Class

UICollectionView

An object that manages an ordered collection of data items and presents them using customizable layouts.

Declaration

```
class UICollectionView : UIScrollView
```

Overview

When adding a collection view to your user interface, your app's main job is to manage the data associated with that collection view. The collection view gets its data from the data source object, which is an object that conforms to the `UICollectionViewDataSource` Protocoland is provided by your app. Data in the collection view is organized into individual items, which can then be grouped into sections for presentation. An item is the smallest unit of data you want to present. For example, in a photos app, an item might be a single image. The collection view presents items onscreen using a cell, which is an instance of the `UICollectionViewCell` class that your data source configures and provides.

In addition to its cells, a collection view can present data using other types of views too. These supplementary views can be things like section headers and footers that are separate from the individual cells but still convey some sort of information. Support for supplementary views is optional and defined by the collection view's layout object, which is also responsible for defining the placement of those views.

Besides embedding it in your user interface, you use the methods of `UICollectionView`object to ensure that the visual presentation of items matches the order in your data source object. Thus, whenever you add, delete, or rearrange data in your collection, you use the methods of this class to insert, delete, and rearrange the corresponding cells. You also use the collection view object to manage the selected items, although for this behavior the collection view works with its associated `delegate` object.

Collection Views and Layout Objects

A very important object associated with a collection view is the layout object, which is a subclass of the `UICollectionViewLayout` class. The layout object is responsible for defining the organization and location of all cells and supplementary views inside the collection view. Although it defines their locations, the layout object does not actually apply that information to the corresponding views. Because the creation of cells and supplementary views involves coordination between the collection view and your data source object, the collection view actually applies layout information to the views. Thus, in a sense, the layout object is like another data source, only providing visual information instead of item data.

You normally specify a layout object when creating a collection view but you can also change the layout of a collection view dynamically. The layout object is stored in the `collectionViewLayout`

property. Setting this property directly updates the layout immediately, without animating the changes. If you want to animate the changes, you must call the `setCollectionViewLayout(_:animated:completion:)` method instead.

If you want to create an interactive transition—one that is driven by a gesture recognizer or touch events—use the `startInteractiveTransition(to:completion:)` method to change the layout object. That method installs an intermediate layout object whose purpose is to work with your gesture recognizer or event-handling code to track the transition progress. When your event-handling code determines that the transition is finished, it calls the `finishInteractiveTransition()` or `cancelInteractiveTransition()` method to remove the intermediate layout object and install the intended target layout object.

Creating Cells and Supplementary Views

The collection view's data source object provides both the content for items and the views used to present that content. When the collection view first loads its content, it asks its data source to provide a view for each visible item. To simplify the creation process for your code, the collection view requires that you always dequeue views, rather than create them explicitly in your code. There are two methods for dequeueing views. The one you use depends on which type of view has been requested:

- Use the `dequeueReusableCell(withReuseIdentifier:for:)` to get a cell for an item in the collection view.
- Use the `dequeueReusableSupplementaryView(ofKind:withReuseIdentifier:for:)` method to get a supplementary view requested by the layout object.

Before you call either of these methods, you must tell the collection view how to create the corresponding view if one does not already exist. For this, you must register either a class or a nib file with the collection view. For example, when registering cells, you use the `register(_:forCellWithReuseIdentifier:)` or `register(_:forCellWithReuseIdentifier:)`method. As part of the registration process, you specify the reuse identifier that identifies the purpose of the view. This is the same string you use when dequeueing the view later.

After dequeueing the appropriate view in your delegate method, configure its content and return it to the collection view for use. After getting the layout information from the layout object, the collection view applies it to the view and displays it.

For more information about implementing the data source methods to create and configure views, see `UICollectionViewDataSource`.

Reordering Items Interactively

Collection views allow you to move items around based on user interactions. Normally, the order of items in a collection view is defined by your data source. If you support the ability for users to reorder items, you can configure a gesture recognizer to track the user's interactions with a collection view item and update that item's position.

To begin the interactive repositioning of an item, call the `beginInteractiveMovementForItem(at:)` method of the collection view. While your gesture recognizer is tracking touch events, call the `updateInteractiveMovementTargetPosition(_:)` method to report changes in the touch location. When you are done tracking the gesture, call the `endInteractiveMovement()` or `cancelInteractiveMovement()` method to conclude the interactions and update the collection view.

During user interactions, the collection view invalidates its layout dynamically to reflect the current position of the item. If you do nothing, the default layout behavior repositions the items for you, but you can customize the layout animations if you want. When interactions finish, updates its data source object with the new location of the item.

The `UICollectionViewController` class provides a default gesture recognizer that you can use to rearrange items in its managed collection view. To install this gesture recognizer, set the `installsStandardGestureForInteractiveMovement` property of the collection view controller to `true`.

Interface Builder Attributes

Table 1 lists the attributes that you configure for collection views in Interface Builder.

Table 1 Collection view attributes

Attribute	Description
Items	The number of prototype cells. This property controls the specified number of prototype cells for you to configure in your storyboard. Collection views must always have at least one cell and may have multiple cells for displaying different types of content or for displaying the same content in different ways.
Layout	The layout object to use. Use this control to select between the `UICollectionViewFlowLayout` object and a custom layout object that you define. When the flow layout is selected, you can also configure the scrolling direction for the collection view's content and whether the flow layout has header and footer views. Enabling header and footer views adds reusable views to your storyboard that you can configure with your header and footer content. You can also create those views programmatically. When a custom layout is selected, you must specify the `UICollectionViewLayout` subclass to use.

When the Flow layout is selected, the Size inspector for the collection view contains additional attributes for configuring flow layout metrics. Use those attributes to configure the size of your cells, the size of headers and footers, the minimum spacing between cells, and any margins around each section of cells. For more information about the meaning of the flow layout metrics, see `UICollectionViewFlowLayout`.

Internationalization

A collection view has no direct content of its own to internationalize. Instead, you internationalize the cells and reusable views of the collection view. For more information about internationalization, see Internationalization and Localization Guide.

Accessibility

A collection view has no content of its own to make accessible. If your cells and reusable views contain standard UIKit controls such as `UILabel` and `UITextField`, you can make those controls accessible. When a collection view changes its onscreen layout, it posts the `UIAccessibilityLayoutChangedNotification` notification.

For general information about making your interface accessible, see Accessibility Programming Guide for iOS.

Topics

Initializing a Collection View

`init(frame: CGRect, collectionViewLayout: UICollectionViewLayout)`
Initializes and returns a newly allocated collection view object with the specified frame and layout.

`init?(coder: NSCoder)`

Providing the Collection View Data

`var dataSource: UICollectionViewDataSource?`
The object that provides the data for the collection view.

`protocol UICollectionViewDataSource`
An object that adopts the `UICollectionViewDataSource` protocol is responsible for providing the data and views required by a collection view. A data source object represents your app's data model and vends information to the collection view as needed. It also handles the creation and configuration of cells and supplementary views used by the collection view to display your data.

`protocol UICollectionViewDataSourcePrefetching`
A protocol that provides advance warning of the data requirements for a collection view, allowing the triggering of asynchronous data load operations.

Managing Collection View Interactions

`var delegate: UICollectionViewDelegate?`
The object that acts as the delegate of the collection view.

`protocol UICollectionViewDelegate`
The `UICollectionViewDelegate` protocol defines methods that allow you to manage the selection and highlighting of items in a collection view and to perform actions on those items. The methods of this protocol are all optional.

Configuring the Background View

`var backgroundView: UIView?`
The view that provides the background appearance.

Prefetching Collection View Cells and Data

`UICollectionView` provides two prefetching techniques you can use to improve responsiveness:

- **Cell prefetching** prepares cells in advance of the time they are required. When a collection view requires a large number of cells simultaneously—for example, a new row of cells in grid layout—the cells are requested earlier than the time required for display. Cell rendering is therefore spread across multiple layout passes, resulting in a smoother scrolling experience. Cell prefetching is enabled by default.
- **Data prefetching** provides a mechanism whereby you are notified of the data requirements of a collection view in advance of the requests for cells. This is useful if the content of your cells relies on an expensive data loading process, such as a network request. Assign an object that conforms to the `UICollectionViewDataSourcePrefetching` protocol to the `prefetchDataSource` property to receive notifications of when to prefetch data for cells.

```
var isPrefetchingEnabled: Bool
```
Denotes whether cell and data prefetching are enabled.

```
var prefetchDataSource: UICollectionViewDataSourcePrefetching?
```
The object that acts as the prefetching data source for the collection view, receiving notifications of upcoming cell data requirements.

Creating Collection View Cells

```
func register(AnyClass?, forCellWithReuseIdentifier: String)
```
Register a class for use in creating new collection view cells.

```
func register(UINib?, forCellWithReuseIdentifier: String)
```
Register a nib file for use in creating new collection view cells.

```
func register(AnyClass?, forSupplementaryViewOfKind: String, withReuseIdentifier: String)
```
Registers a class for use in creating supplementary views for the collection view.

```
func register(UINib?, forSupplementaryViewOfKind: String, withReuseIdentifier: String)
```
Registers a nib file for use in creating supplementary views for the collection view.

```
func dequeueReusableCell(withReuseIdentifier: String, for: IndexPath) -> UICollectionViewCell
```
Returns a reusable cell object located by its identifier

```
func dequeueReusableSupplementaryView(ofKind: String, withReuseIdentifier: String, for: IndexPath) -> UICollectionReusableView
```
Returns a reusable supplementary view located by its identifier and kind.

Changing the Layout

`var collectionViewLayout: UICollectionViewLayout`

The layout used to organize the collected view's items.

`func setCollectionViewLayout(UICollectionViewLayout, animated: Bool)`

Changes the collection view's layout and optionally animates the change.

`func setCollectionViewLayout(UICollectionViewLayout, animated: Bool, completion: ((Bool) -> Void)? = nil)`

Changes the collection view's layout and notifies you when the animations complete.

`func startInteractiveTransition(to: UICollectionViewLayout, completion: UICollectionView.LayoutInteractiveTransitionCompletion? = nil) -> UICollectionViewTransitionLayout`

Changes the collection view's current layout using an interactive transition effect.

`func finishInteractiveTransition()`

Tells the collection view to finish an interactive transition by installing the intended target layout.

`func cancelInteractiveTransition()`

Tells the collection view to abort an interactive transition and return to its original layout object.

{}Customizing Collection View Layouts

Customize a view layout by changing the size of cells in the flow or implementing a mosaic style.

Getting the State of the Collection View

`var numberOfSections: Int`

Returns the number of sections displayed by the collection view.

`func numberOfItems(inSection: Int) -> Int`

Returns the number of items in the specified section.

`var visibleCells: [UICollectionViewCell]`

Returns an array of visible cells currently displayed by the collection view.

Inserting, Moving, and Deleting Items

`func insertItems(at: [IndexPath])`
Inserts new items at the specified index paths.

`func moveItem(at: IndexPath, to: IndexPath)`
Moves an item from one location to another in the collection view.

`func deleteItems(at: [IndexPath])`
Deletes the items at the specified index paths.

Inserting, Moving, and Deleting Sections

`func insertSections(IndexSet)`
Inserts new sections at the specified indexes.

`func moveSection(Int, toSection: Int)`
Moves a section from one location to another in the collection view.

`func deleteSections(IndexSet)`
Deletes the sections at the specified indexes.

Reordering Items Interactively

`func beginInteractiveMovementForItem(at: IndexPath) -> Bool`
Initiates the interactive movement of the item at the specified index path.

`func updateInteractiveMovementTargetPosition(CGPoint)`
Updates the position of the item within the collection view's bounds.

`func endInteractiveMovement()`
Ends interactive movement tracking and moves the target item to its new location.

`func cancelInteractiveMovement()`
Ends interactive movement tracking and returns the target item to its original location.

Managing Drag Interactions

`var dragDelegate: UICollectionViewDragDelegate?`
The delegate object that manages the dragging of items from the collection view.

`protocol UICollectionViewDragDelegate`
The interface for initiating drags from a collection view.

`var hasActiveDrag: Bool`
A Boolean value indicating whether items were lifted from the collection view and have not yet been dropped.

`var dragInteractionEnabled: Bool`
A Boolean value indicating whether the collection view supports drags and drops between apps.

Managing Drop Interactions

`var dropDelegate: UICollectionViewDropDelegate?`
The delegate object that manages the dropping of items into the collection view.

`var hasActiveDrop: Bool`
A Boolean value indicating whether the collection view is currently tracking a drop session.

`var reorderingCadence: UICollectionView.ReorderingCadence`
The speed at which items in the collection view are reordered to show potential drop locations.

`enum UICollectionView.ReorderingCadence`
Constants indicating the speed at which collection view items are reorganized during a drop.

Managing the Selection

`var allowsSelection: Bool`
A Boolean value that indicates whether users can select items in the collection view.

`var allowsMultipleSelection: Bool`
A Boolean value that determines whether users can select more than one item in the collection view.

`var indexPathsForSelectedItems: [IndexPath]?`
The index paths for the selected items.

`func selectItem(at: IndexPath?, animated: Bool, scrollPosition: UICollectionView.ScrollPosition)`
Selects the item at the specified index path and optionally scrolls it into view.

`func deselectItem(at: IndexPath, animated: Bool)`
Deselects the item at the specified index.

Managing Focus

```
var rememberesLastFocusedIndexPath: Bool
```

A Boolean value indicating whether the collection view automatically assigns the focus to the item at the last focused index path.

Locating Items and Views in the Collection View

```
func indexPathForItem(at: CGPoint) -> IndexPath?
```

Returns the index path of the item at the specified point in the collection view.

```
var indexPathsForVisibleItems: [IndexPath]
```

An array of the visible items in the collection view.

```
func indexPath(for: UICollectionViewCell) -> IndexPath?
```

Returns the index path of the specified cell.

```
func cellForItem(at: IndexPath) -> UICollectionViewCell?
```

Returns the visible cell object at the specified index path.

```
func indexPathsForVisibleSupplementaryElements(ofKind: String) ->
[IndexPath]
```

Returns the index paths of all visible supplementary views of the specified type.

```
func supplementaryView(forElementKind: String, at: IndexPath) ->
UICollectionReusableView?
```

Returns the supplementary view at the specified index path.

```
func visibleSupplementaryViews(ofKind: String) -> [UICollection
ReusableView]
```

Returns an array of the visible supplementary views of the specified kind.

Getting Layout Information

```
func layoutAttributesForItem(at: IndexPath) -> UICollectionView
LayoutAttributes?
```

Returns the layout information for the item at the specified index path.

```
func layoutAttributesForSupplementaryElement(ofKind: String, at:
IndexPath) -> UICollectionViewLayoutAttributes?
```

Returns the layout information for the specified supplementary view.

Scrolling an Item Into View

```
func scrollToItem(at: IndexPath, at: UICollectionView.Scroll
Position, animated: Bool)
```

Scrolls the collection view contents until the specified item is visible.

Animating Multiple Changes to the Collection View

```
func performBatchUpdates((() -> Void)?, completion: ((Bool) -> Void)? = nil)
```
Animates multiple insert, delete, reload, and move operations as a group.

Reloading Content

```
var hasUncommittedUpdates: Bool
```
A Boolean value indicating whether the collection view contains drop placeholders or is reordering its items as part of handling a drop.

```
func reloadData()
```
Reloads all of the data for the collection view.

```
func reloadSections(IndexSet)
```
Reloads the data in the specified sections of the collection view.

```
func reloadItems(at: [IndexPath])
```
Reloads just the items at the specified index paths.

Constants

```
struct UICollectionView.ScrollPosition
```
Constants that indicate how to scroll an item into the visible portion of the collection view.

```
typealias UICollectionView.LayoutInteractiveTransitionCompletion
```
The completion block called at the end of an interactive transition for a collection view.

Type Properties

```
class let elementKindSectionFooter: String
```

```
class let elementKindSectionHeader: String
```

Enumerations

```
enum UICollectionView.ElementCategory
```
Constants specifying the type of view.

```
enum UICollectionView.ScrollDirection
```
Constants indicating the direction of scrolling for the layout.

Relationships

Inherits From

UIScrollView

Conforms To

CVarArg
Equatable
Hashable
UIAccessibilityIdentification
UIDataSourceTranslating
UIPasteConfigurationSupporting
UISpringLoadedInteractionSupporting
UIUserActivityRestoring