Collection View Programming Guide for iOS

Developer

Contents

Figures, Tables, and Listings

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. 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, whose job is to specify the placement and visual attributes for your views. 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 about 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, the layout object, and your custom objects is crucial for using collection views in your app.

Relevant chapters: "Collection View Basics" (page 8), "Designing Your Data Source and Delegate" (page 15)

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 27)

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 the collection view.

Relevant chapter: "Incorporating Gesture Support" (page 34)

Custom Layouts Let You Go Beyond Grids

The basic layout object can be subclassed to implement custom layouts for your app. Designing a custom layout does not require a large amount of code in most cases. However, it helps to understand how layouts work so that you can design your layout objects to be efficient.

Relevant chapter: "Creating Custom Layouts" (page 38)

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

For a guided overview of collection views, see the following WWDC videos:

- *WWDC 2012: Introducing Collection Views*
- *WWDC 2012: Advanced Collection Views and Building Custom Layouts*

Collection views are somewhat related to table views, in that both present ordered data to the user. 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.

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. Your app is strictly responsible for managing the data to be presented but many different objects manage the visual presentation. 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 in situations where 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	`UICollectionView` `UICollectionView–Controller`	A `UICollectionView` object defines the visible area for your collection view's content. This class descends from UIScrollView 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. A `UICollectionViewController` object provides view controller–level management support for a collection view. Its use is optional.

Purpose	Classes/Protocols	Description
Content management	`UICollectionView–DataSource` protocol `UICollectionView–Delegate` protocol	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 `UICollectionViewDataSource` protocol. 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. For information about how to implement the data source and delegate objects, see "Designing Your Data Source and Delegate" (page 15).
Presentation	`UICollectionReusable–View` `UICollectionViewCell`	All views displayed in a collection view must be instances of the `UICollectionReusableView` 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. A `UICollectionViewCell` object is a specific type of reusable view that you use for your main data items.

Purpose	Classes/Protocols	Description

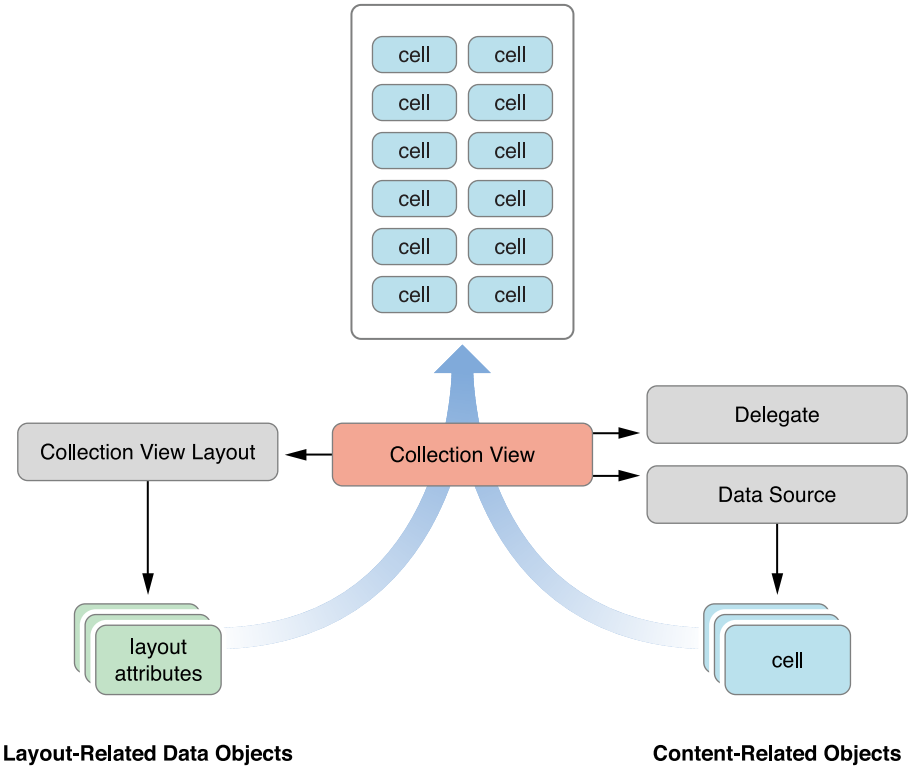
Layout	`UICollectionView-Layout`
	`UICollectionView-LayoutAttributes`
	`UICollectionView-UpdateItem`
		Subclasses of `UICollectionViewLayout` are referred to as **layout objects** and are responsible for defining the location, size, and visual attributes of the cells and reusable views inside a collection view.
		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.
		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.
		For more information about the layout object, see "The Layout Object Controls the Visual Presentation" (page 12).

Flow layout	`UICollectionView-FlowLayout` `UICollectionView-DelegateFlowLayout` protocol	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.

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, the first thing you do is add a `UICollectionView` object and to your storyboard or nib file. After adding that object, you can begin to configure any related objects, such as the data source or delegate. But all configurations are centered around the collection view itself. You can think of the collection view as the central hub, from which all other objects emanate. 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 the collection 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 facilitiate this recycling and reuse, all views displayed by the collection view must descend from the `UICollectionReusableView` 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, although some custom code must be written to actually apply a highlight to a cell.
- **Supplementary views** display information about a section. Like cells, supplementary views are data driven. However, 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 do not impose a specific style on the cells and supplementary views provided by your data source. The basic reusable view classes are blank canvases for you to modify. You can use them to build small view hierarchies, display images, or even draw content dynamically if you want.

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 dequeueing 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 18).

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.

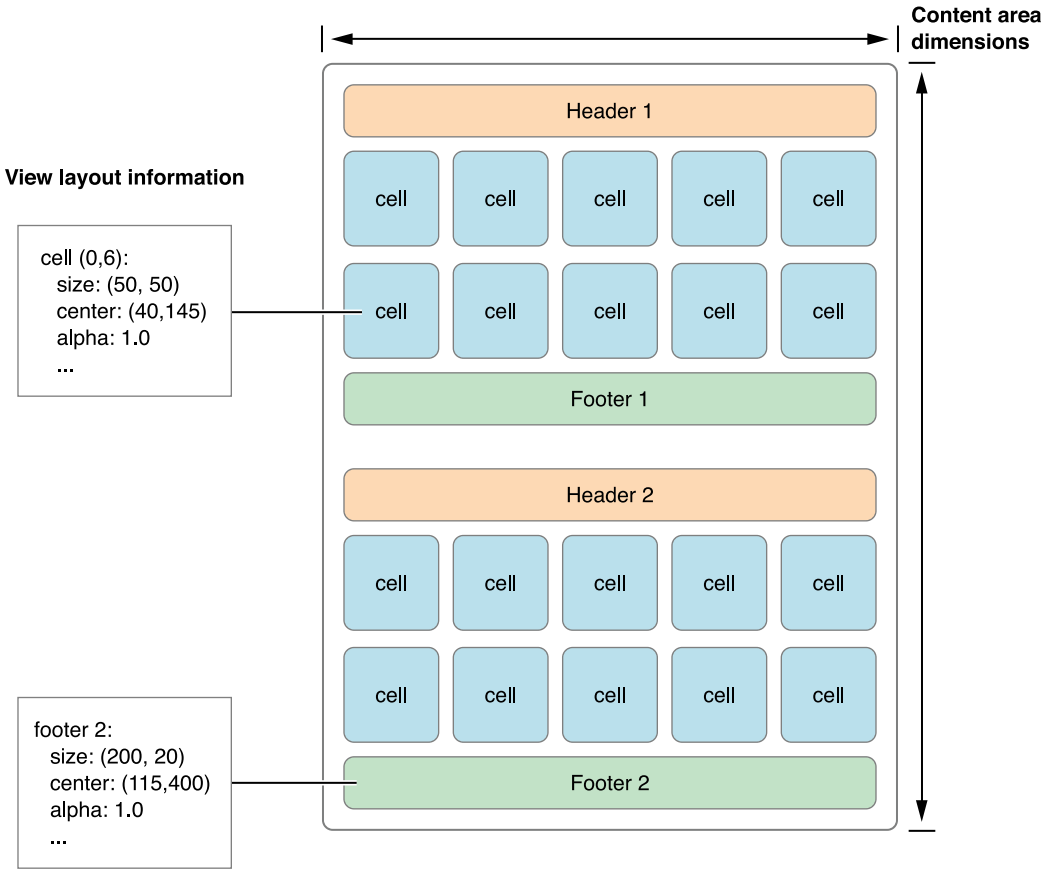
The layout process used by collection views is related to, but distinct from, the layout process used by the rest of your app's views. In other words, do not confuse what a layout object does with the `layoutSubviews` method used to reposition child views inside a parent view. A layout object never touches 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 really 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, it 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 on the layout object or 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 whether it is visibly above or below other views. These attributes let layout objects create more interesting layouts. For example, you could create stacks of cells by placing the views on top of one another and changing their z-ordering or 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 38).

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 knows how to create these animations because it knows 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 redraw its contents. 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. However, the act of invalidating the layout causes the layout object to move items explicitly. 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 providing it with content to display. The data source object is an object that your app provides. 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 the total number of items and the 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. The main job of the delegate is to manage cell highlighting and selection. However, the delegate 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:

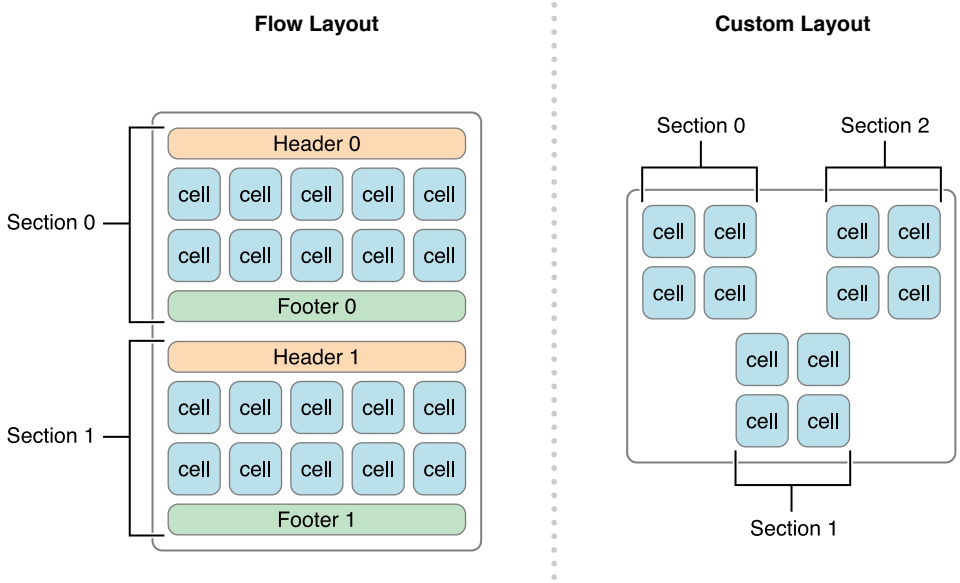
- How many sections does the collection view contain?
- For a given section, how many items are in that section?
- For a given section or item, what views should be used to display the corresponding content?

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 the 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 views, the index path contains whichever values were provided by the layout object.

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 differently by different layout objects



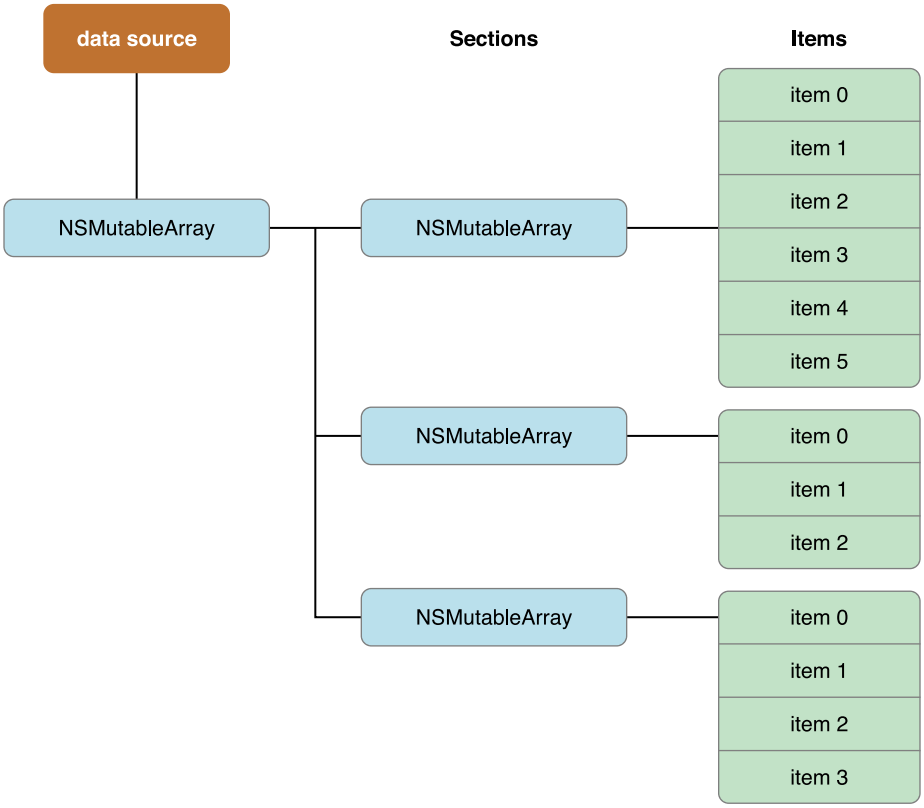
Designing Your Data Objects

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 when it needs to know how many objects there are in total and when it needs views for elements that are currently onscreen. However, if the layout object relies on data from your data objects, performance could be severely impacted when the data source contains thousands of objects. For this reason, you should avoid implementing custom layout objects that rely on data from your data source. Instead, you should design your layout objects to lay out content independently from your data 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.

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 17), 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 knows nothing about any of 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 perform the layout operation. At some point, the collection view asks the layout object to provide the list of elements in a specific rectangle (often the visible rectangle but not necessarily so). 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, you 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 just 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 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.

- **To 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.

- **To 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.
- **To 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. As a result, it defines the string constants `UICollectionElementKindSectionHeader` and `UICollectionElementKindSectionFooter` to identify these two types of views. 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 38).

Registration is a one-time event that must take place before you attempt to dequeue any cells or views. Once registered, you can dequeue as many cells or views as needed without reregistering them. It is not recommended that you change the registration information after dequeueing 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 dequeueing 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 you get back from the dequeueing 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 dequeueing 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. However, an item that was not created from scratch, but was instead retrieved from a reuse queue, may already contain data from a previous usage. In that case, the 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 and use it to reset properties to default values and perform any additional cleanup.

After dequeueing the view, all your data source has to do is configure 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 apply that object's data to the view. After configuring 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 dequeueing 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;
}
```

Inserting, Deleting, and Moving Sections and Items

To insert, delete, or move a single section or item, you must 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, although the animations themselves are performed by the system on a background thread.

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

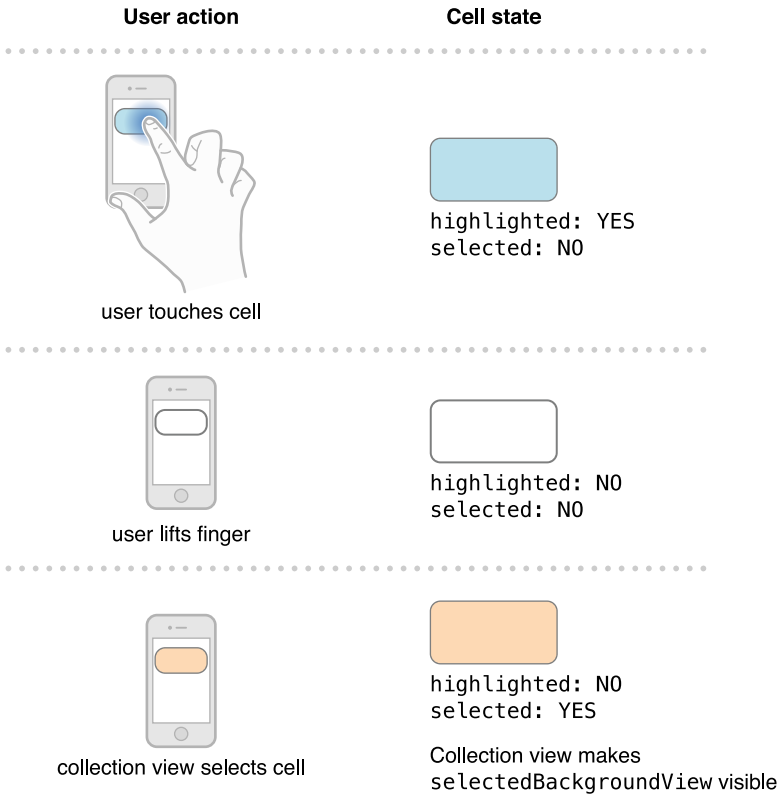
Collection views support single-item selection by default and can be configured to support multiple-item selection or have selections disabled altogether. The collection view detects taps inside its bounds and highlights or selects the corresponding cell accordingly. For the most part, the collection 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.

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`. Changing the selected state

does cause the collection view to display 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



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 2-4 shows a simple way of changing the highlight using the content view's background color.

Listing 2-4 Applying a temporary highlight to a cell

```
- (void)collectionView:(UICollectionView *)colView
didHighlightItemAtIndexPath:(NSIndexPath *)indexPath {

    UICollectionViewCell* cell = [colView cellForItemAtIndexPath:indexPath];

    cell.contentView.backgroundColor = [UIColor blueColor];

}


- (void)collectionView:(UICollectionView *)colView
didUnhighlightItemAtIndexPath:(NSIndexPath *)indexPath {

    UICollectionViewCell* cell = [colView cellForItemAtIndexPath:indexPath];

    cell.contentView.backgroundColor = nil;

}
```

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 2-5 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 2-5 Selectively disabling actions in the Edit menu

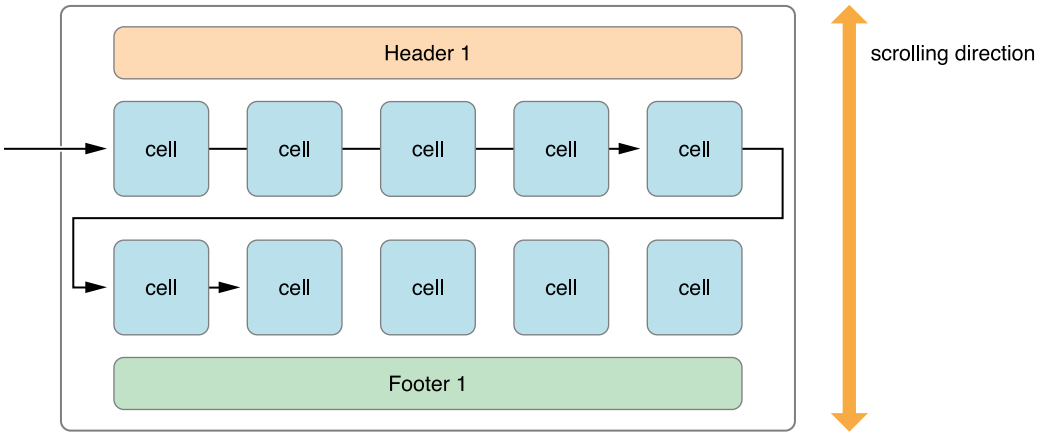
```
- (BOOL)collectionView:(UICollectionView *)collectionView
        canPerformAction:(SEL)action
        forItemAtIndexPath:(NSIndexPath *)indexPath
        withSender:(id)sender {
   // Support only copying nad pasting of cells.
   if ([NSStringFromSelector(action) isEqualToString:@"copy:"]
     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

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

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

Using the Flow Layout

The `UICollectionViewFlowLayout` class is a concrete layout object that you can use to arrange items in your collection views. 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 it 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 a grid of items, you could 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: You must specify the width and height of cells at a minimum. If you do not, 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.

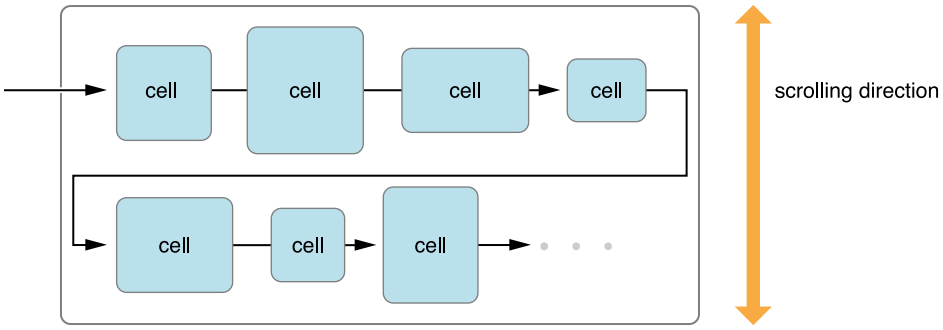
If you want to vary the spacing or size of items dynamically, you can do so using the methods of the `UICollectionViewDelegateFlowLayout` protocol. You implement these methods on the same delegate object you assigned to the collection view itself. If a given method exists, the flow layout object calls 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. (You 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



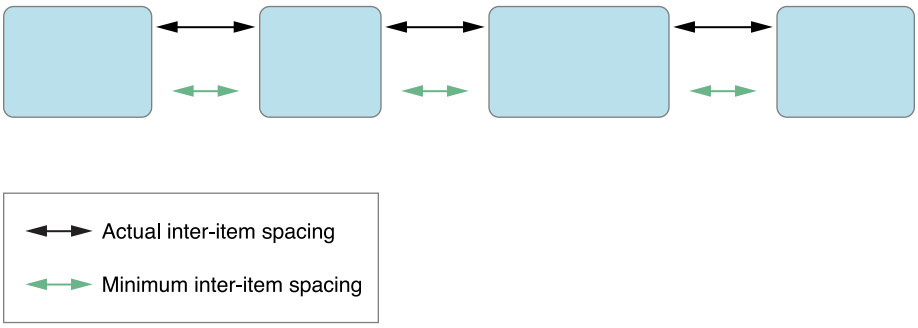
Note: When you specify different sizes for cells, the number of items on a single line can vary from line to line.

Specifying the Space Between Items and Lines

The flow layout lets you specify the minimum spacing between items on the same line and the minimum spacing between successive lines. It is important to remember 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. However, if there is extra space at the end of the line, the layout object increases the inter-item 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.

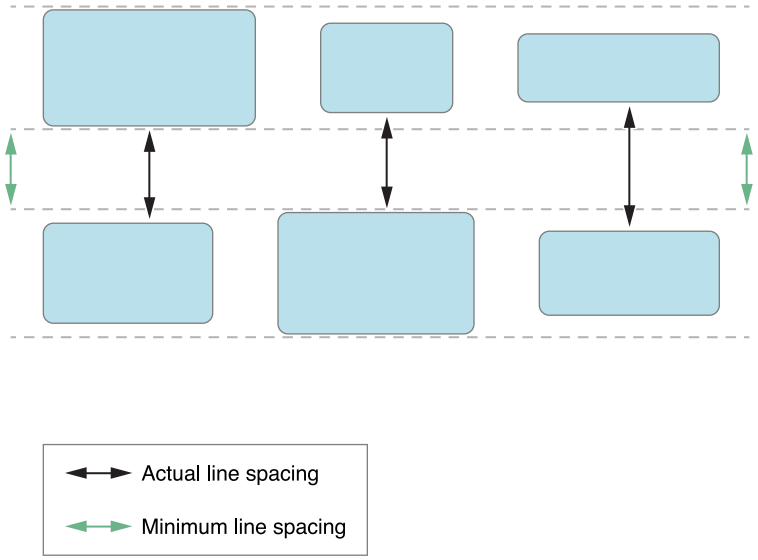
Figure 3-3 Actual spacing between items may be greater than the minimum



For inter-line 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. However, 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. In this case, 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, this results in actual line spacing that appears to be greater than the minimum.

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



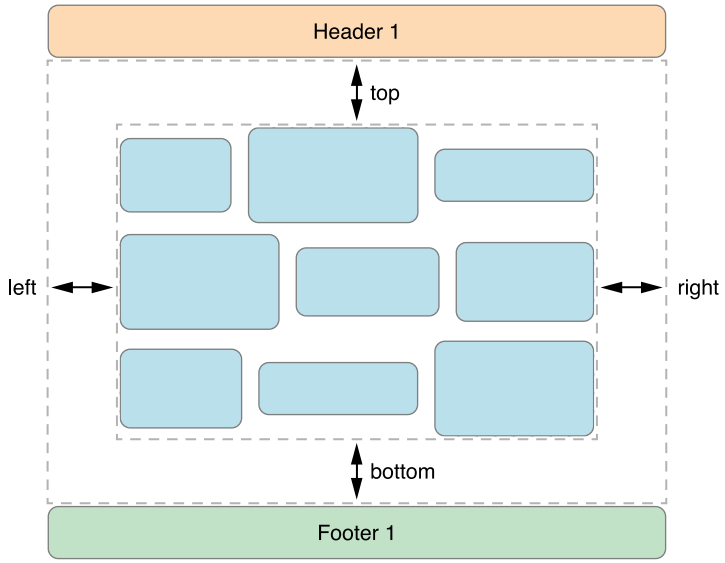
As with other flow layout attributes, you can use fixed spacing values or vary the values dynamically. Line and item spacing is handled on a section-by-section basis. Thus, the line and inter-item spacing is the same for all of the items in a given section but may vary between sections. You set the spacing 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 them 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 a 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 still 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	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: • `layoutAttributesForElementsInRect:` (required) • `layoutAttributesForItemAtIndexPath:` (required) • `layoutAttributesForSupplementaryViewOfKind: atIndexPath:` (to support new supplementary views) • `layoutAttributesForDecorationViewOfKind: atIndexPath:` (to support new decoration views) In your `layoutAttributesForElementsInRect:` method, you can call `super` 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. For information about providing attributes for views during layout, see "Providing Layout Attributes for Displayed Items" (page 40).
You want to tweak the layout attributes being returned by the flow layout	Override the `layoutAttributesForElementsInRect:` method and any of the methods that return layout attributes. The implementation of your methods should call `super`, modify the attributes provided by the parent class, and then return them.
You want to add new layout attributes for your cells and views	Create a custom subclass of `UICollectionViewLayoutAttributes` and add whatever properties you need to represent your custom layout information. Subclass `UICollectionViewFlowLayout` and override the `layoutAttributesClass` method. In your implementation of that method, return your custom subclass. You should also override the `layoutAttributesForElementsInRect:` method, the `layoutAttributesForItemAtIndexPath:` method, and any other methods that return layout attributes. In your custom implementations, you should set the values for any custom attributes you defined.

Scenario	Subclassing tips
You want to specify initial or final locations for items being inserted or deleted	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: • `initialLayoutAttributesForAppearingItemAtIndexPath:` • `initialLayoutAttributesForAppearingSupplementaryElementOfKind: atIndexPath:` • `initialLayoutAttributesForAppearingDecorationElementOfKind: atIndexPath:` • `finalLayoutAttributesForDisappearingItemAtIndexPath:` • `finalLayoutAttributesForDisappearingSupplementaryElementOfKind: atIndexPath:` • `finalLayoutAttributesForDisappearingDecorationElementOfKind: atIndexPath:` 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. If you override these methods, it is also recommended that you override the `prepareForCollectionViewUpdates:` and `finalizeCollectionViewUpdates` methods. You can use these methods to track which items are being inserted or deleted during the current cycle. For more information about how insertions and deletions work, see "Making Insertion and Deletion Animations More Interesting" (page 44).

Incorporating Gesture Support

You can add greater interactivity to your collection views through the use of gesture recognizers. Like any view, you can 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 and not to a specific cell or view. The `UICollectionView` class is a descendant of `UIScrollView`. 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.

Creating a gesture recognizer is 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 the Default Gesture Recognizers

The parent class of `UICollectionView` class installs a default tap gesture recognizer and a default long-press gesture recognizer to handle scrolling interactions. You should never try to reconfigure these default gesture recognizers or replace them with your own versions. 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 default ones that

are already installed. For example, you might configure a tap gesture recognizer to respond only to double-taps. You must then link your custom gesture recognizer to the default versions using the `requireGestureRecognizerToFail:` method. That method causes the default gesture recognizer to fire only when your gesture recognizer decides not to fire.

Listing 4-2 shows how you might link a custom tap gesture recognizer to the default recognizer associated with a collection view. In this case, the owning view controller adds the gesture recognizer to the collection view at load time. After creating the custom gesture recognizer, the code iterates through the collection view's default set of gesture recognizers looking for one of the same type. It then links the default gesture recognizer to the custom one that was just created. Finally, the view controller adds the gesture recognizer to the collection view. It adds the gesture recognizer last to avoid encountering it during the `for` loop.

Listing 4-2 Linking a default gesture recognizer to a custom gesture recognizer

```
UITapGestureRecognizer* tapGesture = [[UITapGestureRecognizer alloc]
initWithTarget:self action:@selector(handleTapGesture:)];
NSArray* recognizers = [self.collectionView gestureRecognizers];

// Make the default gesture recognizer wait until the custom one fails.
for (UIGestureRecognizer* aRecognizer in recognizers) {
   if ([aRecognizer isKindOfClass:[UITapGestureRecognizer class]])
      [aRecognizer requireGestureRecognizerToFail:tapGesture];
}

// Now add the gesture recognizer to the collection view.
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 is moving 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, you should really think about whether doing so is necessary. The `UICollectionViewFlowLayout` class provides a significant amount of behavior and can be adapted in several ways to achieve many different types of 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.
- You want to change all of the cell positions frequently enough that it would be more work to modify the existing flow layout.

The good news is that implementing a custom layout is not difficult from an API perspective. The hardest part is performing the calculations needed to determine the positions of items in the layout. Once 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. Most of the methods of that class can be overridden and used to tweak the layout behavior. However, only a handful of methods provide the core behavior for your layout object. The rest of the methods are there for you to override as needed. The core methods handle the following required tasks:

- They specify the size of the scrollable content area.
- They provide attributes for the cells and views that comprise your layout.

Although you can create a 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.

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

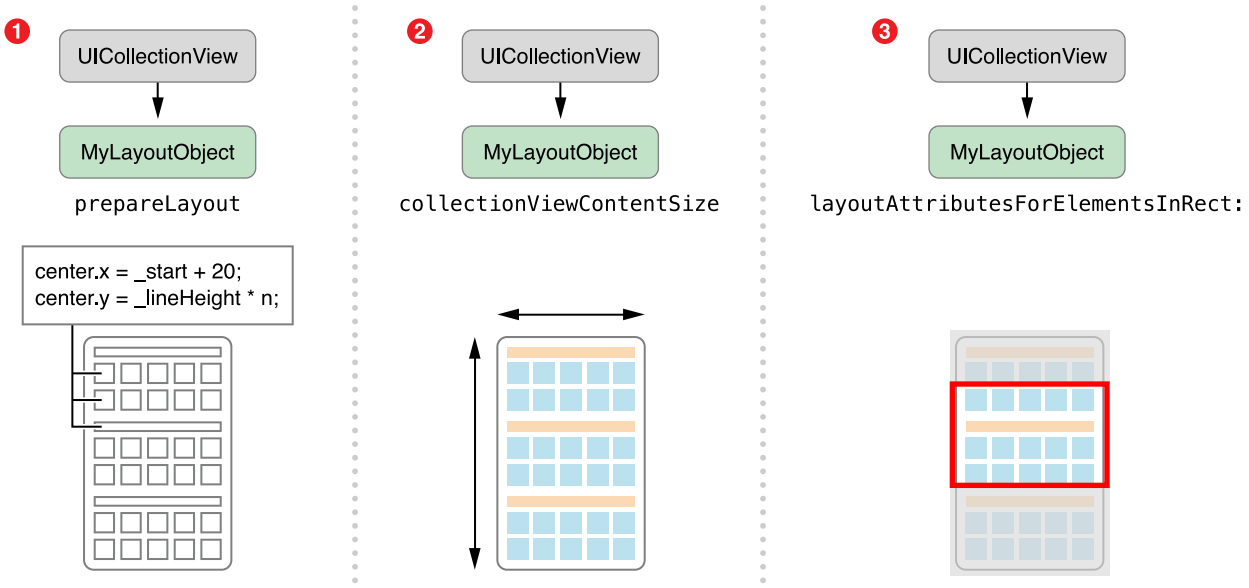
tell the collection view to update its layout explicitly by calling the `invalidateLayout` method of the layout object. That method throws away the existing layout information and forces the layout object to generate new layout information. Regardless of how you initiate a layout update, the actual layout 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. The `prepareLayout` method is your chance to do the up-front calculations needed to provide layout information.
2. The `collectionViewContentSize` method is where you return the overall size of the entire content area based on your initial calculations.
3. The `layoutAttributesForElementsInRect:` method returns 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. 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 two. The collection view uses the content size to configure its scroll view appropriately. 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.

Figure 5-1 Laying out your custom content



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`.

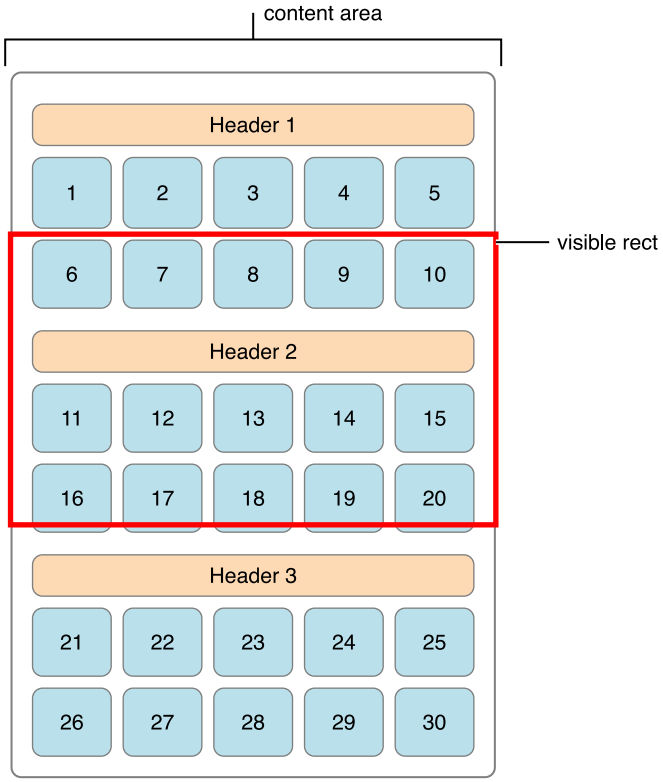
> **Note:** It is useful to remember that calling the `invalidateLayout` method does not begin the layout update process immediately. Instead, invalidating your layout object marks the layout as dirty and in need of being updated. During the next view update cycle, the collection view checks to see if its layout is dirty, and updates it if it is. Thus, you can call the `invalidateLayout` method multiple times in quick succession without triggering an immediate layout update each time.

Providing Layout Attributes for Displayed Items

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 layout object would need to create attribute objects for cells 6

through 20 and for the second header view. However, 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 to return the required attributes. The implementation of your `layoutAttributesForElementsInRect:` method would then follow these steps:

1. Iterate over the data generated by the `prepareLayout` method.
2. Check the frame of each item to see if 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. The `UICollectionViewLayoutAttributes` class is a wrapper for layout-related data and provides a convenient way to store that data. So creating those objects initially is an easy way to store the information and return it quickly.

> **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 42).

When creating new instances of the `UICollectionViewLayoutAttributes` class, you do so using 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. One of the attributes stored by the objects is whether it is a cell, supplementary view, or decoration view. The collection view uses that information to request the appropriate type of view from the data source object.

After creating each attributes object, you should then set the relevant attributes for the corresponding view. At a minimum, you should set the size and position of the view in the layout. In cases where the views of your layout overlap, it is also recommended that you assign a value to the `zIndex` property to ensure a consistent ordering of the views. Other properties let you control the visibility or appearance of the cell or view and can be changed as needed. For more information about layout attributes, see *UICollectionViewLayoutAttributes Class Reference*.

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.

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.

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. 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 solely by the layout object and are not managed by the collection view's data source object.

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

1. Register your decoration view with the layout object using the `registerClass:forDecorationViewOfKind:` or `registerNib:forDecorationViewOfKind:` method.
2. In your layout object's `layoutAttributesForElementsInRect:` method, create attributes for your decoration views just like 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 the appearance and disappearance of your decoration views.

The creation process for decoration views is different than 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. Thus, when a decoration view is needed, the collection view creates it for you and applies the attributes provided by the layout object.

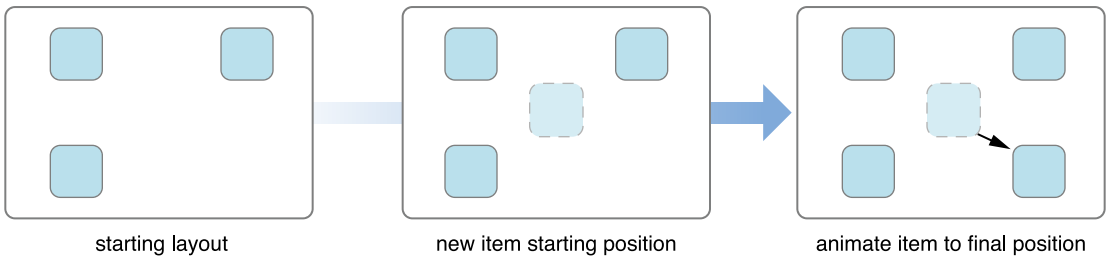
> **Note:** When creating the attributes for your decoration views, do not forget to take into account the `zIndex` property. You can use the zIndex attribute to layer your decoration views behind (or, if you prefer, in front of) the displayed cells and supplementary views.

Making Insertion and Deletion Animations More Interesting

The insertion and deletion of cells and views poses an interesting challenge during layout. Inserting a cell can cause the layout for other cells and views to change. But while the layout object knows how to animate existing cells and views from their current locations to new locations, it does not have a 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 end point of any animations.

To understand how initial attributes work, it helps to see an example. The starting layout in 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 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 5-1 shows the code you might use to specify the initial attributes for the inserted cell from Figure 5-3 (page 44). 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 5-1 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;
}
```

The process for handling deletions is identical 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.

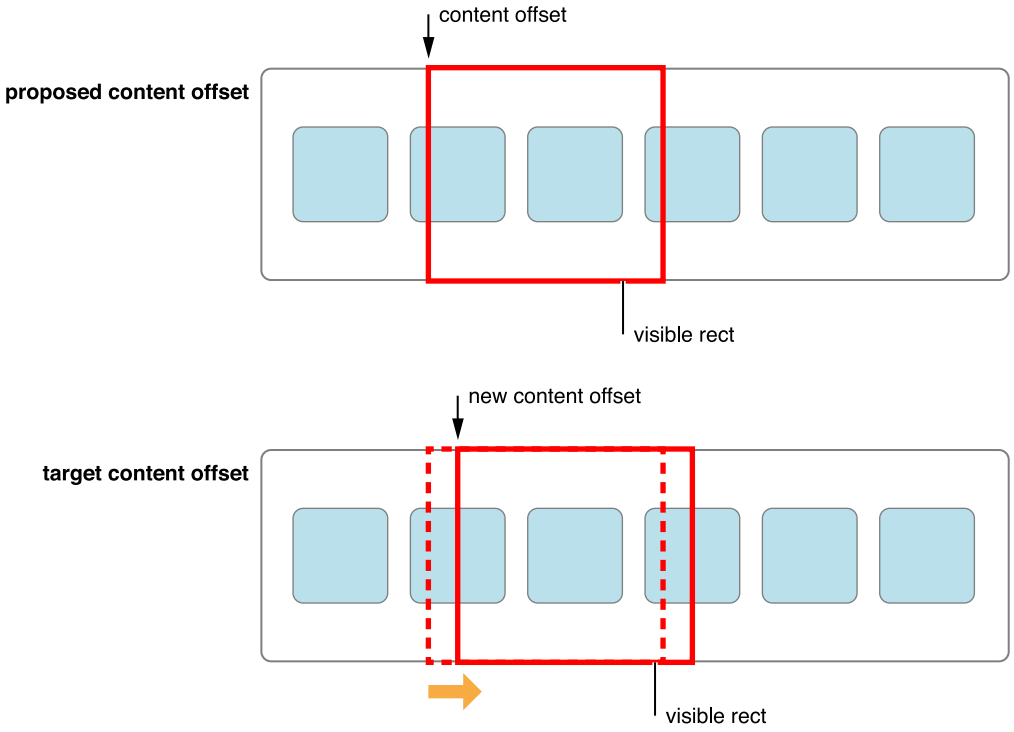
Improving the Scrolling Experience of Your Layout

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. Once 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. Thus, 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.

Document Revision History

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

Date	Notes
2012-09-19	New document that describes how to use collection views in iOS apps.

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