete` methods.

You provide the number of sections using the `numberOfSectionsInCollectionView:` method, and the number of items in each section using the `collectionView:numberOfItemsInSection:` method. You must implement the `collectionView:numberOfItemsInSection:` method, but if your collection view has only one section, implementing the `numberOfSectionsInCollectionView:` method is optional. Both methods return integer values with the appropriate information.

If you implemented your data source as shown in [Figure 2-2](#) (page 18), the implementation of your data source methods could be as simple as those shown in Listing 2-1. In this code, the `_data` variable is a custom member variable of the data source that stores the top-level array of sections. Obtaining the count of that array yields the number of sections. Obtaining the count of one of the subarrays yields the number of items in the section. (Of course, your own code should do whatever error checking is needed to ensure that the values returned are valid.)

Listing 2-1 Providing the section and item counts

```
- (NSInteger)numberOfSectionsInCollectionView:(UICollectionView*)collectionView {
    // _data is a class member variable that contains one array per section.
    return [_data count];
}

- (NSInteger)collectionView:(UICollectionView*)collectionView
numberOfItemsInSection:(NSInteger)section {
    NSArray* sectionArray = [_data objectAtIndex:section];
    return [sectionArray count];
}
```

Configuring Cells and Supplementary Views

Another important task of your data source is to provide the views that the collection view uses to display your content. The collection view does not track your app's content. It simply takes the views you give it and applies the current layout information to them. Therefore, everything that is displayed by the views is your responsibility.

After your data source reports how many sections and items it manages, the collection view asks the layout object to provide layout attributes for the collection view's content. At some point, the collection view asks the layout object to provide the list of elements in a specific rectangle (often this is the visible rectangle). The collection view uses that list to ask your data source for the corresponding cells and supplementary views. To provide those cells and supplementary views, your code must do the following:

1. Embed your template cells and views in your storyboard file. (Alternatively, register a class or nib file for each type of supported cell or view.)
2. In your data source, dequeue and configure the appropriate cell or view when asked.

To ensure that cells and supplementary views are used in the most efficient way possible, the collection view assumes the responsibility of creating those objects for you. Each collection view maintains internal queues of currently unused cells and supplementary views. Instead of creating objects yourself, simply ask the collection view to provide you with the view you want. If one is waiting on a reuse queue, the collection view prepares it and returns it to you quickly. If one is not waiting, the collection view uses the registered class or nib file to create a new one and return it to you. Thus, every time you dequeue a cell or view, you always get a ready-to-use object.

Reuse identifiers make it possible to register multiple types of cells and multiple types of supplementary views. A **reuse identifier** is a string that you use to distinguish between your registered cell and view types. The contents of the string are relevant only to your data source object. But when asked for a view or cell, you can use the provided index path to determine which type of view or cell you might want and then pass the appropriate reuse identifier to the dequeue method.

Registering Your Cells and Supplementary Views

You can configure the cells and views of your collection view programmatically or in your app's storyboard file.

Configure cells and views in your storyboard. When configuring cells and supplementary views in a storyboard, you do so by dragging the item onto your collection view and configuring it there. This creates a relationship between the collection view and the corresponding cell or view.

- For cells, drag a Collection View Cell from the object library and drop it on to your collection view. Set the custom class and the collection reusable view identifier of your cell to appropriate values.

- For supplementary views, drag a Collection Reusable View from the object library and drop it on to your collection view. Set the custom class and the collection reusable view identifier of your view to appropriate values.

Configure cells programmatically. Use either the `registerClass:forCellWithReuseIdentifier:` or `registerNib:forCellWithReuseIdentifier:` method to associate your cell with a reuse identifier. You might call these methods as part of the parent view controller's initialization process.

Configure supplementary views programmatically. Use either the `registerClass:forSupplementaryViewOfKind:withReuseIdentifier:` or `registerNib:forSupplementaryViewOfKind:withReuseIdentifier:` method to associate each kind of view with a reuse identifier. You might call these methods as part of the parent view controller's initialization process.

Although you register cells using only a reuse identifier, supplementary views require that you specify an additional identifier known as a **kind string**. Each layout object is responsible for defining the *kinds* of supplementary views it supports. For example, the `UICollectionViewFlowLayout` class supports two kinds of supplementary views: a section header view and a section footer view. To identify these two types of views, it defines the string constants `UICollectionViewElementKindSectionHeader` and `UICollectionViewElementKindSectionFooter`. During layout, the layout object includes the kind string with the other layout attributes for that view type. The collection view then passes the information along to your data source. Your data source then uses both the kind string and the reuse identifier to decide which view object to dequeue and return.

Note: If you implement your own custom layouts, you are responsible for defining the kinds of supplementary views your layout supports. A layout may support any number of supplementary views, each with its own kind string. For more information about defining custom layouts, see [“Creating Custom Layouts”](#) (page 40).

Registration is a one-time event that must take place before you attempt to dequeue any cells or views. After you've registered, you can dequeue as many cells or views as needed without reregistering them. It's not recommended that you change the registration information after dequeuing one or more items. It is better to register your cells and views once and be done with it.

Dequeueing and Configuring Cells and Views

Your data source object is responsible for providing cells and supplementary views when asked for them by the collection view. The `UICollectionViewDataSource` protocol contains two methods for this purpose: `collectionView:cellForItemAtIndexPath:` and `collectionView:viewForSupplementaryElementOfKind:atIndexPath:`. Because cells are a required

element of a collection view, your data source must implement the `collectionView:cellForItemAtIndexPath:` method, but the `collectionView:viewForSupplementaryElementOfKind:atIndexPath:` method is optional and dependent on the type of layout in use. In both cases, your implementation of these methods follows a very simple pattern:

1. Dequeue a cell or view of the appropriate type using the `dequeueReusableCellWithReuseIdentifier:forIndexPath:` or `dequeueReusableSupplementaryViewOfKind:withReuseIdentifier:forIndexPath:` method.
2. Configure the view using the data at the specified index path.
3. Return the view.

The dequeuing process is designed to relieve you of the responsibility of having to create a cell or view yourself. As long as you registered a cell or view previously, the dequeue methods are guaranteed to never return `nil`. If there is no cell or view of the given type on a reuse queue, the dequeue method simply creates one using your storyboard or using the class or nib file you registered.

The cell returned to you from the dequeuing process should be in a pristine state and ready to be configured with new data. For a cell or view that must be created, the dequeuing process creates and initializes it using the normal processes—that is, by loading the view from a storyboard or nib file or by creating a new instance and initializing it using the `initWithFrame:` method. In contrast, an item that wasn't created from scratch but that was instead retrieved from a reuse queue may already contain data from a previous usage. In that case, dequeue methods call the `prepareForReuse` method of the item to give it a chance to return itself to a pristine state. When you implement a custom cell or view class, you can override this method to reset properties to default values and perform any additional cleanup.

After your data source dequeues the view, it configures the view with its new data. You can use the index path passed to your data source methods to locate the appropriate data object and then apply that object's data to the view. After you configure the view, return it from your method and you are done. Listing 2-2 shows a simple example of how to configure a cell. After dequeuing the cell, the method sets the cell's custom label using the information about the cell's location and then returns the cell.

Listing 2-2 Configuring a custom cell

```
- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
    cellForItemAtIndexPath:(NSIndexPath *)indexPath {
    MyCustomCell* newCell = [self.collectionView
dequeueReusableCellWithReuseIdentifier:MyCellID
forIndexPath:indexPath];
```

```
newCell.cellLabel.text = [NSString stringWithFormat:@"Section:%d, Item:%d",  
indexPath.section, indexPath.item];  
return newCell;  
}
```

Note: When returning views from your datasource, always return a valid view. Returning `nil`, even if for some reason the view that is being asked for should not be displayed, causes an assertion and your app terminates because the layout object expects valid views to be returned by these methods.

Inserting, Deleting, and Moving Sections and Items

To insert, delete, or move a single section or item, follow these steps:

1. Update the data in your data source object.
2. Call the appropriate method of the collection view to insert or delete the section or item.

It is critical that you update your data source before notifying the collection view of any changes. The collection view methods assume that your data source contains the currently correct data. If it does not, the collection view might receive the wrong set of items from your data source or ask for items that are not there and crash your app.

When you add, delete, or move a single item programmatically, the collection view's methods automatically create animations to reflect the changes. If you want to animate multiple changes together, though, you must perform all insert, delete, or move calls inside a block and pass that block to the `performBatchUpdates:completion:` method. The batch update process then animates all of your changes at the same time and you can freely mix calls to insert, delete, or move items within the same block.

Listing 2-3 shows a simple example of how to perform a batch update to delete the currently selected items. The block passed to the `performBatchUpdates:completion:` method first calls a custom method to update the data source. It then tells the collection view to delete the items. Both the update block and the completion block you provide are executed synchronously.

Listing 2-3 Deleting the selected items

```
[self.collectionView performBatchUpdates:^(  
    NSArray* itemPaths = [self.collectionView indexPathsForSelectedItems];
```

```
// Delete the items from the data source.  
[self deleteItemsFromDataSourceAtIndexPaths:itemPaths];  
  
// Now delete the items from the collection view.  
[self.collectionView deleteItemsAtIndexPaths:tempArray];  
} completion:nil];
```

Managing the Visual State for Selections and Highlights

Collection views support single-item selection by default and can be configured to support multiple-item selection or have selections disabled altogether. The collection view detects taps inside its bounds and highlights or selects the corresponding cell accordingly. For the most part, the collection view modifies only the properties of a cell to indicate that it is selected or highlighted; it does not change the visual appearance of your cells, with one exception. If a cell's `selectedBackgroundView` property contains a valid view, the collection view shows that view when the cell is highlighted or selected.

Listing 2-4 shows code that could be incorporated into your implementation of a custom collection view cell to facilitate a changing appearance for highlighted and selected states. The cell's `backgroundView` property will always be the default view when the cell loads for the first time and when the cell is either not highlighted or not selected. The `selectedBackgroundView` property replaces the default background view whenever a cell is highlighted or selected. In this case, the cell's background color would be changed from red to white when selected or highlighted.

Listing 2-4 Setting the background views to indicate changed states

```
UIView* backgroundView = [[UIView alloc] initWithFrame:self.bounds];  
backgroundView.backgroundColor = [UIColor redColor];  
self.backgroundView = backgroundView;  
  
UIView* selectedBGView = [[UIView alloc] initWithFrame:self.bounds];  
selectedBGView.backgroundColor = [UIColor whiteColor];  
self.selectedBackgroundView = selectedBGView;
```

The collection view's delegate provides the collection view with the following methods to facilitate highlighting and selecting:

- `collectionView:shouldSelectItemAtIndexPath:`

- `collectionView:shouldDeselectItemAtIndexPath:`
- `collectionView:didSelectItemAtIndexPath:`
- `collectionView:didDeselectItemAtIndexPath:`
- `collectionView:shouldHighlightItemAtIndexPath:`
- `collectionView:didHighlightItemAtIndexPath:`
- `collectionView:didUnhighlightItemAtIndexPath:`

These methods provide you with many opportunities to tweak the highlighting/selecting behavior of your collection view to the exact desired specifications.

For example, if you prefer to draw the selection state of a cell yourself, you can leave the `selectedBackgroundView` property set to `nil` and apply any visual changes to the cell using your delegate object. You would apply the visual changes in the `collectionView:didSelectItemAtIndexPath::method` and remove them in the `collectionView:didDeselectItemAtIndexPath: method`.

If you prefer to draw the highlight state yourself, you can override the `collectionView:didHighlightItemAtIndexPath:` and `collectionView:didUnhighlightItemAtIndexPath:` delegate methods and use them to apply your highlights. If you also specified a view in the `selectedBackgroundView` property, you should make your changes to the content view of the cell to ensure your changes are visible. Listing 2-5 shows a simple way of changing the highlight using the content view's background color.

Listing 2-5 Applying a temporary highlight to a cell

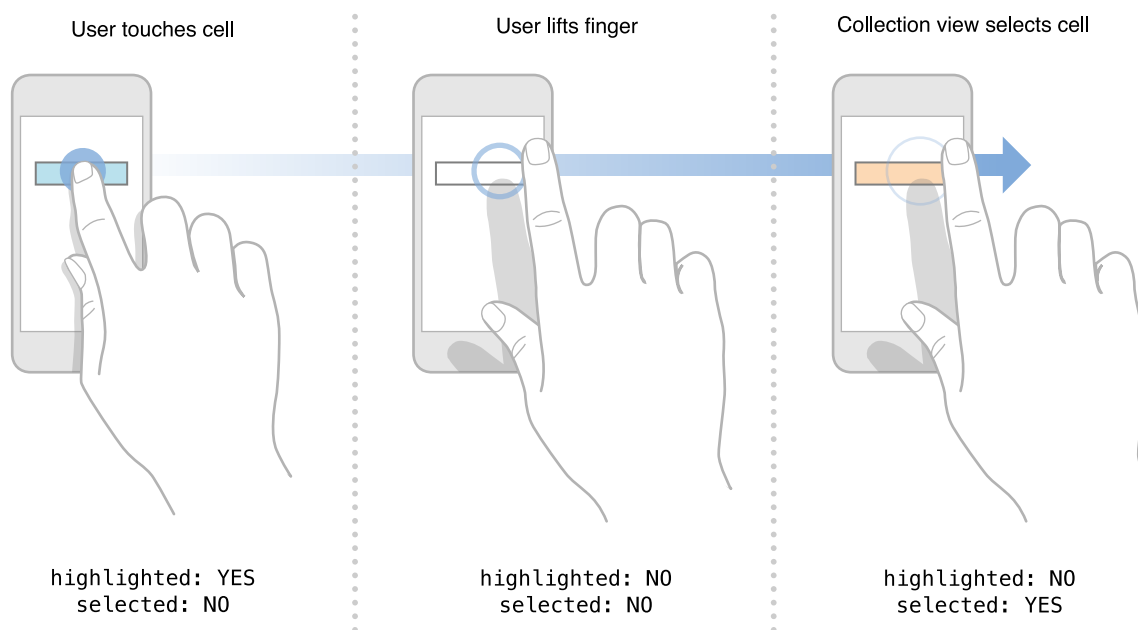
```
- (void)collectionView:(UICollectionView *)collectionView
didHighlightItemAtIndexPath:(NSIndexPath *)indexPath {
    UICollectionViewCell* cell = [collectionView cellForItemAtIndexPath:indexPath];
    cell.contentView.backgroundColor = [UIColor blueColor];
}

- (void)collectionView:(UICollectionView *)collectionView
didUnhighlightItemAtIndexPath:(NSIndexPath *)indexPath {
    UICollectionViewCell* cell = [collectionView cellForItemAtIndexPath:indexPath];
    cell.contentView.backgroundColor = nil;
}
```

There is a subtle but important distinction between a cell's highlighted state and its selected state. The **highlighted state** is a transitional state that you can use to apply visible highlights to the cell while the user's finger is still touching the device. This state is set to YES only while the collection view is tracking touch events over the cell. When touch events stop, the highlighted state returns to the value NO. By contrast, the **selected state** changes only after a series of touch events has ended—specifically, when those touch events indicated that the user tried to select the cell.

Figure 2-3 illustrates the series of steps that occurs when a user touches an unselected cell. The initial touch-down event causes the collection view to change the highlighted state of the cell to YES, although doing so does not automatically change the appearance of the cell. If the final touch up event occurs in the cell, the highlighted state returns to NO and the collection view changes the selected state to YES. When the user changes the selected state, the collection view displays the view in the cell's `selectedBackgroundView` property, but this is the only visual change that the collection view makes to the cell. Any other visual changes must be made by your delegate object.

Figure 2-3 Tracking touches in a cell



Whether the user is selecting or deselecting a cell, the cell's selected state is always the last thing to change. Taps in a cell always result in changes to the cell's highlighted state first. Only after the tap sequence ends and any highlights applied during that sequence are removed, does the selected state of the cell change. When designing your cells, you should make sure that the visual appearance of your highlights and selected state do not conflict in unintended ways.

Showing the Edit Menu for a Cell

When the user performs a long-tap gesture on a cell, the collection view attempts to display an Edit menu for that cell. The Edit menu can be used to cut, copy, and paste cells in the collection view. Several conditions must be met before the Edit menu can be displayed:

- The delegate must implement all three methods related to handling actions:
`collectionView:shouldShowMenuForItemAtIndexPath:`
`collectionView:canPerformAction:forItemAtIndexPath:withSender:`
`collectionView:performAction:forItemAtIndexPath:withSender:`
- The `collectionView:shouldShowMenuForItemAtIndexPath:` method must return YES for the indicated cell.
- The `collectionView:canPerformAction:forItemAtIndexPath:withSender:` method must return YES for at least one of the desired actions. The collection view supports the following actions:
`cut:`
`copy:`
`paste:`

If these conditions are met and the user chooses an action from the menu, the collection view calls the delegate's `collectionView:performAction:forItemAtIndexPath:withSender:` method to perform the action on the indicated item.

Listing 2-6 shows how to prevent one of the menu items from appearing. In this example, the `collectionView:canPerformAction:forItemAtIndexPath:withSender:` method prevents the Cut menu item from appearing in the Edit menu. It enables the Copy and Paste items so that the user can insert content.

Listing 2-6 Selectively disabling actions in the Edit menu

```
- (BOOL)collectionView:(UICollectionView *)collectionView
    canPerformAction:(SEL)action
    forItemAtIndexPath:(NSIndexPath *)indexPath
    withSender:(id)sender {
    // Support only copying and pasting of cells.
    if ([NSStringFromSelector(action) isEqualToString:@"copy:"]
        || [NSStringFromSelector(action) isEqualToString:@"paste:"])
        return YES;
```

```
// Prevent all other actions.  
return NO;  
}
```

For more information on working with the pasteboard commands, see *Text Programming Guide for iOS*.

Transitioning Between Layouts

The easiest way to transition between layouts is by using the `setCollectionViewLayout:animated:` method. However, if you require control of the transition or want it to be interactive, use a `UICollectionViewTransitionLayout` object.

The `UICollectionViewTransitionLayout` class is a special type of layout that gets installed as the collection view's layout object when transitioning to a new layout. With a transition layout object, you can have objects follow a non linear path, use a different timing algorithm, or move according to incoming touch events. The standard class provides a linear transition to a new layout, but like the `UICollectionViewLayout` class, the `UICollectionViewTransitionLayout` class can be subclassed to create any desired effect. In doing so, you need to implement the same methods you would when creating a custom layout and allow your implementation to adapt to input from the user, most often from a gesture recognizer. For more information about creating custom layout objects, see [“Creating Custom Layouts”](#) (page 40).

The `UICollectionViewLayout` class provides several methods for tracking the transition between layouts. `UICollectionViewTransitionLayout` objects track the completion of a transition through the `transitionProgress` property. As the transition occurs, your code updates this property periodically to indicate the completion percentage of the transition. For example, using the `UICollectionViewTransitionLayout` class in conjunction with objects like gesture recognizers, which you can use to transition between layouts, allows you to create interactive transitions. As well, if you implement a custom transition layout object, the `UICollectionViewTransitionLayout` class provides two methods for tracking values relevant to your layout: the `updateValue:forAnimatedKey:` and `valueForAnimatedKey:` methods. These methods track special floating point values that you can set and change during a transition to communicate to the layout important information. For example, if you transitioned between layouts using a pinch gesture, you could use these methods to tell the transition layout object at what offset the view's need to be from one another.

The steps for including a `UICollectionViewTransitionLayout` object in your app are as follows:

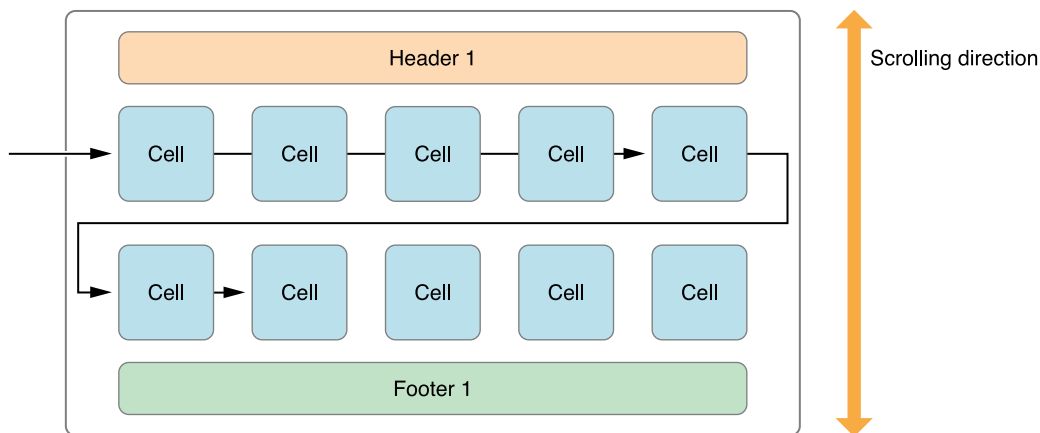
1. Create an instance of the standard class or your own custom class using the `initWithCurrentLayout:nextLayout:` method.

2. Communicate the progress of the transition by periodically modifying the `transitionProgress` property. Do not forget to invalidate the layout using the collection view's `invalidateLayout` method after changing the transition's progress.
3. Implement the `collectionView:transitionLayoutForOldLayout:newLayout:` method in your collection view's delegate and return your transition layout object.
4. Optionally modify values for your layout using the `updateValue:forAnimatedKey:` method to indicate changed values relevant to your layout object. The stable value in this case is 0.

Using the Flow Layout

You can arrange items in your collection views using a concrete layout object, the `UICollectionViewFlowLayout` class. The flow layout implements a line-based breaking layout, which means that the layout object places cells on a linear path and fits as many cells along that line as it can. When the layout object runs out of room on the current line, it creates a new line and continues the layout process there. Figure 3-1 shows what this looks like for a flow layout that scrolls vertically. In this case, lines are laid out horizontally with each new line positioned below the previous line. The cells in a single section can be optionally surrounded with section header and section footer views.

Figure 3-1 Laying out sections and cells using the flow layout



You can use the flow layout to implement grids, but you can also use it for much more. The idea of a linear layout can be applied to many different designs. For example, rather than having a grid of items, you can adjust the spacing to create a single line of items along the scrolling dimension. Items can also be different sizes, which yields something more asymmetrical than a traditional grid but that still has a linear flow to it. There are many possibilities.

You can configure the flow layout either programmatically or using Interface Builder in Xcode. The steps for configuring the flow layout are as follows:

1. Create a flow layout object and assign it to your collection view.
2. Configure the width and height of cells.
3. Set the spacing options (as needed) for the lines and items.
4. If you want section headers or section footers, specify their size.

5. Set the scroll direction for the layout.

Important: At a minimum, you must specify the width and height of cells. If you don't, your items are assigned a width and height of 0 and will never be visible.

Customizing the Flow Layout Attributes

The flow layout object exposes several properties for configuring the appearance of your content. When set, these properties are applied to all items equally in the layout. For example, setting the cell size using the `itemSize` property of the flow layout object causes all cells to have the same size.

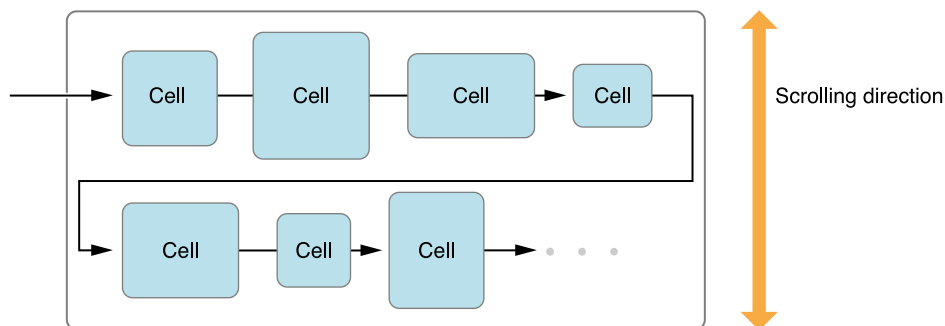
If you want to vary the spacing or size of items dynamically, you can do so using the methods of the `UICollectionViewDelegateFlowLayout` protocol. You implement these methods on the same delegate object you assigned to the collection view itself. If a given method exists, the flow layout object calls that method instead of using the fixed value it has. Your implementation must then return appropriate values for all of the items in the collection view.

Specifying the Size of Items in the Flow Layout

If all of the items in the collection view are the same size, assign the appropriate width and height values to the `itemSize` property of the flow layout object. (Always specify the size of items in points.) This is the fastest way to configure the layout object for content whose size does not vary.

If you want to specify different sizes for your cells, you must implement the `collectionView:layout:sizeForItemAtIndexPath:` method on the collection view delegate. You can use the provided index path information to return the size of the corresponding item. During layout, the flow layout object centers items vertically on the same line, as shown in Figure 3-2. The overall height or width of the line is then determined by the largest item in that dimension.

Figure 3-2 Items of different sizes in the flow layout



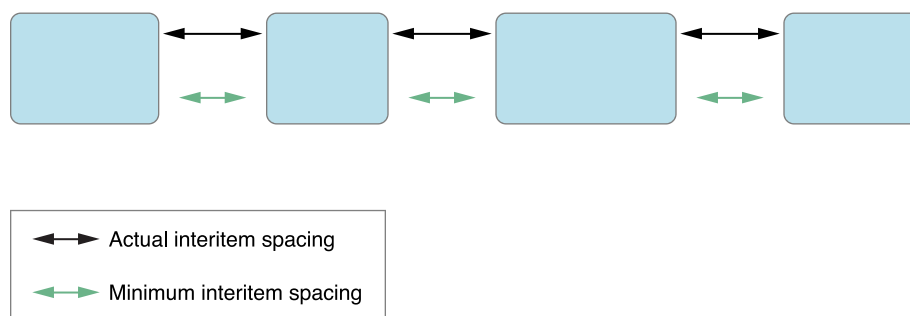
Note: When you specify different sizes for cells, the number of items on a single line can vary from line to line.

Specifying the Space Between Items and Lines

Using the flow layout, you can specify the minimum spacing between items on the same line and the minimum spacing between successive lines. Keep in mind that the spacing you provide is only the minimum spacing. Because of how it lays out content, the flow layout object may increase the spacing between items to a value greater than the one you specified. The layout object may similarly increase the actual line-spacing when the items being laid out are different sizes.

During layout, the flow layout object adds items to the current line until there is not enough space left to fit an entire item. If the line is just big enough to fit an integral number of items with no extra space, then the space between the items would be equal to the minimum spacing. If there is extra space at the end of the line, the layout object increases the interitem spacing until the items fit evenly within the line boundaries, as shown in Figure 3-3. Increasing the spacing improves the overall look of the items and prevents large gaps at the end of each line.

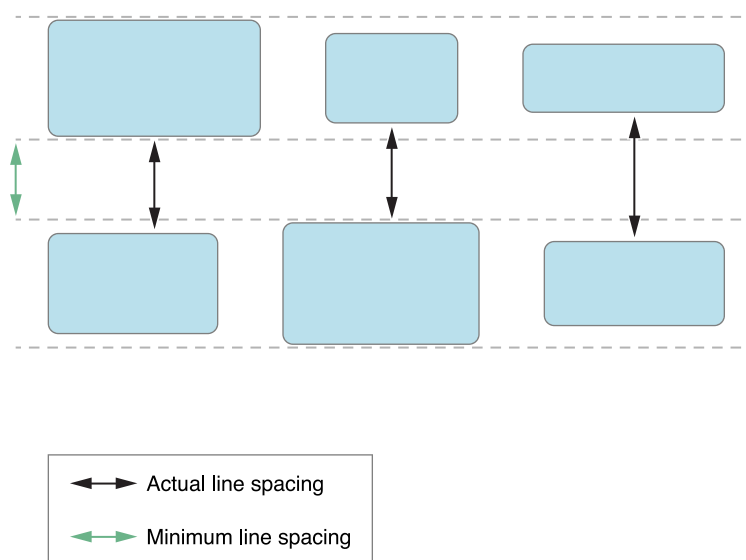
Figure 3-3 Actual spacing between items may be greater than the minimum



For interline spacing, the flow layout object uses the same technique that it does for inter-item spacing. If all items are the same size, the flow layout is able to respect the minimum line spacing value absolutely and all items in one line appear to be spaced evenly from the items in the next line. If the items are of different sizes, the actual spacing between individual items can vary.

Figure 3-4 demonstrates what happens with the minimum line spacing when items are of different sizes. With differently sized items, the flow layout object picks the item from each line whose dimension in the scrolling direction is the largest. For example, in a vertically scrolling layout, it looks for the item in each line with the greatest height. It then sets the spacing between those items to the minimum value. If the items are on different parts of the line, as shown in the figure, the actual line spacing appears to be greater than the minimum.

Figure 3-4 Line spacing varies if items are of different sizes



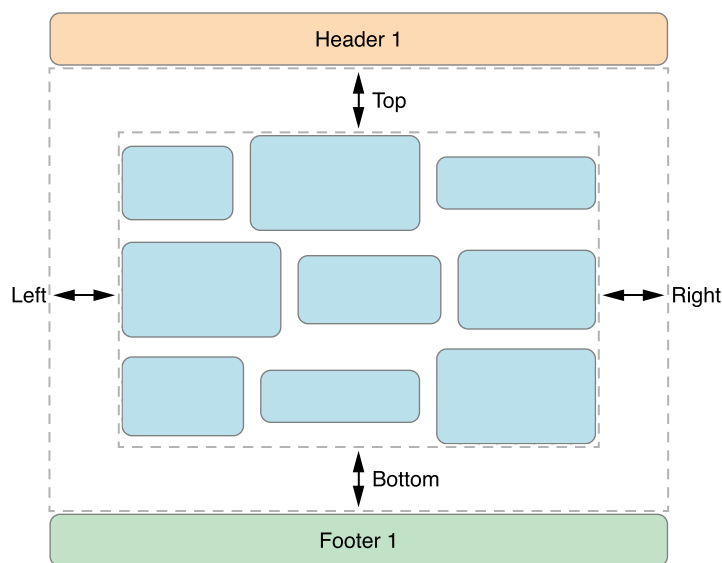
As with other flow layout attributes, you can use fixed spacing values or vary the values dynamically. Line and item spacing is handled on a section-by-section basis. Thus, the line and interitem spacing is the same for all of the items in a given section but may vary between sections. You set the spacing statically using the `minimumLineSpacing` and `minimumInteritemSpacing` properties of the flow layout object or using the `collectionView:layout:minimumLineSpacingForSectionAtIndex:` and `collectionView:layout:minimumInteritemSpacingForSectionAtIndex:` methods of your collection view delegate.

Using Section Insets to Tweak the Margins of Your Content

Section insets are a way to adjust the space available for laying out cells. You can use insets to insert space after a section's header view and before its footer view. You can also use insets to insert space around the sides of the content. Figure 3-5 demonstrates how insets affect some content in a vertically scrolling flow layout.

Figure 3-5 Section insets change the available space for laying out cells

```
inset = UIEdgeInsetsMake(top, left, bottom, right)
```



Because insets reduce the amount of space available for laying out cells, you can use them to limit the number of cells in a given line. Specifying insets in the nonscrolling direction is one way to constrict the space for each line. If you combine that information with an appropriate cell size, you can control the number of cells on each line.

Knowing When to Subclass the Flow Layout

Although you can use the flow layout very effectively without subclassing, there are still times when you might need to subclass to get the behavior you need. Table 3-1 lists some of the scenarios for which subclassing `UICollectionViewFlowLayout` is necessary to achieve the desired effect.

Table 3-1 Scenarios for subclassing UICollectionViewFlowLayout

Scenario	Subclassing tips
You want to add new supplementary or decoration views to your layout	<p>The standard flow layout class supports only section header and section footer views and no decoration views. To support additional supplementary and decoration views, you need to override the following methods at a minimum:</p> <ul style="list-style-type: none"> • <code>layoutAttributesForElementsInRect:</code> (required) • <code>layoutAttributesForItemAtIndexPath:</code> (required) • <code>layoutAttributesForSupplementaryViewOfKind: atIndexPath:</code> (to support new supplementary views) • <code>layoutAttributesForDecorationViewOfKind: atIndexPath:</code> (to support new decoration views) <p>In your <code>layoutAttributesForElementsInRect:</code> method, you can call <code>super</code> to get the layout attributes for the cells and then add the attributes for any new supplementary or decoration views that are in the specified rectangle. Use the other methods to provide attributes on demand.</p> <p>For information about providing attributes for views during layout, see “Creating Layout Attributes” (page 43) and “Providing Layout Attributes for Items in a Given Rectangle” (page 44).</p>
You want to tweak the layout attributes being returned by the flow layout	<p>Override the <code>layoutAttributesForElementsInRect:</code> method and any of the methods that return layout attributes. The implementation of your methods should call <code>super</code>, modify the attributes provided by the parent class, and then return them.</p> <p>For in-depth discussions of what these methods entail, see “Creating Layout Attributes” (page 43) and “Providing Layout Attributes for Items in a Given Rectangle” (page 44).</p>
You want to add new layout attributes for your cells and views	<p>Create a custom subclass of <code>UICollectionViewLayoutAttributes</code> and add whatever properties you need to represent your custom layout information.</p> <p>Subclass <code>UICollectionViewFlowLayout</code> and override the <code>layoutAttributesClass</code> method. In your implementation of that method, return your custom subclass.</p> <p>You should also override the <code>layoutAttributesForElementsInRect:</code> method, the <code>layoutAttributesForItemAtIndexPath:</code> method, and any other methods that return layout attributes. In your custom implementations, you should set the values for any custom attributes you defined.</p>

Scenario	Subclassing tips
You want to specify initial or final locations for items being inserted or deleted	<p>By default, a simple fade animation is created for items being inserted or deleted. To create custom animations, you must override some or all of the following methods:</p> <ul style="list-style-type: none">• <code>initialLayoutAttributesForAppearingItemAtIndexPath:</code>• <code>initialLayoutAttributesForAppearingSupplementaryElementOfKind:atIndexPath:</code>• <code>initialLayoutAttributesForAppearingDecorationElementOfKind:atIndexPath:</code>• <code>finalLayoutAttributesForDisappearingItemAtIndexPath:</code>• <code>finalLayoutAttributesForDisappearingSupplementaryElementOfKind:atIndexPath:</code>• <code>finalLayoutAttributesForDisappearingDecorationElementOfKind:atIndexPath:</code> <p>In your implementations of these methods, specify the attributes you want each view to have prior to being inserted or after they are removed. The flow layout object uses the attributes you provide to animate the insertions and deletions.</p> <p>If you override these methods, it is also recommended that you override the <code>prepareForCollectionViewUpdates:</code> and <code>finalizeCollectionViewUpdates</code> methods. You can use these methods to track which items are being inserted or deleted during the current cycle.</p> <p>For more information about how insertions and deletions work, see “Making Insertion and Deletion Animations More Interesting” (page 49).</p>

There are also instances in which the right thing to do is to create a custom layout from scratch. Before you decide to do this, take the time to consider whether or not it is really necessary. The flow layout provides a lot of customizable behavior that is appropriate for many different kinds of layouts, and because it is provided to you, it is easy to use and contains numerous optimizations to make it efficient. However, all this is not to say that you should never create a custom layout, because there are circumstances in which doing so make absolute sense. The flow layout limits the scroll direction to one direction, so if your layout contains content that stretches farther than the bounds of the screen in both directions, a custom layout makes more sense to implement. Creating a custom layout is the right decision if your layout is not a grid or a line-based breaking layout, as described above, or if the items within your layout move so frequently that subclassing the flow layout is more complicated than creating your own.

For more on creating a custom layout, see [“Creating Custom Layouts”](#) (page 40).

Incorporating Gesture Support

You can add greater interactivity to your collection views through the use of gesture recognizers. Add a gesture recognizer to a collection view, and use it to trigger actions when those gestures occur. For a collection view, there are two types of actions that you might likely want to implement:

- You want to trigger changes to the collection view’s layout information.
- You want to manipulate cells and views directly.

You should always attach your gesture recognizers to the collection view itself—not to a specific cell or view. The `UICollectionView` class is a descendant of `UIScrollView`, so attaching your gesture recognizers to the collection view is less likely to interfere with the other gestures that must be tracked. In addition, because the collection view has access to your data source and your layout object, you still have access to all the information you need to manipulate cells and views appropriately.

Using a Gesture Recognizer to Modify Layout Information

A gesture recognizer offers an easy way to modify layout parameters dynamically. For example, you might use a pinch gesture recognizer to change the spacing between items in a custom layout. The process for configuring such a gesture recognizer is relatively simple.

1. Create the gesture recognizer.
2. Attach the gesture recognizer to the collection view.
3. Use the handler method of the gesture recognizer to update the layout parameters and invalidate the layout object.

You create a gesture recognizer using the same `alloc/init` process that you use for all objects. During initialization, you specify the target object and action method to call when the gesture is triggered. You then call the collection view’s `addGestureRecognizer:` method to attach it to the view. Most of the actual work happens in the action method you specify at initialization time.

Listing 4-1 shows an example of an action method that is called by a pinch gesture recognizer attached to a collection view. In this example, the pinch data is used to change the distance between cells in a custom layout. The layout object implements the custom `updateSpreadDistance` method, which validates the new distance value and stores it for use during the layout process later. The action method then invalidates the layout and forces it to update the position of items based on the new value.

Listing 4-1 Using a gesture recognizer to change layout values

```
- (void)handlePinchGesture:(UIPinchGestureRecognizer *)sender {
    if ([sender numberOfTouches] != 2)
        return;

    // Get the pinch points.
    CGPoint p1 = [sender locationOfTouch:0 inView:[self collectionView]];
    CGPoint p2 = [sender locationOfTouch:1 inView:[self collectionView]];

    // Compute the new spread distance.
    CGFloat xd = p1.x - p2.x;
    CGFloat yd = p1.y - p2.y;
    CGFloat distance = sqrt(xd*xd + yd*yd);

    // Update the custom layout parameter and invalidate.
    MyCustomLayout* myLayout = (MyCustomLayout*)[[self collectionView]
collectionViewLayout];
    [myLayout updateSpreadDistance:distance];
    [myLayout invalidateLayout];
}
```

For more information about creating gesture recognizers and attaching them to views, see *Event Handling Guide for iOS*.

Working with Default Gesture Behaviors

The `UICollectionView` class listens for single taps to initiate its delegate methods for highlighting and selecting. If you want to add custom tap or long-press gestures to a collection view, configure the values of your gesture recognizer to be different than the values the collection view already uses. For example, you might configure a tap gesture recognizer to respond only to double-taps.

Listing 4-2 shows how you might make the collection view respond to your gesture instead of listening for cell selection/highlighting. Because the collection view does not use a gesture recognizer to initiate its delegate methods, your custom gesture recognizer gets priority over the default selection listeners by delaying the registering of other touch events by setting the `delaysTouchesBegan` property of your gesture recognizer to YES or cancelling touch events by setting the `cancelsTouchesInView` property of your gesture recognizer to YES. Whenever a tap is registered, it will first check to see if your gesture recognizer should have priority or not. If the input is not valid for your gesture recognizer, then the delegate methods will be called as normal.

Listing 4-2 Prioritizing your gesture recognizer

```
UITapGestureRecognizer* tapGesture = [[UITapGestureRecognizer alloc]
initWithTarget:self action:@selector(handleTapGesture:)];
tapGesture.delaysTouchesBegan = YES;
tapGesture.numberOfTapsRequired = 2;
[self.collectionView addGestureRecognizer:tapGesture];
```

Manipulating Cells and Views

How you use a gesture recognizer to manipulate cells and views depends on the types of manipulations you plan to make. Simple insertions and deletions can be performed inside the action method of a standard gesture recognizer. But if you plan more complex manipulations, you probably need to define a custom gesture recognizer to track the touch events yourself.

One type of manipulation that requires a custom gesture recognizer to move a cell in your collection view from one location to another. The most straightforward way to move a cell is to delete it (temporarily) from the collection view, use the gesture recognizer to drag around a visual representation of that cell, and then insert the cell at its new location when the touch events end. All of this requires managing the touch events yourself, working closely with the layout object to determine the new insertion location, manipulating the data source changes, and then inserting the item at the new location.

For more information about creating custom gesture recognizers, see *Event Handling Guide for iOS*.

Creating Custom Layouts

Before you start building custom layouts, consider whether doing so is really necessary. The `UICollectionViewFlowLayout` class provides a significant amount of behavior that has already been optimized for efficiency and that can be adapted in several ways to achieve many different types of standard layouts. The only times to consider implementing a custom layout are in the following situations:

- The layout you want looks nothing like a grid or a line-based breaking layout (a layout in which items are placed into a row until it's full, then continue on to the next line until all items are placed) or necessitates scrolling in more than one direction.
- You want to change all of the cell positions frequently enough that it would be more work to modify the existing flow layout than to create a custom layout.

The good news is that, from an API perspective, implementing a custom layout is not difficult. The hardest part is performing the calculations needed to determine the positions of items in the layout. When you know the locations of those items, providing that information to the collection view is straightforward.

Subclassing `UICollectionViewLayout`

For custom layouts, you want to subclass `UICollectionViewLayout`, which provides you with a fresh starting point for your design. Only a handful of methods provide the core behavior for your layout object and are required in your implementation. The rest of the methods are there for you to override as needed to tweak the layout behavior. The core methods handle the following crucial tasks:

- Specify the size of the scrollable content area.
- Provide attribute objects for the cells and views that make up your layout so that the collection view can position each cell and view.

Although you can create a functional layout object that implements just the core methods, your layout is likely to be more engaging if you implement several of the optional methods as well.

The layout object uses information provided by its data source to create the collection view's layout. Your layout communicates with the data source by calling methods on the `collectionView` property, which is accessible in all of the layout's methods. Keep in mind what your collection view knows and doesn't know during the layout process. Because the layout process is under way, the collection view cannot track the layout

or positioning of views. So even though the layout object will not restrict you from calling any of the collection view's methods, refrain from relying on the collection view for anything other than the data necessary to compute your layout.

Understanding the Core Layout Process

The collection view works directly with your custom layout object to manage the overall layout process. When the collection view determines that it needs layout information, it asks your layout object to provide it. For example, the collection view asks for layout information when it is first displayed or is resized. You can also tell the collection view to update its layout explicitly by calling the `invalidateLayout` method of the layout object. That method throws away the existing layout information and forces the layout object to generate new layout information.

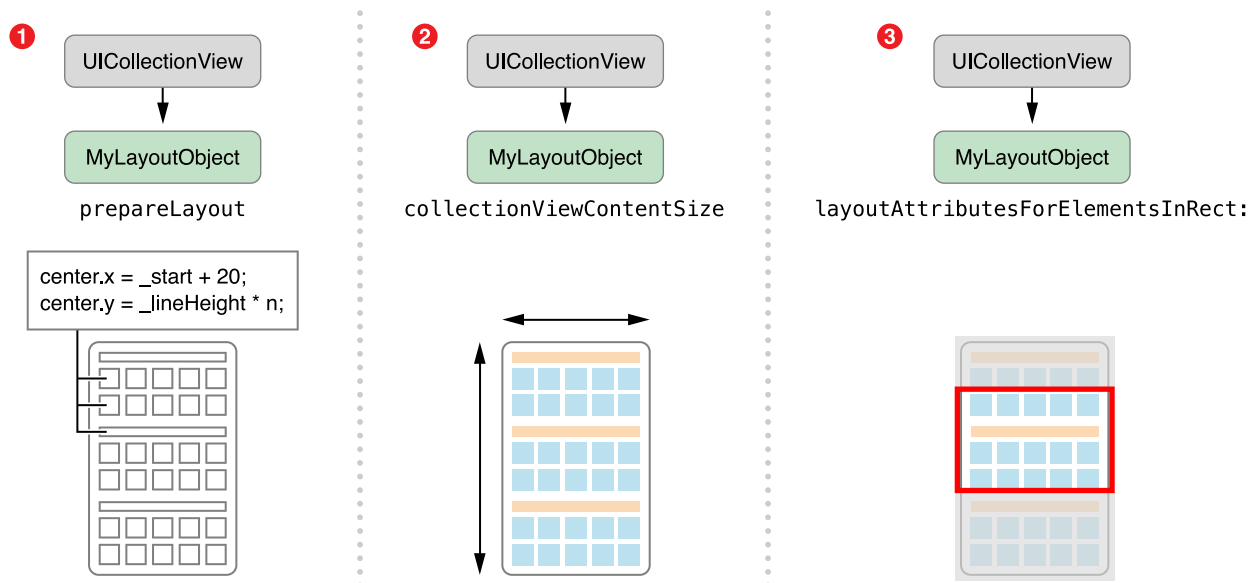
Note: Be careful not to confuse the layout object's `invalidateLayout` method with the collection view's `reloadData` method. Calling the `invalidateLayout` method does not necessarily cause the collection view to throw out its existing cells and subviews. Rather, it forces the layout object to recompute all of its layout attributes as is necessary when moving and adding or deleting items. If data within the data source has changed, the `reloadData` method is appropriate. Regardless of how you initiate a layout update, the actual layout process is the same.

During the layout process, the collection view calls specific methods of your layout object. These methods are your chance to calculate the position of items and to provide the collection view with the primary information it needs. Other methods may be called, too, but these methods are always called during the layout process in the following order:

1. Use the `prepareLayout` method to perform the up-front calculations needed to provide layout information.
2. Use the `collectionViewContentSize` method to return the overall size of the entire content area based on your initial calculations.
3. Use the `layoutAttributesForElementsInRect:` method to return the attributes for cells and views that are in the specified rectangle.

Figure 5-1 illustrates how you can use the preceding methods to generate your layout information.

Figure 5-1 Laying out your custom content



The `prepareLayout` method is your opportunity to perform whatever calculations are needed to determine the position of the cells and views in the layout. At a minimum, you should compute enough information in this method to be able to return the overall size of the content area, which you return to the collection view in step 2.

The collection view uses the content size to configure its scroll view appropriately. For instance, if your computed content size expands past the bounds of the current device's screen both vertically and horizontally, the scroll view adjusts to allow scrolling in both directions simultaneously. Unlike the `UICollectionViewFlowLayout`, it does not by default adjust the layout of content to scroll in only one direction.

Based on the current scroll position, the collection view then calls your `layoutAttributesForElementsInRect:` method to ask for the attributes of the cells and views in a specific rectangle, which may or may not be the same as the visible rectangle. After returning that information, the core layout process is effectively complete.

After layout finishes, the attributes of your cells and views remain the same until you or the collection view invalidates the layout. Calling the `invalidateLayout` method of your layout object causes the layout process to begin again, starting with a new call to the `prepareLayout` method. The collection view can also invalidate your layout automatically during scrolling. If the user scrolls its content, the collection view calls the layout object's `shouldInvalidateLayoutForBoundsChange:` method and invalidates the layout if that method returns YES.

Note: It is useful to remember that calling the `invalidateLayout` method does not begin the layout update process immediately. The method merely marks the layout as being inconsistent with the data and in need of being updated. During the next view update cycle, the collection view checks to see whether its layout is dirty and updates it if it is. In fact, you can call the `invalidateLayout` method multiple times in quick succession without triggering an immediate layout update each time.

Creating Layout Attributes

The attributes objects that your layout is responsible for are instances of the `UICollectionViewLayoutAttributes` class. These instances can be created in a variety of different methods in your app. When your app is not dealing with thousands of items, it makes sense to create these instances while preparing the layout, because the layout information can be cached and referenced rather than computed on the fly. If the costs of computing all the attributes up front outweighs the benefits of caching in your app, it is just as easy to create attributes in the moment when they are requested.

Regardless, when creating new instances of the `UICollectionViewLayoutAttributes` class, use one of the following class methods:

- `layoutAttributesForCellWithIndexPath:`
- `layoutAttributesForSupplementaryViewOfKind:withIndexPath:`
- `layoutAttributesForDecorationViewOfKind:withIndexPath:`

You must use the correct class method based on the type of the view being displayed because the collection view uses that information to request the appropriate type of view from the data source object. Using the incorrect method causes the collection view to create the wrong views in the wrong places and your layout does not appear as intended.

After creating each attributes object, set the relevant attributes for the corresponding view. At a minimum, set the size and position of the view in the layout. In cases where the views of your layout overlap, assign a value to the `zIndex` property to ensure a consistent ordering of the overlapping views. Other properties let you control the visibility or appearance of the cell or view and can be changed as needed. If the standard attributes class does not suit your app's needs, you can subclass and expand it to store other information about each view. When subclassing layout attributes, it's required that you implement the `isEqual:` method for comparing your custom attributes because the collection view uses this method for some of its operations.

For more information about layout attributes, see *UICollectionViewLayoutAttributes Class Reference*.

Preparing the Layout

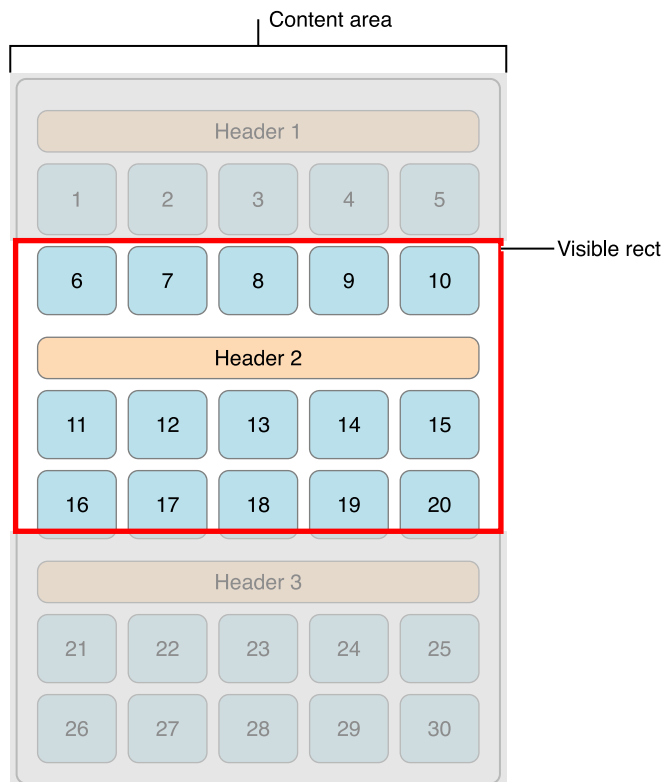
At the beginning of the layout cycle, the layout object calls `prepareLayout` before beginning the layout process. This method is your chance to calculate information that later informs your layout. The `prepareLayout` method is not required to implement a custom layout but is provided as an opportunity to make initial calculations if necessary. After this method is called, your layout must have enough information to calculate the collection view's content size, the next step in the layout process. The information, however, can range from this minimum requirement to creating and storing all the layout attributes objects your layout will use. Use of the `prepareLayout` method is subject to the infrastructure of your app and to what makes sense to compute up front versus what to compute upon request. For an example of what the `prepareLayout` method might look like, see [“Preparing the Layout”](#) (page 58).

Providing Layout Attributes for Items in a Given Rectangle

During the final step of the layout process, the collection view calls your layout object's `layoutAttributesForElementsInRect:` method. The purpose of this method is to provide layout attributes for every cell and every supplementary or decoration view that intersects the specified rectangle. For a large scrollable content area, the collection view may just ask for the attributes of items in the portion of that content area that is currently visible. In Figure 5-2, the currently visible content that your layout object needs to create

attribute objects for is cells 6 through 20 along with the second header view. You must be prepared to provide layout attributes for any portion of your collection view content area. Such attributes might be used to facilitate animations for inserted or deleted items.

Figure 5-2 Laying out only the visible views



Because the `layoutAttributesForElementsInRect:` method is called after your layout object's `prepareLayout` method, you should already have most of the information you need in order to return or create the required attributes. The implementation of your `layoutAttributesForElementsInRect:` method follows these steps:

1. Iterate over the data generated by the `prepareLayout` method to either access cached attributes or create new ones.
2. Check the frame of each item to see whether it intersects the rectangle passed to the `layoutAttributesForElementsInRect:` method.
3. For each intersecting item, add a corresponding `UICollectionViewLayoutAttributes` object to an array.
4. Return the array of layout attributes to the collection view.

Depending on how you manage your layout information, you might create `UICollectionViewLayoutAttributes` objects in your `prepareLayout` method or wait and do it in your `layoutAttributesForElementsInRect:` method. While forming an implementation that matches the needs of your application, keep in mind the benefits of caching layout information. Computing new layout attributes repeatedly for cells is an expensive operation, one that can have noticeably detrimental effects on your app's performance. That said, when the amount of items your collection view manages is large, it may make more sense (for performance) to create the layout attributes when requested. It's simply a matter of figuring out which strategy makes most sense for your app.

Note: Layout objects also need to be able to provide layout attributes on demand for individual items. The collection view might request that information outside of the normal layout process for several reasons, including to create appropriate animations. For more information about providing layout attributes on demand, see [“Providing Layout Attributes On Demand”](#) (page 46).

For a specific example of how one might implement `layoutAttributesForElementsInRect:`, see [“Providing Layout Attributes”](#) (page 62).

Providing Layout Attributes On Demand

The collection view periodically asks your layout object to provide attributes for individual items outside of the formal layout process. For example, the collection view asks for this information when configuring insertion and deletion animations for an item. Your layout object must be prepared to provide the layout attributes for each cell, supplementary view, and decoration view it supports. You do this by overriding the following methods:

- `layoutAttributesForItemAtIndexPath:`
- `layoutAttributesForSupplementaryViewOfKind:atIndexPath:`
- `layoutAttributesForDecorationViewOfKind:atIndexPath:`

Your implementation of these methods should retrieve the current layout attributes for the given cell or view. Every custom layout object is expected to implement the `layoutAttributesForItemAtIndexPath:` method. If your layout does not contain any supplementary views, you do not need to override the `layoutAttributesForSupplementaryViewOfKind:atIndexPath:` method. Similarly, if it does not contain decoration views, you do not need to override the `layoutAttributesForDecorationViewOfKind:atIndexPath:` method. When returning attributes, you should not update the layout attributes. If you need to change the layout information, invalidate the layout object and let it update that data during a subsequent layout cycle.

Connecting Your Custom Layout for Use

There are two ways to link your custom layout to the collection view: programmatically or through storyboards. The collection view links to its layout through a writable property, `collectionViewLayout`. To set the layout to your custom implementation, set your collection view's layout property to an instance of your custom layout object. Listing 5-1 shows the line of code needed.

Listing 5-1 Linking your custom layout

```
self.collectionView.collectionViewLayout = [[MyCustomLayout alloc] init];
```

Otherwise, from your storyboard, open the Document Outline panel and select your collection view (it is listed in the drop-down menu for your controller). With the collection view selected, open the Attributes inspector in the Utilities pane, and underneath the section labeled Collection View change the Layout option from Flow to Custom. The option beneath it changes from Scroll Direction to Class, and you can now select your custom layout class.

Making Your Custom Layouts More Engaging

Providing layout attributes for each cell and view during the layout process is required, but there are other behaviors that can improve the user experience with your custom layout. Implementing these behaviors is optional but recommended.

Elevating Content Through Supplementary Views

Supplementary views are separate from the collection view's cells and have their own set of layout attributes. Like cells, these views are provided by the data source object, but their purpose is to enhance the main content of your app. For example, the `UICollectionViewFlowLayout` uses supplementary views for section headers and footers. Another app could use supplementary views to give each cell its own text label to display information about that cell. Like collection view cells, supplementary views undergo a recycling process to optimize the amount of resources used by the collection view. Therefore, all supplementary views used in your app should be subclassed from the `UICollectionViewReusableView` class.

The steps for adding supplementary views to your layouts are as follows:

1. Register your supplementary view to the collection view's layout object using either the `registerClass:forSupplementaryViewOfKind:withReuseIdentifier:` or `registerNib:forSupplementaryViewOfKind:withReuseIdentifier:` method.

2. In your data source, implement `collectionView:viewForSupplementaryElementOfKind:atIndexPath:`. Because these views are reusable, call `dequeueReusableCellSupplementaryViewOfKind:withReuseIdentifier:forIndexPath:` to dequeue, or create, a new reusable view and set any necessary data before returning it.
3. Create layout attributes objects for your supplementary views just as you do for cells.
4. Include these layout attributes objects in the array of attributes returned by the `layoutAttributesForElementsInRect:` method.
5. Implement the `layoutAttributesForSupplementaryViewOfKind:atIndexPath:` method to return the attributes object for the specified supplementary view whenever queried.

The process for creating the attributes objects for supplementary views in your custom layout is nearly identical to the process for cells, but differs in that a custom layout can have multiple types of supplementary views but is restricted to one type of cell. This is because supplementary views are meant to enhance the main content and are therefore separate from it. There are many ways in which an app's content can be supplemented, and so each of the supplementary view's methods specifies which kind of view is being addressed to distinguish it from the others and allow your layout to compute its attributes correctly based on its type. When registering a supplementary view for use, the string you provide is used by the layout object to distinguish the view from others. For an example of incorporating supplementary views into your custom layout, see [“Incorporating Supplementary Views”](#) (page 66).

Including Decoration Views in Your Custom Layouts

Decoration views are visual adornments that enhance the appearance of your collection view layouts. Unlike cells and supplementary views, decoration views provide visual content only and are thus independent of the data source. You can use them to provide custom backgrounds, fill in the spaces around cells, or even obscure cells if you want. Decoration views are defined and managed solely by the layout object and do not interact with the collection view's data source object.

To add decoration views to your layouts, do the following:

1. Register your decoration view with the layout object using the `registerClass:forDecorationViewOfKind:` or `registerNib:forDecorationViewOfKind:` method. Although this approach is similar to registering cells and supplementary views, remember that registering decoration views occurs within the layout object, not within the data source.
2. In your layout object's `layoutAttributesForElementsInRect:` method, create attributes for your decoration views just as you do for your cells and supplementary views.
3. Implement the `layoutAttributesForDecorationViewOfKind:atIndexPath:` method in your layout object and return the attributes for your decoration views when asked.

4. Optionally, implement the `initialLayoutAttributesForAppearingDecorationElementOfKind:atIndexPath:` and `finalLayoutAttributesForDisappearingDecorationElementOfKind:atIndexPath:` methods to handle animations for the appearance and disappearance of your decoration views. For more information, see [“Making Insertion and Deletion Animations More Interesting”](#) (page 49)

The creation process for decoration views is different from the process for cells and supplementary views. Registering a class or nib file is all you have to do to ensure that decoration views are created when they are needed. Because they are purely visual, decoration views are not expected to need any configuration beyond what is already done in the provided nib file or by the object’s `initWithFrame:` method. For this reason, when a decoration view is needed, the collection view creates it for you and applies the attributes provided by the layout object. Any decoration views should still be a subclass of `UICollectionViewReusableView` because the layout object employs a recycling mechanism for its decoration views.

Note: When creating the attributes for your decoration views, don’t forget to take into account the `zIndex` property. You can use the `zIndex` attribute to layer your decoration views behind (or, if you prefer, in front of) the displayed cells and supplementary views.

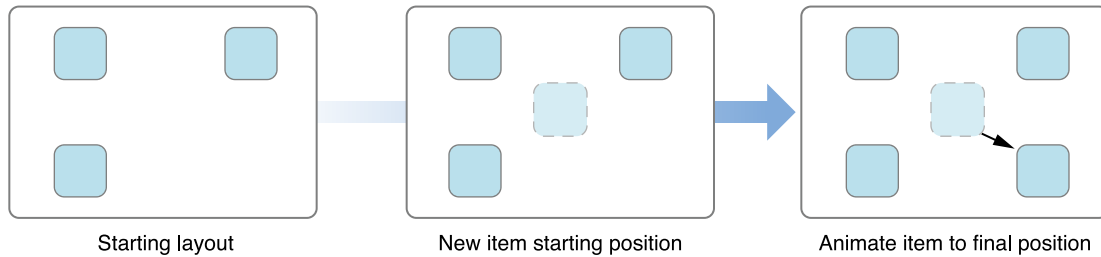
Making Insertion and Deletion Animations More Interesting

Inserting and deleting cells and views poses an interesting challenge during layout. Inserting a cell can cause the layout for other cells and views to change. Even though the layout object knows how to animate existing cells and views from their current locations to new locations, it has no current location for the cell being inserted. Rather than insert the new cell without animations, the collection view asks the layout object to provide a set of initial attributes to use for the animation. Similarly, when a cell is deleted, the collection view asks the layout object to provide a set of final attributes to use for the endpoint of any animations.

To understand how initial attributes work, it helps to see an example. The starting layout (Figure 5-3) shows a collection view that initially contains only three cells. When a new cell is inserted, the collection view asks the layout object to provide initial attributes for the cell being inserted. In this case, the layout object sets the initial

position of the cell to the middle of the collection view and sets its alpha value to 0 to hide it. During the animations, this new cell appears to fade in and move from the center of the collection view to its final position in the lower-right corner.

Figure 5-3 Specifying the initial attributes for an item appearing onscreen



Listing 5-2 shows the code you might use to specify the initial attributes for the inserted cell from Figure 5-3. This method sets the position of the cell to the center of the collection view and makes it transparent. The layout object would then provide the final position and alpha for the cell as part of the normal layout process.

Listing 5-2 Specifying the initial attributes for an inserted cell

```
- (UICollectionViewLayoutAttributes
*)initialLayoutAttributesForAppearingItemAtIndexPath:(NSIndexPath *)itemIndexPath
{
    UICollectionViewLayoutAttributes* attributes = [self
layoutAttributesForItemAtIndexPath:itemIndexPath];
    attributes.alpha = 0.0;

    CGSize size = [self collectionView].frame.size;
    attributes.center = CGPointMake(size.width / 2.0, size.height / 2.0);
    return attributes;
}
```

Note: Listing 5-2 would animate all cells when one is inserted, so the three cells that were already present before the fourth was inserted would also pop out from the center of the collection view. To animate only the cell being inserted, check to see if the index path of the item matches the index path of an item passed to the `prepareForCollectionViewUpdates:` method and only perform the animation if a match is found. Otherwise, return the attributes returned by calling the super method of `initialLayoutAttributesForAppearingItemAtIndexPath:`.

The process for handling deletions is identical to the process for insertions except that you specify the final attributes instead of the initial attributes. From the previous example, if you used the same attributes that you used when inserting the cell, deleting the cell would cause it to fade out while moving to the center of the collection view. There are six methods available to you within the `UICollectionViewLayout` class—two separate methods (for initial and final attributes) for items, supplementary views, and decoration views.

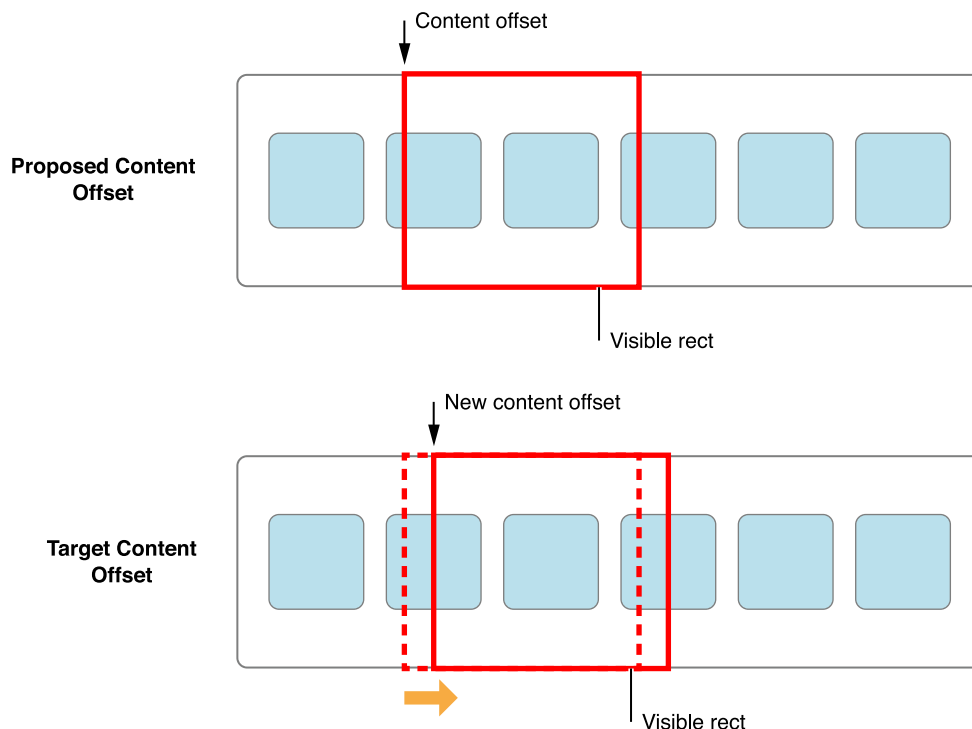
Improving the Scrolling Experience of Your Layout

Your custom layout object can influence the scrolling behavior of the collection view to create a better user experience. When scrolling-related touch events end, the scroll view determines the final resting place of the scrolling content based on the current speed and deceleration rate in effect. When the collection view knows that location, it asks its layout object if the location should be modified by calling its `targetContentOffsetForProposedContentOffset:withScrollingVelocity:` method. Because it calls this method while the underlying content is still moving, your custom layout can affect the final resting point of the scrolling content.

Figure 5-4 demonstrates how you might use your layout object to change the scrolling behavior of the collection view. Suppose the collection view offset starts at (0, 0) and the user swipes left. The collection view computes where the scrolling would naturally stop and provides that value as the “proposed” content offset value. Your layout object might change the proposed value to ensure that when scrolling stops, an item is centered

precisely in the visible bounds of the collection view. This new value becomes the target content offset and is what you return from your `targetContentOffsetForProposedContentOffset:withScrollingVelocity:` method.

Figure 5-4 Changing the proposed content offset to a more appropriate value



Tips for Implementing Your Custom Layouts

Here are some tips and suggestions for implementing your custom layout objects:

- Consider using the `prepareLayout` method to create and store the `UICollectionViewLayoutAttributes` objects you need for later. The collection view is going to ask for layout attribute objects at some point, so in some cases it makes sense to create and store them up front. This is especially true if you have a relatively small number of items (several hundred) or the actual layout attributes for those items change infrequently.

If your layout needs to manage thousands of items, though, you need to weigh the benefits of caching versus recomputing. For variable-size items whose layout changes infrequently, caching generally eliminates the need to recompute complex layout information regularly. For large numbers of fixed-size items, it may just be simpler to compute attributes on demand. And for items whose attributes change frequently, you might be recomputing all the time anyway so caching may just take up extra space in memory.

- Avoid subclassing `UICollectionView`. The collection view has little or no appearance of its own. Instead, it pulls all of its views from your data source object and all of the layout-related information from the layout object. If you are trying to lay out items in three dimensions, the proper way to do it is to implement a custom layout that sets the 3D transform of each cell and view appropriately.
- Never call the `visibleCells` method of `UICollectionView` from the `layoutAttributesForElementsInRect:` method of your custom layout object. The collection view knows nothing of where items are positioned, other than what the layout object tells it. So asking for the visible cells is just going to forward the request to your layout object.

Your layout object should always know the location of items in the content area and should be able to return the attributes of those items at any time. In most cases, it should do this on its own. In a limited number of cases, the layout object might rely on information in the data source to position items. For example, a layout that displays items on a map might retrieve the map location of each item from the data source.

Custom Layouts: A Worked Example

Creating a custom collection view layout is simple with straightforward requirements, but the implementation details of the process may vary. Your layout must produce layout attributes objects for every view your collection view contains. The order in which these attributes are created depend on the nature of your application. With a collection view that houses thousands of items, computing and caching layout attributes upfront is a time-consuming process, so it makes more sense to create attributes only when requested for a specific item. For an application with fewer items, computing layout information once and caching it to reference whenever attributes are requested can save your application a lot of unneeded recalculation. The worked example in this chapter belongs in the second category.

Keep in mind that the provided sample code is by no means the definitive way to create a custom layout. Before you begin creating your custom layout, take time to devise an implementation structure that makes the most sense for your app to get the best performance. For a conceptual overview of the layout customizing process, see [“Creating Custom Layouts”](#) (page 40).

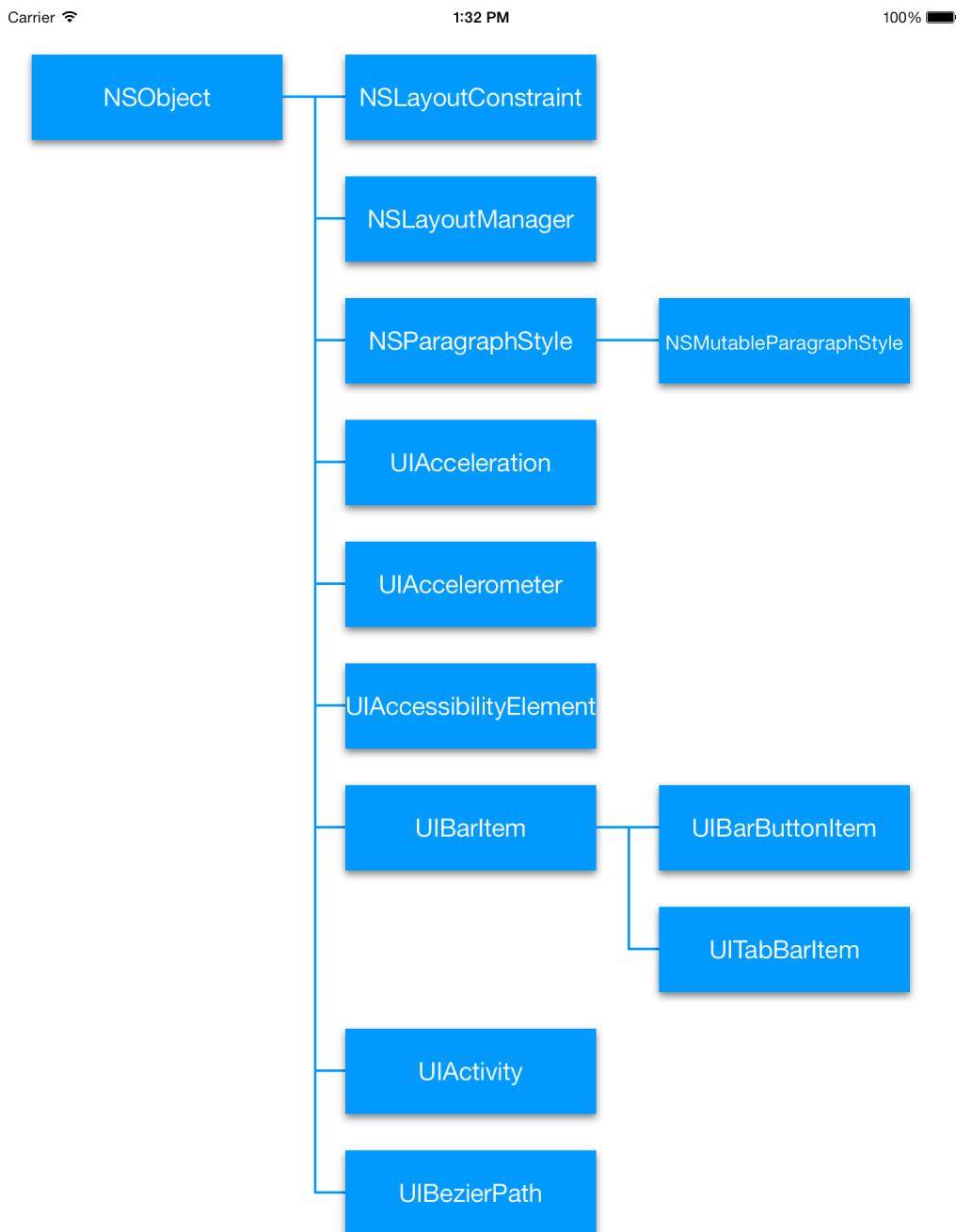
Because this chapter presents the custom layout implementation in a specific order, follow along the example from top to bottom with a specific implementation goal in mind. This chapter focuses on creating a custom layout, not on implementing a complete app. Therefore, implementations of the views and controllers used to create the final product are not provided. The layout uses custom collection view cells as its cells and a custom view for creating the lines connecting cells to one another. Creating custom cells and views for collection views, as well as the requirements for using a collection view, are covered in previous chapters. To review this information, see [“Collection View Basics”](#) (page 9) and [“Designing Your Data Source and Delegate”](#) (page 16).

The Concept

The purpose of this worked example is to implement a custom layout for displaying a hierarchical tree of information such as the diagram seen in Figure 6-1. The example provides snippets of code followed by an explanation of the code, along with the point in the customization process you have reached. Each section of the collection view constitutes one level of depth into the tree: Section 0 contains only the NSObject cell. Section 1 contains all of the children cell's of NSObject. Section 2 contains all of the children cells of those children, and so on. Each of the cells is a custom cell, with a label for the associated class name, and the

connections between cells are supplementary views. Because the connector view class must determine how many connections to draw, it needs access to the data in our data source. It therefore makes sense to implement these connections as supplementary views and not decoration views.

Figure 6-1 Class hierarchy



Initialization

The first step in creating a custom layout is to subclass the `UICollectionViewLayout` class. Doing so provides you with the foundations necessary to build a custom layout.

For this example, a custom protocol is necessary to inform the layout's spacing between certain items. If the attributes of specific items require extra information from the data source, it's best to implement a protocol for a custom layout rather than to initiate a direct connection to the data source. Your resulting layout is more robust and reusable; it won't be attached to a specific data source but will instead respond to any object that implements its protocol.

Listing 6-1 shows the necessary code for the custom layout's header file. Now, any class that implements the *MyCustomProtocol* protocol can utilize the custom layout, and the layout can query that class for the information it needs.

Listing 6-1 Connecting to the custom protocol

```
@interface MyCustomLayout : UICollectionViewLayout
@property (nonatomic, weak) id<MyCustomProtocol> customDataSource;
@end
```

Next, because the number of items the collection view will manage is relatively low, the custom layout makes use of a caching system to store the layout attributes it generates when preparing the layout and then retrieves these stored values whenever the collection view asks for them. Listing 6-2 shows the three private properties our layout will need to maintain and the `init` method. The `layoutInformation` dictionary houses all of the layout attributes for all types of views within our collection view, and the `maxNumRows` property keeps track of how many rows are needed to populate the tallest column of our tree. The `insets` object controls spacing between cells and is used in setting the frames for views and the content size. The values for the first two properties are set while preparing the layout, but the `insets` object should be set using the `init` method. In this case, `INSET_TOP`, `INSET_LEFT`, `INSET_BOTTOM`, and `INSET_RIGHT` refer to constants you define for each parameter.

Listing 6-2 Initializing variables

```
@interface MyCustomLayout()

@property (nonatomic) NSDictionary *layoutInformation;
@property (nonatomic) NSInteger maxNumRows;
@property (nonatomic) UIEdgeInsets insets;
```



```
@end

-(id)init {
    if(self = [super init]) {
        self.insets = UIEdgeInsetsMake(INSET_TOP, INSET_LEFT, INSET_BOTTOM,
        INSET_RIGHT);
    }
    return self;
}
```

The last step for this custom layout is to create custom layout attributes. Although this step is not always necessary, in this case, as cells are being placed, the code needs access to the index paths of the current cell's children so that it can adjust the children cell's frames to match that of their parent. So subclassing `UICollectionViewLayoutAttributes` to store an array of the cell's children provides that information. You subclass `UICollectionViewLayoutAttributes`, and in the header file add the following code:

```
@property (nonatomic) NSArray *children;
```

As explained in the `UICollectionViewLayoutAttributes` class reference, subclassing layout attributes requires that you override the inherited `isEqual:` method in iOS 7 and later. For more information on why this is, see *UICollectionViewLayoutAttributes Class Reference*.

The implementation for the `isEqual:` method in this case is simple because there's only one field to compare—the contents of the children array. If the arrays of both layout attributes objects match, then they must be equal because children can belong to only one class. Listing 6-3 shows the implementation for the `isEqual:` method.

Listing 6-3 Fulfilling requirements for subclassing layout attributes

```
-(BOOL)isEqual:(id)object {
    MyCustomAttributes *otherAttributes = (MyCustomAttributes *)object;
    if ([self.children isEqualToArray:otherAttributes.children]) {
        return [super isEqual:object];
    }
    return NO;
}
```

Remember to include the header file for the custom layout attributes in the custom layout file.

At this point in the process, you are ready to start implementing the main part of the custom layout with the foundations you have laid.

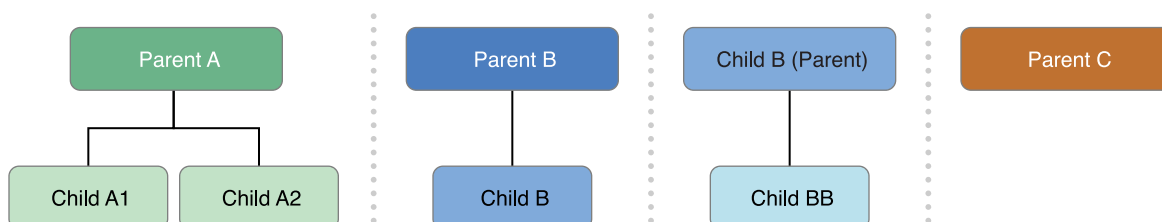
Preparing the Layout

Now that all of the necessary components have been initialized, you can prepare the layout. The collection view first calls the `prepareLayout` method during the layout process. In this example, the `prepareLayout` method is used to instantiate all of the layout attributes objects for every view in the collection view and then cache those attributes in our `layoutInformation` dictionary for later use. For more information on the `prepareLayout` method, see [“Preparing the Layout”](#) (page 44).

Creating the Layout Attributes

The example implementation of the `prepareLayout` method is split into two parts. Figure 6-2 shows the goal for the first half of the method. The code iterates over every cell, and if that cell has children, relates those children to the parent cell. As you can see in the figure, this process is done for every cell, including the children cells of other parent cells.

Figure 6-2 Connecting parent and child index paths



Listing 6-4 shows the first half of the `prepareLayout` method’s implementation. Both mutable dictionaries initialized at the beginning of the code form the basis of the caching mechanism. The first, `layoutInformation`, is the local equivalent of the `layoutInformation` property. Creating a local mutable copy allows the instance variable to be immutable, which makes sense in the custom layout’s implementation because layout attributes should not be modified after the `prepareLayout` method finishes running. The code then iterates over each section in increasing order and then over each item within each section to create attributes for every cell. The custom method `attributesWithChildrenForIndexPath:` returns an instance of the custom layout attributes, with the `children` property populated with the index paths of the children for the item at the current index path. The attributes object is then stored within the local `cellInformation` dictionary with its index path as the key. This initial pass over all of the items allows the code to set the children for each item before setting the item’s frame.

Listing 6-4 Creating layout attributes

```
- (void)prepareLayout {
    NSMutableDictionary *layoutInformation = [NSMutableDictionary dictionary];
    NSMutableDictionary *cellInformation = [NSMutableDictionary dictionary];
    NSIndexPath *indexPath;
    NSInteger numSections = [self.collectionView numberOfSections];
    for(NSInteger section = 0; section < numSections; section++){
        NSInteger numItems = [self.collectionView numberOfItemsInSection:section];
        for(NSInteger item = 0; item < numItems; item++){
            indexPath = [NSIndexPath indexPathForItem:item inSection:section];
            MyCustomAttributes *attributes =
                [self attributesWithChildrenAtIndexPath:indexPath];
            [cellInformation setObject:attributes forKey:indexPath];
        }
    }
    //end of first section
}
```

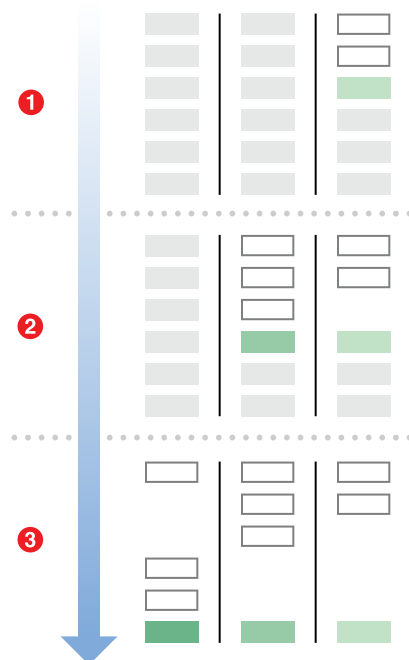
Storing the Layout Attributes

Figure 6-3 depicts the process that occurs in the second half of the `prepareLayout` method in which the tree hierarchy is built from the last row to the first. This approach may at first seem peculiar, but it is a clever way of eliminating the complexity involved with adjusting children cell's frames. Because the frames of children cells need to match up with those of their parent, and because the amount of space between cells on a row-to-row basis is dependent upon how many children a cell has (including how many children each child cell has and so on), you want to set the child's frame before setting the parent's. In this way, the child cell and all of its children cells can be adjusted to match their overall parent's cell.

In step 1, the cells of the last column have been placed in sequential order. In step 2, the layout is determining the frames for the second column. In this column, the cells can be laid out sequentially since no cell has more than one child. However, the green cell's frame must be adjusted to match that of its parent cell, so it is shifted down one space. In the final step, the cells for the first column are being placed. The first three cells of the second column are the children of the first cell in the first column, so the cell's following the first cell in the first column are shifted down. In this case, it is not actually necessary to do so since the two cell's following

the first have no children of their own, but the layout object is not smart enough to know this. Rather, it always adjusts the space in case any cell following one with children has children of its own. As well, the green cells have now both shifted down to match that of their parent.

Figure 6-3 The framing process



Listing 6-5 shows the second half of the `prepareLayout` method, in which the frames for each item are set. The commented numbers following some lines of code correspond to the numbered explanations found after the code.

Listing 6-5 Storing layout attributes

```
//continuation of prepareLayout implementation
for(NSInteger section = numSections - 1; section >= 0; section--){
    NSInteger numItems = [self.collectionView numberOfItemsInSection:section];
    NSInteger totalHeight = 0;
    for(NSInteger item = 0; item < numItems; item++){
        indexPath = [NSIndexPath indexPathForItem:item inSection:section];
        MyCustomAttributes *attributes = [cellInfo objectForKey:indexPath];
// 1
        attributes.frame = [self frameForCellAtIndexPath:indexPath
                        withHeight:totalHeight];
    }
}
```

```
        [self adjustFramesOfChildrenAndConnectorsForClassAtIndexPath:indexPath]; // 2

        cellInfo[indexPath] = attributes;
        totalHeight += [self.customDataSource
                        numRowsForClassAndChildrenAtIndexPath:indexPath]; // 3
    }
    if(section == 0){
        self.maxNumRows = totalHeight; // 4
    }
}
[layoutInformation setObject:cellInformation forKey:@"MyCellKind"]; // 5
self.layoutInformation = layoutInformation
}
```

In Listing 6-5, the code traverses the sections in descending order, building the tree from the back to the front. The `totalHeight` variable tracks how many rows down the current item needs to be. This implementation does not track spacing smartly, simply leaving empty space below cells with children so that two cell's children never overlap, and the `totalHeight` variable helps accomplish this. The code accomplishes this in the following order:

1. The layout attributes created in our first pass over the data are retrieved from the local dictionary before the cell's frame is set.
2. The custom `adjustFramesOfChildrenAndConnectorsForClassAtIndexPath:` method recursively adjusts the frames of all the cell's children and grandchildren and so on to match the cell's frame.
3. After putting the adjusted attributes back in the dictionary, the `totalHeight` variable is adjusted to reflect where the next item's frame needs to be. This is where the code takes advantage of the custom protocol. Whatever object implements that protocol needs to implement the `numRowsForClassAndChildrenAtIndexPath:` method, which returns how many rows each class needs to occupy given how many children it has.
4. The `maxNumRows` property (later needed to set the content size) is set to section 0's total height. The column with the longest height is always section 0, which has a height adjusted for all of the children in the tree, because this implementation doesn't include smart space adjusting.
5. The method concludes by inserting the dictionary with all of the cell attributes into the local `layoutInformation` dictionary with a unique string identifier as its key.

The string identifier used to insert the dictionary in the final step is used throughout the rest of the custom layout to retrieve the correct attributes for the cells. It becomes even more important when supplementary views come into play further along in the example.

Providing the Content Size

In preparing the layout, the code sets the value of `maxNumRows` to be the number of rows in the largest section in the layout. This information can be leveraged to set the appropriate content size, which is the next step in the layout process. Listing 6-6 shows the implementation for `collectionViewContentSize`. It relies on the constants `ITEM_WIDTH` and `ITEM_HEIGHT`, which are presumably global to the application (for instance, they are needed in the custom cell implementation to size the cell label correctly).

Listing 6-6 Sizing the content area

```
- (CGSize)collectionViewContentSize {  
    CGFloat width = self.collectionView.numberOfSections * (ITEM_WIDTH +  
self.insets.left + self.insets.right);  
    CGFloat height = self.maxNumRows * (ITEM_HEIGHT + _insets.top + _insets.bottom);  
    return CGSizeMake(width, height);  
}
```

Providing Layout Attributes

With all of the layout attributes objects initialized and cached, the code is fully prepared to provide all of the layout information requested in the `layoutAttributesForElementsInRect:` method. This method is the second step in the layout process, and unlike the `prepareLayout` method, it is required. The method provides a rectangle and expects an array of layout attributes objects for any views contained within the provided rectangle. In some cases, collection views that house thousands of items might wait until this method is called to initialize layout attributes objects for only the elements contained within the provided rectangle, but this implementation relies on caching instead. Therefore, the `layoutAttributesForElementsInRect:` method simply requires looping over all of the stored attributes and gathering them into a single array returned to the caller.

Listing 6-7 shows the implementation of the `layoutAttributesForElementsInRect:` method. The code traverses all of the subdictionaries that contain layout attributes objects for specific types of views within the main dictionary `_layoutInformation`. If the attributes examined within the subdictionary are contained within the given rectangle, they're added to an array storing all of the attributes within that rectangle, which is returned after all of the stored attributes have been checked.

Listing 6-7 Collecting and processing stored attributes

```
- (NSArray*)layoutAttributesForElementsInRect:(CGRect)rect {
    NSMutableArray *myAttributes [NSMutableArray
arrayWithCapacity:self.layoutInformation.count];
    for(NSString *key in self.layoutInformation){
        NSDictionary *attributesDict = [self.layoutInformation objectForKey:key];
        for(NSIndexPath *key in attributesDict){
            UICollectionViewLayoutAttributes *attributes =
            [attributesDict objectForKey:key];
            if(CGRectIntersectsRect(rect, attributes.frame)){
                [attributes addObject:attributes];
            }
        }
    }
    return myAttributes;
}
```

Note: The implementation for `layoutAttributesForElementsInRect:` never references whether or not the view for a given attributes is visible. Remember that the rectangle provided by this method is not necessarily going to be the visible rectangle and that, no matter what your implementation is, it should never assume the attributes it is returning are for visible views. For a lengthier discussion about the `layoutAttributesForElementsInRect:` method, see [“Providing Layout Attributes for Items in a Given Rectangle”](#) (page 44).

Providing Individual Attributes When Requested

As discussed in the section [“Providing Layout Attributes On Demand”](#) (page 46), the layout object must be prepared to return layout attributes for any singular item of any kind of view within your collection view at any time once the layout process is complete. There are methods for all three kinds of views—cells,

supplementary views and decoration views—but the app currently uses cells exclusively, so the only method that requires an implementation for the time being is the `layoutAttributesForItemAtIndexPath:` method.

Listing 6-8 shows the implementation for this method. It taps into the stored dictionary for cells, and within that subdictionary, it returns the attributes object stored with the specified index path as its key.

Listing 6-8 Providing attributes for specific items

```
- (UICollectionViewLayoutAttributes *)layoutAttributesForItemAtIndexPath: (NSIndexPath *)indexPath {  
    return self.layoutInfo[@"MyCellKind"][indexPath];  
}
```


Figure 6-4 shows what the layout looks like at this point in the code. All of the cells have been placed and adjusted correctly to match their parents, but the lines connecting them haven't been drawn yet.

Figure 6-4 The layout so far



Incorporating Supplementary Views

In its current state, the app displays all of its cells correctly in a hierarchical sense, but because there are no lines connecting parents to their children, the class diagram is difficult to interpret. To draw lines connecting class cells to their children, this app implementation relies on a custom view that can be incorporated into the layout as a supplementary view. For more information on designing supplementary views, see [“Elevating Content Through Supplementary Views”](#) (page 47).

Listing 6-9 shows the lines of code that could be incorporated into the implementation of `prepareLayout` to include supplementary views. The minor difference between creating attributes objects for cells and for supplementary views is that the method for supplementary views requires a string identifier to tell what kind of supplementary view the attributes object is for. This is because a custom layout can have multiple different types of supplementary views, whereas each layout can have only one type of cell.

Listing 6-9 Creating attributes objects for supplementary views

```
// create another dictionary to specifically house the attributes for the
supplementary view
NSMutableDictionary *supplementaryInfo = [NSMutableDictionary dictionary];

...

// within the initial pass over the data, create a set of attributes for the
supplementary views as well
UICollectionViewLayoutAttributes *supplementaryAttributes =
[UICollectionViewLayoutAttributes
layoutAttributesForSupplementaryViewOfKind:@"ConnectionViewKind"
withIndexPath:indexPath];

[supplementaryInfo setObject: supplementaryAttributes forKey:indexPath];

...

// in the second pass over the data, set the frame for the supplementary views
just as you did for the cells
UICollectionViewLayoutAttributes *supplementaryAttributes = [supplementaryInfo
objectForKey:indexPath];

supplementaryAttributes.frame = [self
frameForSupplementaryViewOfKind:@"ConnectionViewKind" AtIndexPath:indexPath];

[supplementaryInfo setObject:supplementaryAttributes forKey:indexPath];

...

// before setting the instance version of _layoutInformation, insert the local
supplementaryInfo dictionary into the local layoutInformation dictionary
[layoutInformation setObject:supplementaryInfo forKey:@"ConnectionViewKind"];
```

Because the code for supplementary views is similar to that for cells, incorporating this code into the `prepareLayout` method is simple. The code employs the same caching mechanism for supplementary views as it does for the cells, using another dictionary specifically for the *CollectionViewKind* supplementary view. If you were going to add more than one kind of supplementary view, you would create another dictionary for that kind of view and add these lines of code for that kind of view, too. But in this case, the layout requires only one kind of supplementary view. As in the code for initializing the cell layout attributes, this implementation employs the custom `frameForSupplementaryViewOfKind:AtIndexPath:` method for determining a supplementary view's frame based on what kind of view it is. Remember that the custom `adjustFramesOfChildrenAndConnectorsForClassAtIndexPath:` shown in the implementation of the `prepareLayout` method needs to incorporate the adjustment of any supplementary views relevant to the class hierarchy layout.

In the case of the example code, nothing needs to be modified in the `layoutAttributesForElementsInRect:` implementation because it is designed to loop over all the attributes stored in the main dictionary. As long as the supplementary view attributes are added to the main dictionary, the provided implementation of `layoutAttributesForElementsInRect:` works as expected.

Last, as was the case for cells, the collection view may request supplementary view attributes for specific views at any time. As such, an implementation of `layoutAttributesForSupplementaryElementOfKind:atIndexPath:` is required.

Listing 6-10 shows the implementation for the method, which is nearly identical to that of `layoutAttributesForItemAtIndexPath:`. As an exception, using the provided `kind` string rather than hardcoding a type of view into the return value allows you to use multiple supplementary views in your custom layout.

Listing 6-10 Providing supplementary view attributes on demand

```
- (UICollectionViewLayoutAttributes *)
layoutAttributesForSupplementaryViewOfKind:(NSString *)kind atIndexPath:(NSIndexPath
*)indexPath {
    return self.layoutInfo[kind][indexPath];
}
```

Recap

By including supplementary views, you now have a layout object that can adequately reproduce a class hierarchy diagram. In a final implementation, you probably want to incorporate adjustments into your custom layout to conserve space. This example explores what a real, base implementation of a custom collection view layout might look like. Collection views are incredibly robust, and provide so much more capability than seen here.

Highlighting and selecting (or even animating) cells when moved, inserted, or deleted are all easy enhancements that can be incorporated into your app. To take your custom layout to the next level, take a look at the last few sections of [“Creating Custom Layouts”](#) (page 40).

Document Revision History

This table describes the changes to *Collection View Programming Guide for iOS*.

Date	Notes
2013-09-18	Made major changes to custom layout content and fixed technical errors. Added a new chapter providing an example of building a custom layout. Updated for features new in iOS 7.
2012-09-19	New document that describes how to use collection views in iOS apps.



Apple Inc.
Copyright © 2013 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.