

[UIKit](#) / [...](#) / [Layouts](#) / Customizing collection view layouts

## Sample Code

# Customizing collection view layouts

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

[Download](#)

iOS 12.0+ | iPadOS 12.0+ | Xcode 13.0+

## Overview

To lay out `UICollectionView` cells in a simple grid, you can use [UICollectionViewFlowLayout](#) directly. For more flexibility, you can subclass [UICollectionViewLayout](#) to create advanced layouts.

This sample app demonstrates two custom layout subclasses:

- `ColumnFlowLayout` — A `UICollectionViewFlowLayout` subclass that arranges cells in a list format for narrow screens, or as a grid for wider screens. See “For a Simple Grid, Size Cells Dynamically,” below.
- `MosaicLayout` — A `UICollectionViewLayout` subclass that lays out cells in a mosaic-style, nonconforming grid. See “For a Complex Grid, Define Cell Sizes Explicitly,” below.

The app opens to the Friends view controller, which uses a column flow layout to display a list of people. Tapping any cell takes you to the Feed view controller, which uses a mosaic layout to display photos from the user’s photo library.

Tapping the cloud icon to the right of the navigation bar demonstrates batched animations for inserting, deleting, moving, and reloading items in the collection view. For more information, see “Perform Batch Updates,” below. Using pull-to-refresh on the collection view resets the data.

## For a simple grid, size cells dynamically

`ColumnFlowLayout` is a subclass of `UICollectionViewFlowLayout` that uses the size of the collection view to determine the width of its cells. If only one cell fits comfortably horizontally, the layout arranges the cells to occupy the entire width of the collection view. Otherwise, the layout displays multiple columns of cells with a fixed width.

In practice, on iPhone devices in portrait mode, `ColumnFlowLayout` displays a single vertical column of cells. In landscape mode, or on an iPad, it displays a grid layout.

Use the `prepare()` function to compute the available screen width of the device and set the `itemSize` property accordingly.

```
override func prepare() {
    super.prepare()

    guard let collectionView = collectionView else { return }

    let availableWidth = collectionView.bounds.inset(by: collectionView.layoutMargin)
    let maxNumColumns = Int(availableWidth / minColumnWidth)
    let cellWidth = (availableWidth / CGFloat(maxNumColumns)).rounded(.down)

    self.itemSize = CGSize(width: cellWidth, height: cellHeight)
    self.sectionInset = UIEdgeInsets(top: self.minimumInteritemSpacing, left: 0.0, bottom: self.minimumInteritemSpacing, right: 0.0)
    self.sectionInsetReference = .fromSafeArea
}
```

## For a complex grid, define cell sizes explicitly

If you need more customization than is possible with a subclass of `UICollectionViewFlowLayout`, subclass `UICollectionViewLayout` instead.

`MosaicLayout` is a `UICollectionViewLayout` subclass that displays an arbitrary number of cells with differing sizes and aspect ratios. The `FeedViewController` class uses a mosaic layout to display images from the user's photo library. Cells are organized into rows in one of four styles, from a single cell to multiple cells in varying layouts.



Calculate cell dimensions

The `prepare()` method is called whenever a layout is invalidated. Override this method to calculate the position and size of every cell, as well as the total dimensions for the entire layout.

```
override fun prepare() {
    super.prepare()

    guard let collectionView = collectionView else { return }

    // Reset cached information.
    cachedAttributes.removeAll()
    contentBounds = CGRect(origin: .zero, size: collectionView.bounds.size)

    // For every item in the collection view:
    // - Prepare the attributes.
    // - Store attributes in the cachedAttributes array.
    // - Combine contentBounds with attributes.frame.
    let count = collectionView.numberOfItems(inSection: 0)

    var currentIndex = 0
    var segment: MosaicSegmentStyle = .fullWidth
    var lastFrame: CGRect = .zero

    let cvWidth = collectionView.bounds.size.width

    while currentIndex < count {
        let segmentFrame = CGRect(x: 0, y: lastFrame.maxY + 1.0, width: cvWidth, height: 1.0)

        var segmentRects = [CGRect]()
        switch segment {
            case .fullWidth:
                segmentRects = [segmentFrame]

            case .fiftyFifty:
                let horizontalSlices = segmentFrame.dividedIntegral(fraction: 0.5, from: 0.0, to: 1.0)
                segmentRects = [horizontalSlices.first, horizontalSlices.second]

            case .twoThirdsOneThird:
                let horizontalSlices = segmentFrame.dividedIntegral(fraction: (2.0 / 3.0), from: 0.0, to: 1.0)
                let verticalSlices = horizontalSlices.second.dividedIntegral(fraction: 0.5, from: 0.0, to: 1.0)
                segmentRects = [horizontalSlices.first, verticalSlices.first, verticalSlices.second]

            case .oneThirdTwoThirds:
                let horizontalSlices = segmentFrame.dividedIntegral(fraction: (1.0 / 3.0), from: 0.0, to: 1.0)
                let verticalSlices = horizontalSlices.second.dividedIntegral(fraction: 0.5, from: 0.0, to: 1.0)
                segmentRects = [horizontalSlices.first, verticalSlices.first, verticalSlices.second]
```

```

        let horizontalSlices = segmentFrame.dividedIntegral(fraction: (1.0 / 3.0))
        let verticalSlices = horizontalSlices.first.dividedIntegral(fraction: 0.5)
        segmentRects = [verticalSlices.first, verticalSlices.second, horizontalSlices.first, horizontalSlices.second]
    }

    // Create and cache layout attributes for calculated frames.
    for rect in segmentRects {
        let attributes = UICollectionViewLayoutAttributes(forCellWith: IndexPath(item: count - 1, section: 0))
        attributes.frame = rect

        cachedAttributes.append(attributes)
        contentBounds = contentBounds.union(lastFrame)

        currentIndex += 1
        lastFrame = rect
    }

    // Determine the next segment style.
    switch count - currentIndex {
    case 1:
        segment = .fullWidth
    case 2:
        segment = .fiftyFifty
    default:
        switch segment {
        case .fullWidth:
            segment = .fiftyFifty
        case .fiftyFifty:
            segment = .twoThirdsOneThird
        case .twoThirdsOneThird:
            segment = .oneThirdTwoThirds
        case .oneThirdTwoThirds:
            segment = .fiftyFifty
        }
    }
}
}

```

## Provide the content size

Override the [collectionViewContentSize](#) property, providing a size for the collection view.

```
override var collectionViewContentSize: CGSize {  
    return contentBounds.size  
}
```

## Define the layout attributes

Override `layoutAttributesForElements(in:)`, defining the layout attributes for a geometric region. The collection view calls this function periodically to display items, which is known as *querying by geometric region*.

```
override func layoutAttributesForElements(in rect: CGRect) -> [UICollectionViewLayout  
    var attributesArray = [UICollectionViewLayoutAttributes]()  
  
    // Find any cell that sits within the query rect.  
    guard let lastIndex = cachedAttributes.indices.last,  
        let firstMatchIndex = binSearch(rect, start: 0, end: lastIndex) else { return attributesArray }  
  
    // Starting from the match, loop up and down through the array until all the attributes  
    // have been added within the query rect.  
    for attributes in cachedAttributes[..<firstMatchIndex].reversed() {  
        guard attributes.frame.maxY >= rect.minY else { break }  
        attributesArray.append(attributes)  
    }  
  
    for attributes in cachedAttributes[firstMatchIndex...] {  
        guard attributes.frame.minY <= rect.maxY else { break }  
        attributesArray.append(attributes)  
    }  
  
    return attributesArray  
}
```

Also provide the layout attributes for a specific item by implementing `layoutAttributesForItem(at:)`. The collection view calls this function periodically to display one particular item, which is known as *querying by index path*.

```
override func layoutAttributesForItem(at indexPath: IndexPath) -> UICollectionViewLayoutAttributes {  
    return cachedAttributes[indexPath.item]  
}
```

Because these functions are called often, they can affect the performance of your app. To make them as efficient as possible, follow the example code as closely as you can.

## Handle bounds changes

The `shouldInvalidateLayout(forBoundsChange:)` function is called for every bounds change from the collection view, or whenever its size or origin changes. This function is also called frequently during scrolling. The default implementation returns `false`, or, if the size and origin change, it returns `true`.

```
override fun shouldInvalidateLayout(forBoundsChange newBounds: CGRect) -> Bool {  
    guard let collectionView = collectionView else { return false }  
    return !newBounds.size.equalTo(collectionView.bounds.size)  
}
```

For optimum performance, this sample performs a binary search inside `layoutAttributesForElements(in:)` instead of a linear search of the attributes it needs for each element in a given bounds area.

## Perform batch updates

Tapping the top-right button in the navigation bar triggers the collection view to perform a *batch update* of multiple animated operations (insert, delete, move, and reload) of its collection view cells all at the same time.

Within a call to `performBatchUpdates(_:completion:)`, the system simultaneously animates all insert, delete, move, and reload operations. In this sample, the app batches updates by processing an array of `PersonUpdate` objects, each of which encapsulates one update:

- insert with a `Person` object and insertion index.
- delete with an index.
- move from one index to another.
- reload with an index.

First, the reload operations are performed without animation because no cell movement is involved:

```
// Perform any cell reloads without animation because there is no movement.  
UIView.performWithoutAnimation {  
    collectionView.performBatchUpdates({  
        for update in remoteUpdates {  
            if case let .reload(index) = update {
```

```

        people[index].isUpdated = true
        collectionView.reloadItems(at: [IndexPath(item: index, section: 0)])
    }
}
}))
}

```

Next, the remaining operations are animated:

```

// Animate all other update types together.
collectionView.performBatchUpdates({
    var deletes = [Int]()
    var inserts = [(person:Person, index:Int)]()

    for update in remoteUpdates {
        switch update {
        case let .delete(index):
            collectionView.deleteItems(at: [IndexPath(item: index, section: 0)])
            deletes.append(index)

        case let .insert(person, index):
            collectionView.insertItems(at: [IndexPath(item: index, section: 0)])
            inserts.append((person, index))

        case let .move(fromIndex, toIndex):
            // Updates that move a person are split into an addition and a deletion.
            collectionView.moveItem(at: IndexPath(item: fromIndex, section: 0),
                                     to: IndexPath(item: toIndex, section: 0))
            deletes.append(fromIndex)
            inserts.append((people[fromIndex], toIndex))

        default: break
        }
    }

    // Apply deletions in descending order.
    for deletedIndex in deletes.sorted().reversed() {
        people.remove(at: deletedIndex)
    }

    // Apply insertions in ascending order.
    let sortedInserts = inserts.sorted(by: { (personA, personB) -> Bool in

```

```
        return personA.index <= personB.index
    })
    for insertion in sortedInserts {
        people.insert(insertion.person, at: insertion.index)
    }

    // The update button is enabled only if the list still has people in it.
    navigationItem.rightBarButtonItem?.isEnabled = !people.isEmpty
}
```

## See Also

### Manual layouts

`class UICollectionViewLayout`

An abstract base class for generating layout information for a collection view.

`class UICollectionViewFlowLayout`

A layout object that organizes items into a grid with optional header and footer views for each section.

`class UICollectionViewTransitionLayout`

A special type of layout object that lets you implement behaviors when changing from one layout to another in your collection view.

`class UICollectionViewLayoutAttributes`

A layout object that manages the layout-related attributes for a given item in a collection view.

`class UICollectionViewFlowLayoutInvalidationContext`

A set of properties for determining whether to recompute the size of items or their position in the layout.