Class

NSCollectionView

An ordered collection of data items displayed in a customizable layout.

macOS 10.5+

```
@MainActor
class NSCollectionView
```

Overview

The simplest type of collection view displays its items in a grid, but you can define layouts to arrange items however you like. For example, you might create a layout where items are arranged in a circle. You can also change layouts dynamically at runtime whenever you need to present items differently.

You can add collection views to your interface using Interface Builder or create them programmatically in your view controller or window controller code. It is recommended that you configure your collection view with a data source object, which is an object that conforms to the `NSCollectionViewDataSource` protocol. Data sources support multiple sections and the modern layout architecture and are the preferred way for specifying your data.

In addition to displaying items, collection views support the display of supplementary and decoration views. Support for supplementary and decoration views is defined by the current layout object, but both types of views add to the visual presentation of your content. Supplementary views are associated with a specific section and can be used to create header and footer views for a related group of items. Decoration views are purely visual adornments and can be used to implement dynamic backgrounds or other types of configurable visual content.

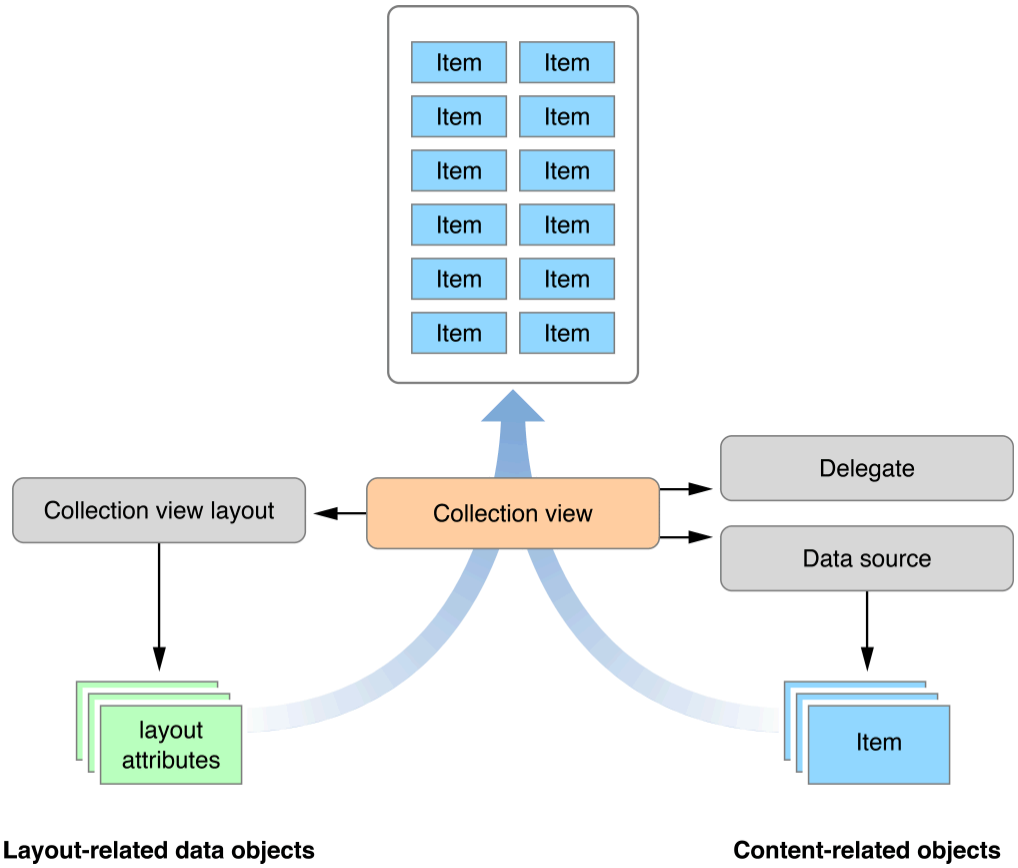
The layout of a collection view can be changed dynamically by assigning a new layout object to the `collectionViewLayout` property. Changing the layout object updates the appearance of the collection view without animating the changes.

The Objects of a Collection View Interface

An `NSCollectionView` object itself is a facilitator, taking information from disparate sources and merging them together to create an overall interface:

- The data source object provides both the data and the views used to display that data. You define the data source object by implementing the methods of the `NSCollectionViewDataSource` protocol in one of your app's objects.
- The visual representation of items is provided by the `NSCollectionViewItem` class. Item objects are view controllers and you use their views to display your app's data. The data source creates items on demand and returns them to the collection view for display.
- The collection view delegate makes decisions about behaviors. The delegate also coordinates the dragging and dropping of items. You define the delegate by implementing the methods of the `NSCollectionViewDelegate` protocol in one of your app's objects.
- The layout object specifies the position and appearance of items onscreen. AppKit defines layout objects that you can use as-is, but you can also define custom layouts by subclassing `NSCollectionViewLayout`.

Figure 1 illustrates how the collection view works with its other objects to create its final appearance. The collection view obtains the views for items and supplementary views from its data source, which creates the views and fills them with data. The layout object provides the layout attributes needed to position those items and supplementary views onscreen. The collection view merges the two sets of information to create the final appearance that the user sees onscreen.



There are other helper classes and protocols that you can use to customize the layout behavior and other aspects of the collection view interface. For example, when using a flow layout object (`NSCollectionViewFlowLayout`), you can modify the flow layout's behavior using the methods of the `NSCollectionViewDelegateFlowLayout` protocol. When implementing a custom layout, you might also work with `NSCollectionViewUpdateItem` and `NSCollectionViewLayoutInvalidationContext` objects, which help the layout object manage updates.

Managing the Collection View's Content

Data for the collection view is managed by the *data source object*—that is an object that adopts the methods of the `NSCollectionViewDataSource` protocol. You are responsible for defining the data source used by your collection view. The data source provides information about the number of sections and items in the collection view and it provides the visual representation of that data. Every data source object is required to implement the following methods:

- `collectionView(_:numberOfItemsInSection:)`
- `collectionView(_:itemForRepresentedObjectAt:)`

The `NSCollectionViewItem` class defines the visual appearance of items in the collection view. Your data source object vends items from its `collectionView(_:itemForRepresentedObjectAt:)` method, creating and configuring the item in one step. Each item is essentially a snapshot of the data it represents. Items are often short-lived because they can be recycled by the collection view and reused to display new data. As a result, never store references to items in your app.

Supplementary views are another way to display data in your interface. Each layout object defines the supplementary views it supports, and different layouts can define supplementary views for different purposes. For example, an `NSCollectionViewFlowLayout` object lets you add header and footer views to each section. Your data source must know enough about the layout to know which supplementary views are supported by the layout object and how those views are displayed. The data source can then provide supplementary views when asked for them.

When your content changes in a way that requires you to update what the collection view displays, call the `reloadData()`, `reloadSections(_:)`, or `reloadItems(at:)` method to perform that update. These methods cause the collection view to discard the views currently being used to display your content and ask for new ones. Never try to modify the views associated with your items directly. The collection view does not maintain views for all items, only those that are currently being displayed. Reloading the items ensures that the views are updated correctly.

For more information on defining your data source object, see `NSCollectionViewDataSource`.

Inserting, Deleting, and Moving Content

The collection view includes methods for inserting, deleting, and moving items and sections. All of these methods affect only what the collection view displays onscreen; they do not change the data in the associated data source object. As a result, when updating your collection view's content, always do the following:

1. Update the internal structures of your data source object first.

2. Call the `NSCollectionView` methods to insert, delete, or move items and sections.

When you call methods like `insertItems(at:)` or `deleteSections(_:)`, the collection view fetches any new data from your data source object and then updates the layout. When inserting, moving, or deleting items, the collection view updates the layout for all affected items, which might include items not directly affected by the operation. For example, inserting one item might require adjusting the onscreen position of many other items. When the layout attributes for any visible items changes, the collection view animates those changes into place automatically.

The layout object determines how inserted and deleted items are animated into position. Because newly inserted items are not onscreen initially, the layout object provides the initial layout attributes for those items. Similarly, the layout object provides the final layout attributes for any items that are being deleted. For example, the layout object might specify final layout attributes that are offscreen so that a deleted item animates out of the visible rectangle.

Because individual methods for inserting, deleting, and moving content animate their changes right away, you must use the `performBatchUpdates(_:completionHandler:)` method when you want to animate multiple changes together. The `performBatchUpdates(_:completionHandler:)` method takes a block containing all of the insert, delete, move, and

reload method calls you need to update the collection view. All of those operations are captured and performed as a single animated sequence.

Interface Builder Configuration Options

Xcode lets you configure information about your collection view in your storyboard and nib files. The table below shows the basic collection view attributes. Additional attributes are available based on the selected value for the Layout attribute.

Attribute	Description
Layout	The type of layout object to use. The Flow, Grid, and Custom options are preferred because they enable the modern collection view behavior.
Colors	The option to specify alternating colors for the collection view's background.
Primary	The primary color to use with the collection view.
Secondary	The secondary color to use with the collection view.
Selection	The options for selecting items. Use these options to enable or disable selections altogether and to specify whether the collection view supports the selection of multiple items or no items.

The table below shows the attributes you can configure when you set the Layout attribute to Flow.

Attribute	Description
Scroll Direction	The scrolling direction for content. The flow layout allows scrolling in one dimension only. The other dimension is pinned to the size of the collection view itself. For example, when vertical scrolling is selected, the width of the content area is set to the width of the collection view.
Item Size	The default size of newly created items. The collection view's delegate can override the default size values and specify different values for each item.
Header Size	The default size of header views. The layout object uses only the dimension that does not match the current scrolling direction. For example, for a vertically scrolling collection view, the layout sets only the width of the footer to the specified value. The collection view's delegate can override the default size values.

Attribute	Description
Footer Size	The default size of footer views. The layout object uses only the dimension that does not match the current scrolling direction. For example, for a vertically scrolling collection view, the layout sets only the width of the footer to the specified value. The collection view's delegate can override the default size values using methods of the `NSCollectionViewDelegateFlowLayout` protocol.
Min Spacing	The minimum spacing between items and lines. The item spacing is the minimum amount of space for items in the same row or column (depending on the scroll direction). The line spacing is the minimum space between rows or columns. The actual amount of space used between items and lines may be greater than the minimum.
Section Inset	The margins imposed on each section. Margins set the distance between the header view and the items, between the sides of the collection view and the items, and between the items and the footer view.

The table below shows the attributes you can configure when you set the Layout attribute to Grid.

Attribute	Description
Dimensions	The number of rows and columns to display. Use these attributes to configure the grid dimensions.
Min Item Size	The minimum width and height for items.
Max Item Size	The maximum width and height for items.

The table below shows the attributes you can configure when you set the Layout attribute to Custom.

Attribute	Description
Class	The name of the `NSCollectionViewLayout` subclass you want to use.
Module	The Swift module containing the class. Leave this attribute blank for classes in the current module.

The table below shows the attributes you can configure when you set the Layout attribute to Content Array (Legacy).

Attribute	Description
Dimensions	The number of rows and columns to display. Use these attributes to configure the grid dimensions.

Legacy Collection View Support

Prior to OS X v10.11, the collection view always displayed its contents in a grid structure that could not be changed. The data for the collection view was stored in the `content` property, which was often populated with data using bindings. You specified the visual appearance for the collection view's data by creating an `NSCollectionViewItem` object and assigning it to the `itemPrototype` property. That item object acted as a template and was used to create all of the items in the collection view.

You are encouraged to use the modern collection view architecture when configuring collection views in macOS 10.11 and later. Use the legacy architecture only for apps that must run in earlier versions of macOS.

For more information about how to configure a collection view using the legacy architecture, see Collection View Programming Guide for macOS.

Topics

Providing the Collection View's Data

`var dataSource: (any NSCollectionViewDataSource)?`

An object that provides data for the collection view.

`protocol NSCollectionViewDataSource`

A set of methods that a data source object implements to provide the information and view objects that a collection view requires to present content.

Configuring the Collection View

`var delegate: (any NSCollectionViewDelegate)?`

The collection view's delegate object.

```
protocol NSCollectionViewDelegate
```

A set of methods that you use to manage the behavior of a collection view.

```
var content: [Any]
```

An array that provides data for the collection view.

```
var backgroundView: NSView?
```

The background view placed behind all items and supplementary views.

```
var backgroundColors: [NSColor]!
```

An array containing the collection view's background colors.

```
var backgroundViewScrollsWithContent: Bool
```

A Boolean value that indicates whether the collection view's background view scrolls with the items and other content.

Creating Collection View Items

```
func makeItem(withIdentifier: NSUserInterfaceItemIdentifier, for: IndexPath) -> NSCollectionViewItem
```

Creates or returns a reusable item object of the specified type.

```
func register(AnyClass?, forItemWithIdentifier: NSUserInterfaceItemIdentifier)
```

Registers a class to use when creating new items in the collection view.

```
func register(NSNib?, forItemWithIdentifier: NSUserInterfaceItemIdentifier)
```

Registers a nib file to use when creating items in the collection view.

```
func makeSupplementaryView(ofKind: NSCollectionView.SupplementaryElementKind, withIdentifier: NSUserInterfaceItemIdentifier, for: IndexPath) -> NSView
```

Creates or returns a reusable supplementary view of the specified type.

```
func register(AnyClass?, forSupplementaryViewOfKind: NSCollectionView.SupplementaryElementKind, withIdentifier: NSUserInterfaceItemIdentifier)
```

Registers a class to use when creating new supplementary views in the collection view.

```
func register(NSNib?, forSupplementaryViewOfKind: NSCollectionView.
SupplementaryElementKind, withIdentifier: NSUserInterfaceItemIdentifier
)
```

Registers a nib file to use when creating supplementary views in the collection view.

```
typealias SupplementaryElementKind
```

```
struct NSUserInterfaceItemIdentifier
```

Changing the Layout

```
var collectionViewLayout: NSCollectionViewLayout?
```

The layout object used to organize the collection view's content.

Reloading Content

```
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: Set<IndexPath>)
```

Reloads only the specified items.

Prefetching Collection View Cells and Data

```
var prefetchDataSource: (any NSCollectionViewPrefetching)?
```

```
protocol NSCollectionViewPrefetching
```

Getting the State of the Collection View

```
var numberOfSections: Int
```

The number of sections in the collection view.

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

Returns the number of items in the specified section.

Inserting, Moving, and Deleting Items

`func insertItems(at: Set<IndexPath>)`

Inserts new items into the collection view at the specified locations.

`func moveItem(at: IndexPath, to: IndexPath)`

Moves an item from one location to another in the collection view.

`func deleteItems(at: Set<IndexPath>)`

Deletes the items at the specified index paths.

Inserting, Moving, Deleting, and Collapsing Sections

`func insertSections(IndexSet)`

Inserts new sections at the specified indexes.

`func moveSection(Int, toSection: Int)`

Moves a section from its current location to a new location.

`func deleteSections(IndexSet)`

Deletes the specified sections and their contained items.

`func toggleSectionCollapse(Any)`

Collapses the section in which the sender resides into a single horizontally scrollable row.

Managing the Selection

`var isSelectable: Bool`

A Boolean value that indicates whether the user may select items in the collection view.

`var allowsMultipleSelection: Bool`

A Boolean value that indicates whether the user may select more than one item in the collection view.

`var allowsEmptySelection: Bool`

A Boolean value indicating whether the collection view may have no selected items.

`var selectionIndexPaths: Set<IndexPath>`

The set of index paths representing the currently selected items.

`func selectAll(Any?)`

Selects all items in the collection view, if doing so is possible.

```
func deselectAll(Any?)
```

Deselects all items in the collection view.

```
func selectItems(at: Set<IndexPath>, scrollPosition: NSCollectionView.ScrollPosition)
```

Adds the specified items to the current selection and optionally scrolls the items into position.

```
func deselectItems(at: Set<IndexPath>)
```

Removes the specified items from the current selection.

Locating Items and Views

```
func visibleItems() -> [NSCollectionViewItem]
```

Returns an array of the actively managed items in the collection view.

```
func indexPathsForVisibleItems() -> Set<IndexPath>
```

Returns the index paths of the currently active items.

```
func visibleSupplementaryViews(ofKind: NSCollectionView.SupplementaryElementKind) -> [any NSView & NSCollectionViewElement]
```

Returns an array of the actively managed supplementary views in the collection view.

```
func indexPathsForVisibleSupplementaryElements(ofKind: NSCollectionView.SupplementaryElementKind) -> Set<IndexPath>
```

Returns the index paths of the currently active supplementary views.

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

Returns the index path of the specified item.

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

Returns the index path of the item at the specified point.

```
func item(at: IndexPath) -> NSCollectionViewItem?
```

Returns the item associated with the specified index path.

```
func supplementaryView(forElementKind: NSCollectionView.SupplementaryElementKind, at: IndexPath) -> (any NSView & NSCollectionViewElement)?
```

Returns the supplementary view associated with the specified index path.

```
func scrollToItems(at: Set<IndexPath>, scrollPosition: NSCollectionView.ScrollPosition)
```

Scrolls the collection view contents until the specified items are visible.

Getting Layout Information

```
func layoutAttributesForItem(at: IndexPath) -> NSCollectionViewLayoutAttributes?
```

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

```
func layoutAttributesForSupplementaryElement(ofKind: NSCollectionView.SupplementaryElementKind, at: IndexPath) -> NSCollectionViewLayoutAttributes?
```

Returns the layout information for the supplementary view at the specified index path.

Animating Multiple Changes

```
func performBatchUpdates((() -> Void)?, completionHandler: ((Bool) -> Void)?)
```

Encapsulates multiple insert, delete, reload, and move operations into a single animated operation.

Working with the Responder Chain

```
var isFirstResponder: Bool
```

A Boolean value indicating whether the collection view is the first responder.

Getting a Drag Image

```
func draggingImageForItems(at: Set<IndexPath>, with: NSEvent, offset: NSPointPointer) -> NSImage
```

Returns an image to use for dragging the specified items.

Legacy Collection View Support

~~`var itemPrototype: NSCollectionViewItem?`~~

The receiver's collection view item prototype.

Deprecated

~~`func newItem(forRepresentedObject: Any) -> NSCollectionViewItem`~~

Returns the collection view item that is used for the specified object.

Deprecated

`var selectionIndexes: IndexSet`

The indexes of the currently selected items.

~~`var maxNumberOfRows: Int`~~

The maximum number of rows that the collection view displays.

Deprecated

~~`var maxNumberOfColumns: Int`~~

The maximum number of columns that the collection view displays.

Deprecated

~~`var minItemSize: NSSize`~~

The minimum size (in points) of items in the collection view grid.

Deprecated

~~`var maxItemSize: NSSize`~~

The maximum size (in points) of items in the collection view grid.

Deprecated

`func item(at: Int) -> NSCollectionViewItem?`

Returns the collection view item for the represented object at the specified index.

`func frameForItem(at: Int) -> NSRect`

Returns the frame of the collection view item at the specified index.

`func frameForItem(at: Int, withNumberOfItems: Int) -> NSRect`

Returns the frame of an item based on the number of items in the collection view.

`func draggingImageForItems(at: IndexSet, with: NSEvent, offset: NSPoint Pointer) -> NSImage`

This method computes and returns an image to use for dragging.

`func setDraggingSourceOperationMask(NSDragOperation, forLocal: Bool)`

Configures the drag operation mask.

Constants

`enum DropOperation`

These constants specify if acceptance of a drop should be at the item it is dropped on or before the item. These constants are used by the `collectionView(_:acceptDrop:`

index:dropOperation:) and collectionView(_:validateDrop:proposedIndex:dropOperation:) methods in NSCollectionViewDelegate

`struct ScrollPosition`

Constants indicating the options for scrolling the collection view's content.

Type Aliases

`typealias DecorationElementKind`

Type Properties

`class let elementKindInterItemGapIndicator: String`

The element kind string assigned to the attributes object when it represents an inter-item gap.

`class let elementKindSectionFooter: String`

A supplementary view that acts as a footer for a given section.

`class let elementKindSectionHeader: String`

A supplementary view that acts as a header for a given section.

Enumerations

`enum ScrollDirection`

Constants indicating the scrolling direction for the layout.

`enum UpdateAction`

Constants indicating the type of action being performed on an item.

Relationships

Inherits From

NSView

Conforms To

CVarArg
CustomDebugStringConvertible
CustomStringConvertible
Equatable
Hashable
NSAccessibilityElementProtocol
NSAccessibilityProtocol
NSAnimatablePropertyContainer
NSAppearanceCustomization
NSCoding
NSDraggingDestination
NSDraggingSource
NSObjectProtocol
NSStandardKeyBindingResponding
NSTouchBarProvider
NSUserActivityRestoring
NSUserInterfaceItemIdentification
Sendable
SendableMetatype

See Also

View

`protocol` NSCollectionViewSectionHeaderView

A protocol that defines a button to control the collapse of a collection view's section.