

iOS Dev Accelerator

Week 2 Day 4

- Photos framework
- Custom Protocol
- UICollectionView Layout
- Gesture Recognizers
- Social Sharing

Photos Framework

Photos Framework

- New Apple Framework that allows access to photos and videos from the photo library.
- Also used for creating photo editing app extensions, a new feature with iOS8
- First-class citizen, you can create a full-featured photo library browser and editor on par with Apple's Photos App.
- Intended to supersede ALAssetsLibrary
- It is highly asynchronous!

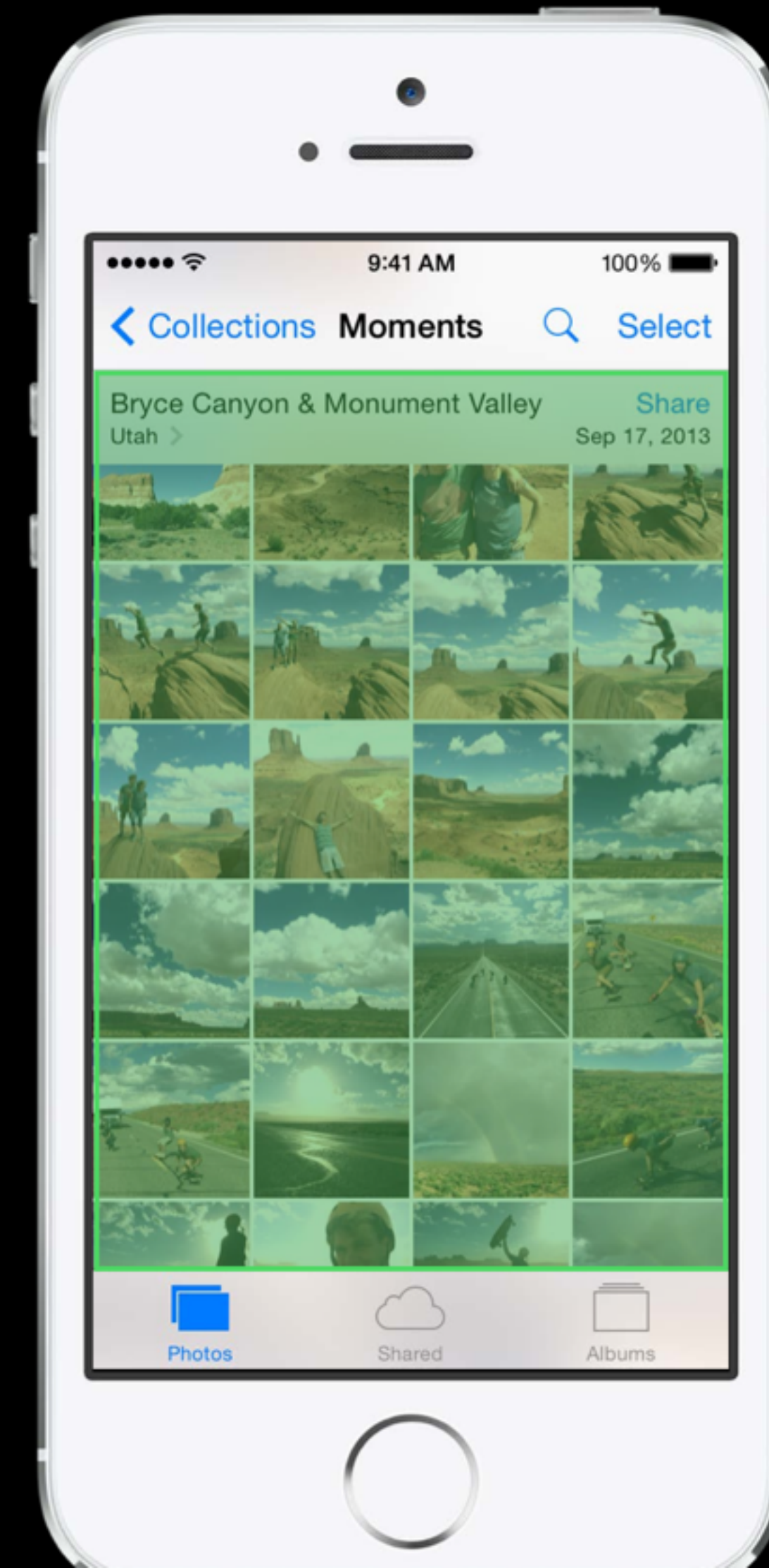
PHAsset

- The Photos framework model object that represents a single photo or video.
- Has properties for Media type, Creation date, Location, and Favorite.



PHAssetCollection

- A Photos framework model object representing an ordered collection of assets.
- Albums, moments, and smart albums.
- Has properties for Type, Title, and Start and End Date.



PHCollectionList

- A Photos framework model object representing an ordered collection of collections.
- This can be a folder, moment, or year
- Has properties for Type, Title, and Start and End Date.



Fetching Model Objects

- You fetch via class methods on the models:
 - `PHAsset.fetchAssetsWithMediaType(PHAssetMediaType.Photo, options:nil)`
 - `PHAssetCollection.fetchMomentsWithOptions(nil)`
- Collections do not cache their contents, so you still have to fetch all the assets inside of it. This is because the results of a fetch can be very large, and you don't want all of those objects in memory at once.

PHFetchResult

- Results returned in a PHFetchResult
- Similar to an Array.



Making Changes

- You can favorite a photo and add an asset to an album
- You cannot directly mutate an asset, they are read only (thread safe!)
- To make a change, you have to make a change request.
- There's a request class for each model class:

PHAssetChangeRequest

PHAssetCollectionChangeRequest

PHCollectionListChangeRequest

Making Changes

```
func toggleFavorite(asset : PHAsset) {  
    PHPhotoLibrary.sharedPhotoLibrary().performChanges({  
        //create a change request object for the asset  
        var changeRequest = PHAssetChangeRequest(forAsset: asset) as  
        PHAssetChangeRequest  
        //make your change  
        changeRequest.favorite = !changeRequest.favorite  
  
    }, completionHandler: { ( success : Bool,error : NSError!) -> Void in  
  
        //asset change complete  
    })  
}
```

Making New Objects

Create via creation request

```
var request = PHAssetChangeRequest.creationRequestForAssetFromImage(UIImage())
```

Placeholder objects

```
var placeholder = request.placeholderForCreatedAsset
```

- Reference to a new, unsaved object
- Add to collections
- Can provide unique, persistent **localIdentifier**

Getting to the actual data

- Many different sizes of an image may be available or different formats of a video
- Use `PHImageManager` to request images/videos
- Request an image based on target size for displaying
- Request a video based on the usage
- Asynchronous API, because you don't know how long it will take to load the data, it could be very expensive
- Will optionally retrieve the data from the network if it's only on iCloud
- Use a `PHCachingImageManager` when displaying a collection of images for better performance.

Requesting an Image

```
let manager = PHImageManager.defaultManager()

manager.requestImageForAsset(photo,
    targetSize: cellSize,
    contentMode: PHImageContentMode.AspectFill,
    options: nil,
    resultHandler: {(result : UIImage!, [NSObject : AnyObject]!) -> Void in
        if result {
            imageView.image = result
        } else {
            //tell user something went wrong
        }
    })
```


Advanced Image Request

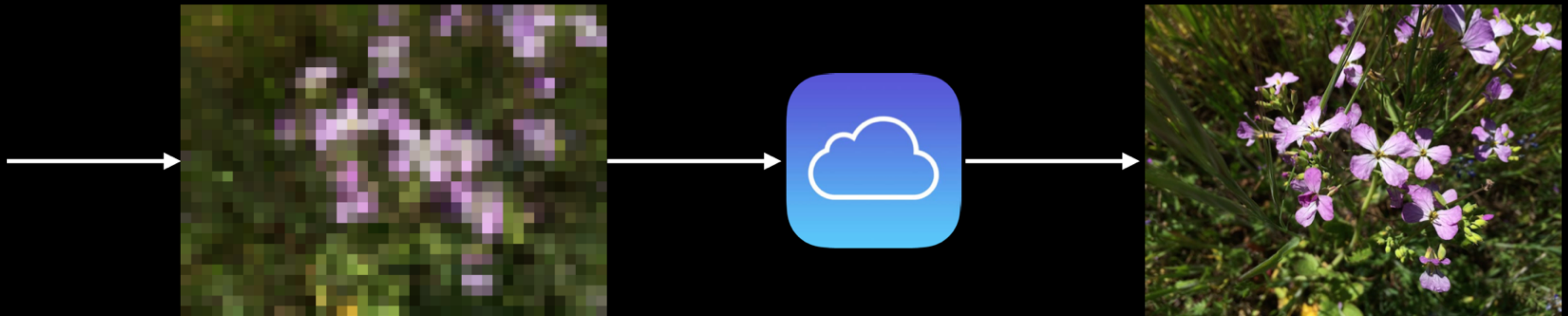
```
var options = PHImageRequestOptions()

options.networkAccessAllowed = true
options.progressHandler = {(progress : Double, error : NSError!, degraded : UnsafePointer<ObjCBool>,
[NSObject : AnyObject]!) in
    //update visible progress UI
}

//use your options to control the request behavior
manager.requestImageForAsset(photo,
    targetSize: cellSize,
    contentMode: PHImageContentMode.AspectFill,
    options: options,
    resultHandler: {(result : UIImage!, [NSObject : AnyObject]!) -> Void in
```

Advanced Image Request

```
[manager requestImageForAsset: ... ^(UIImage *result, NSDictionary *info) {  
    // This block can be called multiple times  
}];
```



First callback synchronous

Second callback asynchronous

Demo

Custom Protocols

Protocols

- In the real world, people are often required to follow strict procedures in certain situations.
- For example, firefighters are supposed to follow a specific protocol during emergencies.
- In the world of object oriented programming, its critical to be able to define a set of behaviors that is expected of an object in certain situations.
- This is what a protocol is used for.

Protocols

- Table views are an example that we have already used in our apps.
- A table view expects to be able to communicate with a data source object in order to find out what it is required to display.
- This means that which ever object is the data source must respond to specific messages
- **The datasource could be an instance of any class**, such as a UIViewController or some custom data source class, but its important that it implements the required methods to function as a table view datasource
- Swift & Objective-C allow you to create protocols which declare methods expected to be used in a particular situation.
- This is similar to interfaces in Java

Declaring a Protocol

- Declaring protocols is rather simple in Swift:

```
protocol RandomNumberGenerator {  
    func random() -> Double  
}
```

- Above is an example of a protocol called RandomNumberGenerator
- It has one required method, its called random and it returns a Double.
- So any class that conforms to this protocol must have a method called RandomNumberGenerator that return a Double.
- It is important to remember that protocols don't define the implementation of these methods. That is up to the object that conforms to the protocol!

Declaring a Protocol

- Protocols can also declare properties that it expects its conformers to have:

```
protocol SomeProtocol {  
    var mustBeSettable: Int { get set }  
    var doesNotNeedToBeSettable: Int { get }  
}
```

- Properties in a protocol must be designated as read write or read only, as seen above
- Methods and properties declared in a protocol can be marked as optional:

**@objc tag required
to have optional things
in your protocol**

```
@objc protocol CounterDataSource {  
    optional func incrementForCount(count: Int) ->  
        Int  
    optional var fixedIncrement: Int { get }  
}
```

Conforming to a protocol

- Conforming to a protocol is also a pretty simple operation.
- Here is an example of a protocol called FullyNamed and then an example of a struct that conforms to it:

```
protocol FullyNamed {  
    var fullName: String { get }  
}
```

```
struct Person: FullyNamed {  
    var fullName: String  
}
```


Best practice delegate method convention

- Delegation methods should begin with the name of object doing the delegating — application, control, controller, etc.
- The name is then followed by a verb of what just occurred — willSelect, didSelect, openFile, etc.
- For example, our protocol will be called `imageSelectedDelegate` and the method we will define in it will be called `controllerDidSelectImage`

Creating a delegate property

- Once your protocol is setup, you need to add a delegate property to whatever class is going to have the delegate:

```
var delegate : ImageSelectedDelegate?
```



the delegate property's type is the protocol. This basically just says this property can be set to any type as long as it conforms to this protocol

Making Delegates Weak

- A delegate property should always be weak.
- The delegator should never own the delegate, because if it does, it will probably be a retain cycle.
- To make a protocol-property weak, you must make the protocol inherit from class

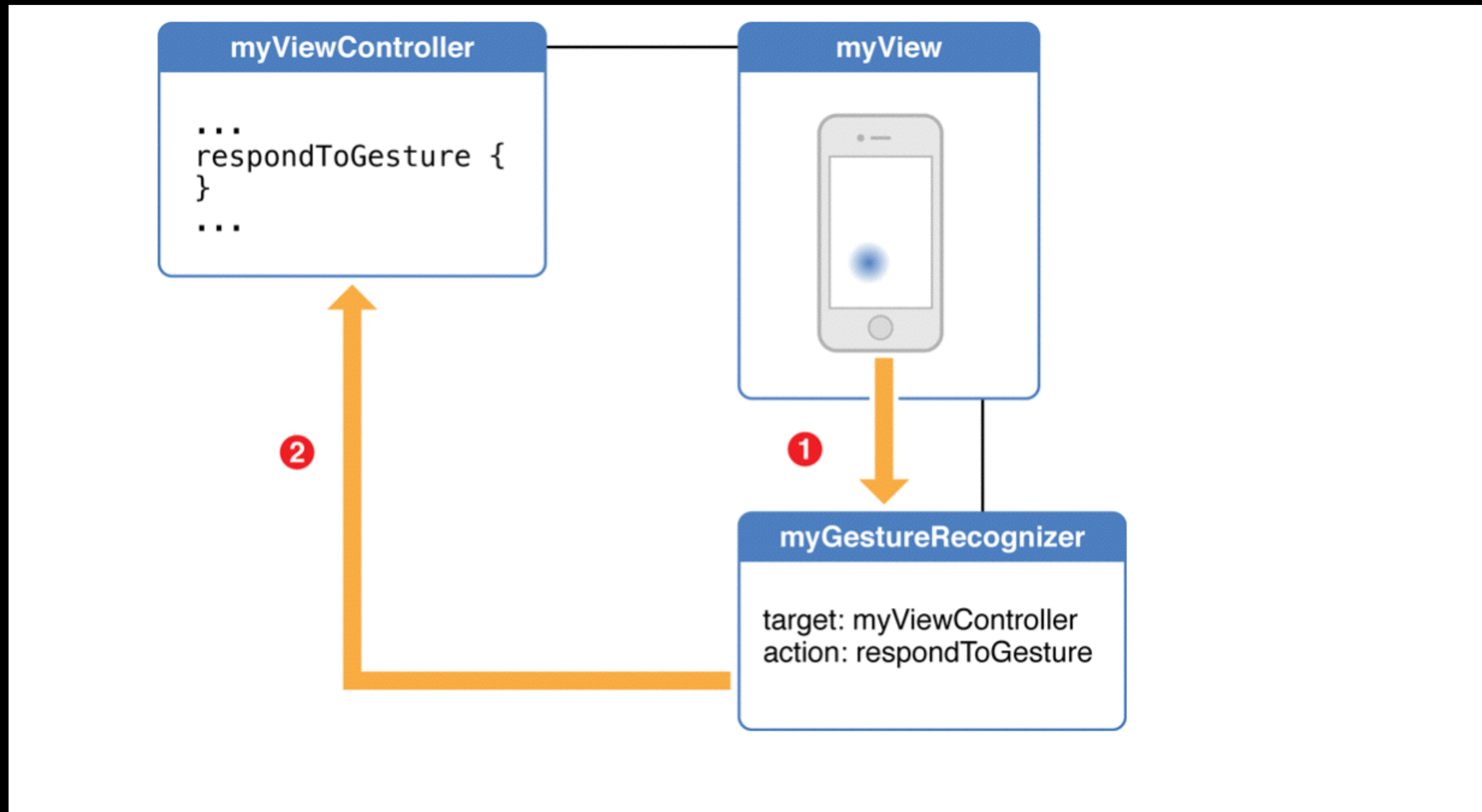
Demo

Gesture Recognizers

Gesture Recognizers

- “Gesture Recognizers convert low level event handling code into higher level actions”
- Gesture recognizers are attached to views.
- If the desired gesture is detected on the view the recognizer is attached to, an action message is sent to a target object.
- The target is usually the view’s view controller.

Gesture Recognizers



Predefined vs Custom Gesture Recognizers

- UIKit has a good amount of predefined gesture recognizers that you should always use when possible.
- It is much more simple to use one of their recognizers vs implementing your own.
- If your app needs to recognize a custom gesture, like a figure 8 or checkmark, you will need to implement your own custom gesture recognizer.

Built-in Gesture Recognizers

- UITapGestureRecognizer - any number of taps
- UIPinchGestureRecognizer - pinch in and out for zooming
- UIPanGestureRecognizer - panning or dragging
- UISwipeGestureRecognizer - swiping in any direction
- UIRotationGestureRecognizer - finger moving in opposite direction
- UILongPressGestureRecognizer - touch and hold for a certain amount of time
- Refer to the HIG for recommended usage for each type of gesture

Discrete vs Continuous Gestures

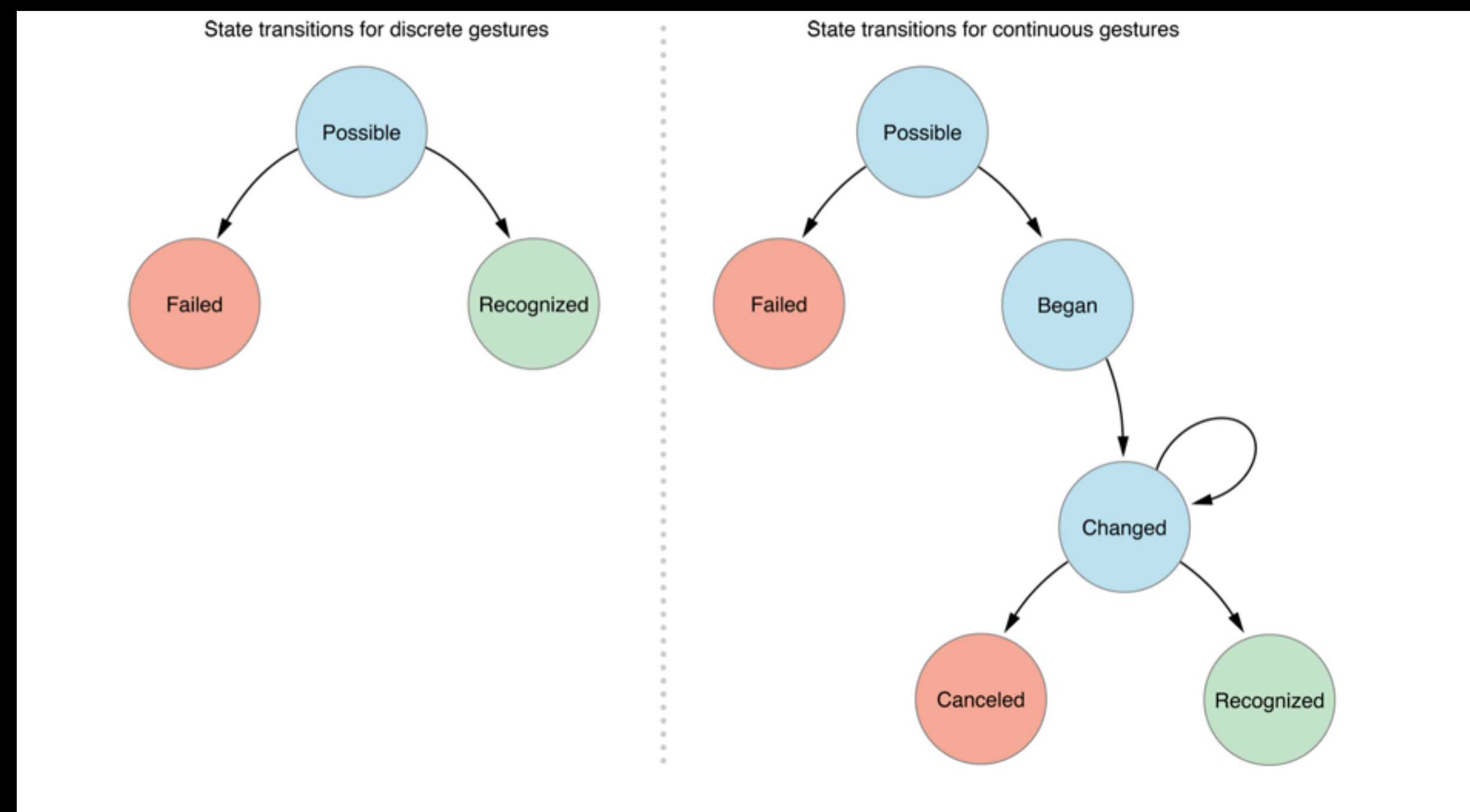
- Gestures are either discrete or continuous.
- A discrete gesture only happens once it is detected. Like a tap.
- A continuous gesture takes place over time, like a pan.
- If it is discrete, only one action message is sent. If it is continuous, many action messages are sent until the gesture is over.

Gesture Recognizer Setup

1. Create and configure a gesture recognizer instance. Either in code or in storyboard. If its storyboard, this includes step 2.
2. Attach the gesture recognizer to a view.
3. Implement the action method that handles the gesture.

Gesture Recognizer State

- Gesture Recognizers transition from one state to another in a predefined way.
- From each state, they can move to one of several possible next states based on whether they meet certain conditions:



Demo

CollectionView
FlowLayout

UICollectionViewLayout

- Computes layout attributes as needed for:
- CollectionView Cells
- CollectionView Supplementary Views
- Decoration Views

UICollectionViewLayout

- Every collection view uses a layout object to determine where each view it manages should be placed and behave on screen.
- Apple provides a concrete subclass of UICollectionViewLayout called UICollectionViewFlowLayout that gives us a line based layout that we can use right out of the box.
- A collection view's layout is highly customizable. When you want to create a custom layout, you first need to determine if it is suitable for you to subclass flow layout (less work), or create a brand new subclass of UICollectionViewLayout (more work).

UICollectionViewLayoutAttributes

- Manages the following layout-related attributes for a given item in a collection view:
 - Position
 - Size
 - Opacity
 - zIndex (overlapping cells, above or below)
 - Transforms
- One attribute instance per view!

UICollectionViewFlowLayout

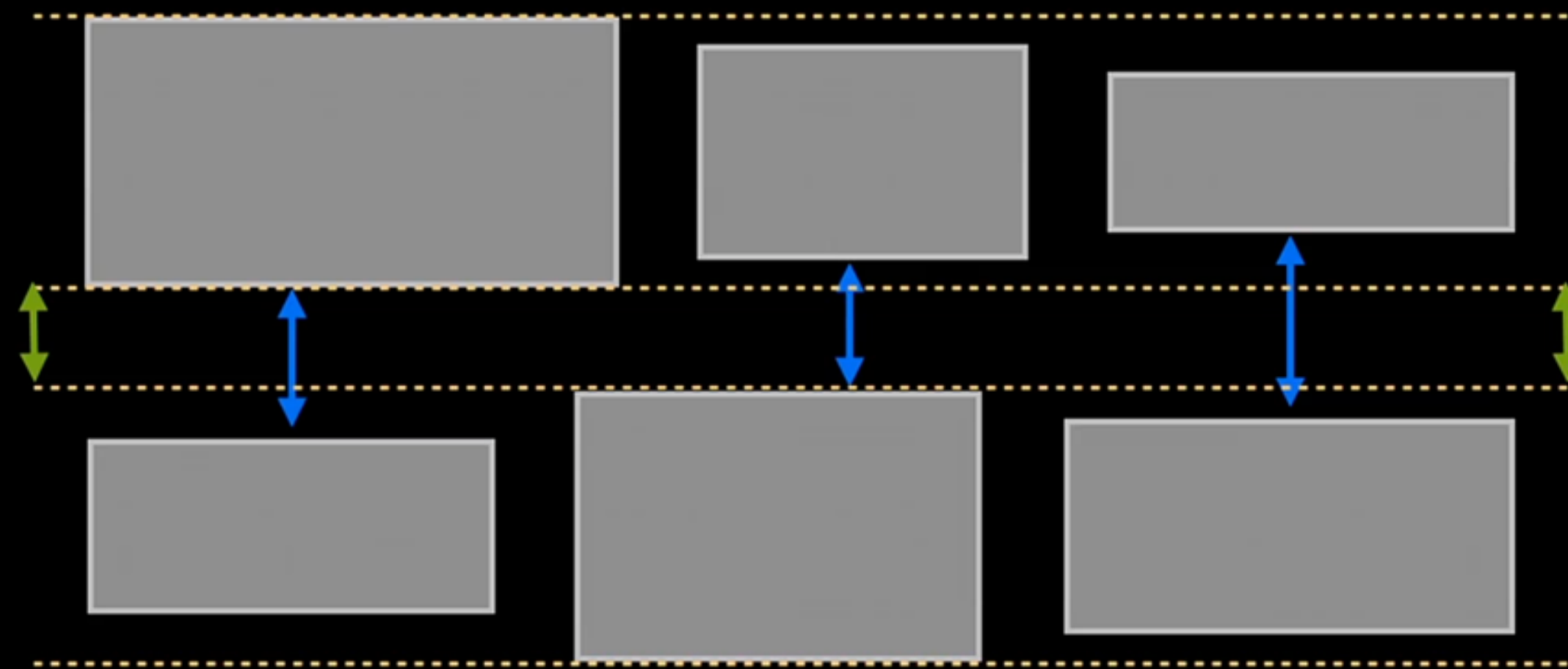
- Flow layout is a line-oriented layout. The layout object places cells on a linear path and fits as many cells as it can along the line. When the line runs out of room, it creates a new line and continues the process.
- Can be configured as a grid or as a group of lines.
- Out of the box, it has lots of things you can customize:
 - Item Size
 - Line Spacing and Inter Cell spacing
 - Scrolling direction
 - Header and footer size
 - Section Inset
- And you customize each of those things either globally with a single property, or through a delegate

Item Size

- The item size for each cell can be set globally by setting the `itemSize` property on your flow layout.
- Or if you want different size per item, you can do it through the delegate method `collectionView:layout:sizeForItemAtIndexPath()`

Line Spacing

- You can set a minimum line spacing, either globally or through the delegate:



Minimum line spacing



Actual line spacing

Inter-item Spacing

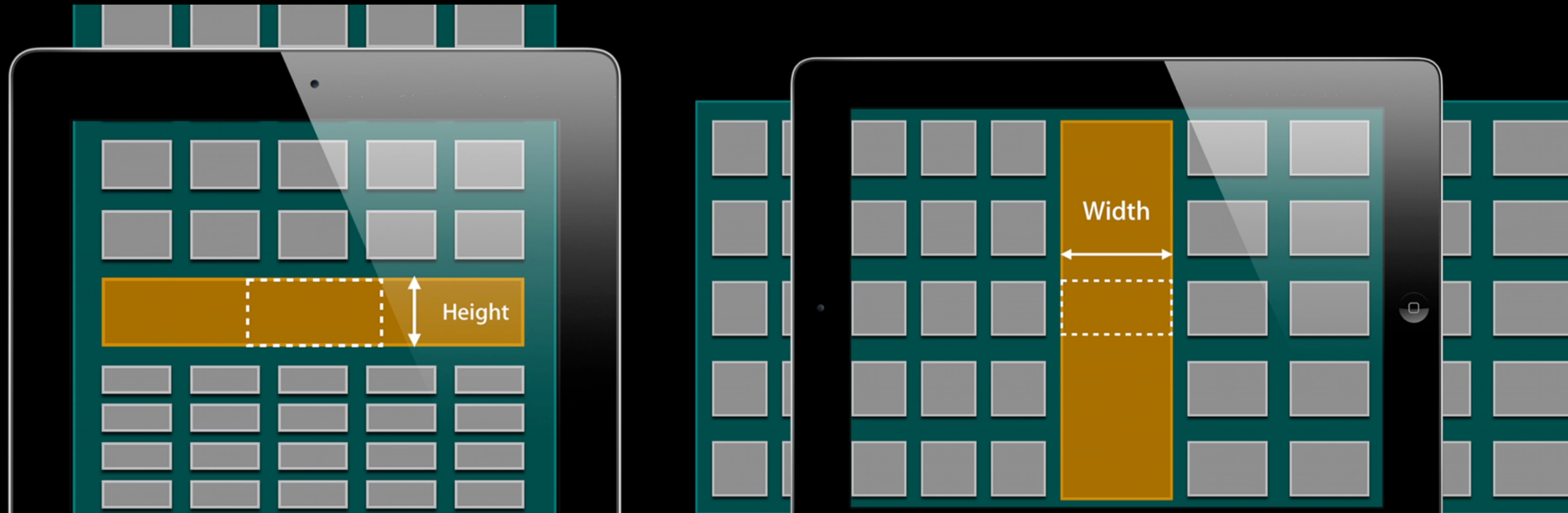
- Same with spacing between individual items:



↔ Actual interitem spacing
↔ Minimum interitem spacing

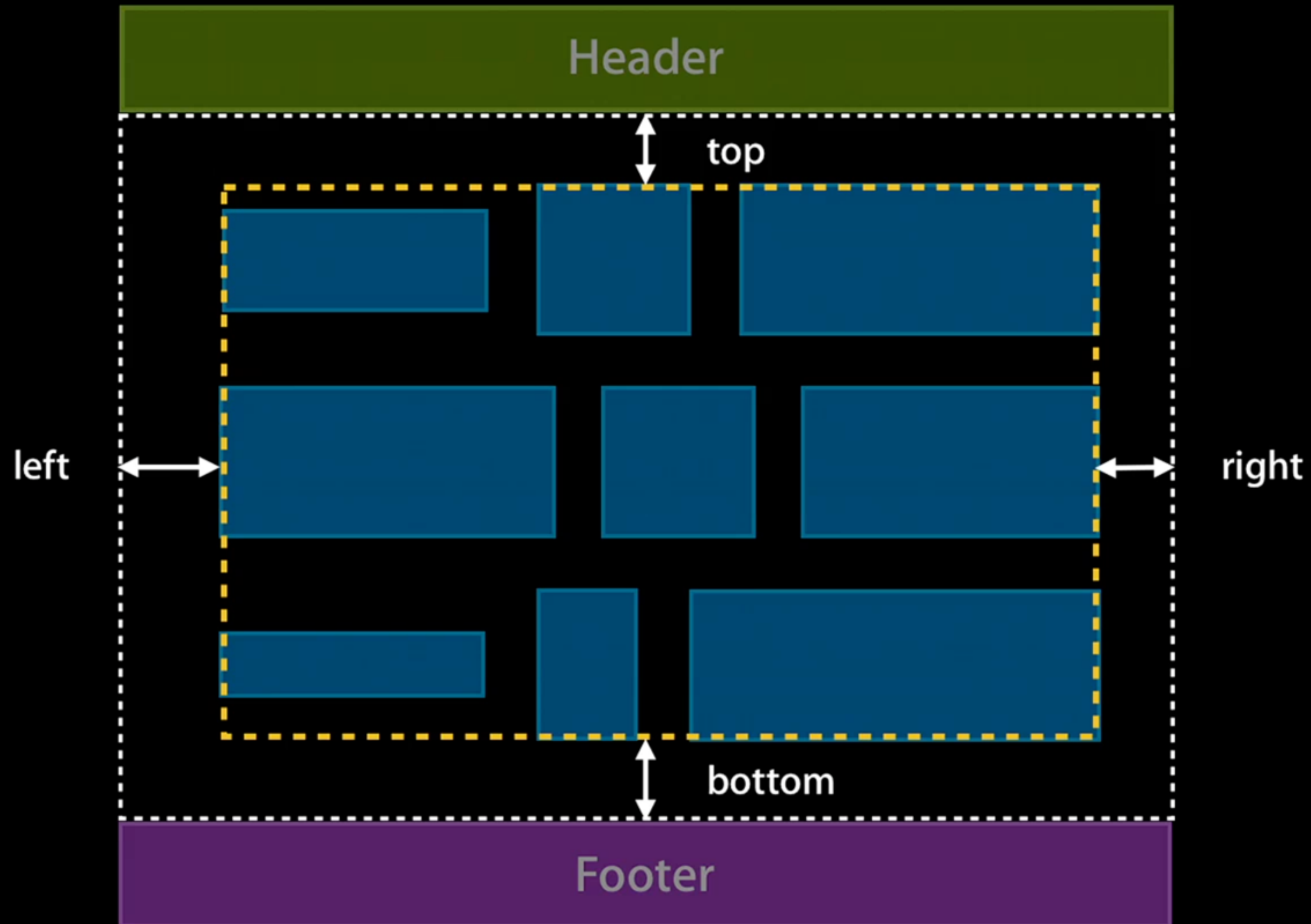
Scrolling Direction

- The scroll direction of your flow layout can defines the base behavior of your entire flow layout
- Dictates the dimensions of the header and footer views:



Section Insets

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



Changing the layout

- When you want your layout to change, you need to invalidate your layout.
- You can call `invalidateLayout` to trigger a layout update.
- You can use `performBatchUpdates:completion:` and anything you change inside the update block will invalidate the layout AND cause awesome animations.
- Whenever the bounds of the collection view changes, the layout is invalidated (rotation, scrolling)

Demo

When to go custom?

- If you are constantly changing the location of all the cells.
- Basically, if your collection view doesn't resemble a grid, its time to go custom.

Required overrides on UICollectionViewLayout

- `collectionViewContentSize`: Returns the width and height of the collection view's contents. **This is the entire size of the collection view's content, not just what is visible.**
- `layoutAttributesForElementsInRect`: Returns the layout information for the cells and views that intersect the specified rectangle. **In order for the collection view to know which attribute goes to cells or views, you must specify the `elementCategory` on the attribute (cell, supplementary view, decoration view)** **This is constantly called, every time the user scrolls the collection view. Yikes.**
- `layoutAttributeForItemAtIndexPath`: Use this method to provide layout information for your collection view's cells. Do not use this method for supplementary or decoration views.
- `layoutAttributesForSupplementaryViewOfKind:atIndexPath:` &
`layoutAttributesForDecorationViewWithReuseIdentifier:atIndexPath:` same as above but for supplementary views

UICollectionViewLayout Order of Operations

1. `prepareLayout`
2. `collectionViewContentSize`
3. `layoutAttributeForElementsInRect` (which will probably call `layoutAttributesForIndexPath`)

If the layout is invalidated, `prepareLayout` is called and this cycle is repeated.

Social Sharing

SLComposeController

- SLComposerController class presents a view to the user to compose a post for the supported social networking services
- First check if the service type(s) you are going to offer are available on the user's device (aka they are signed in) by calling `isAvailableForServiceType`
- The available service types are facebook, twitter, weibo, and tencentWeibo.

SLComposeController Workflow

1. Check if the service type is available
2. instantiate an SLCompViewController object, and use the init that takes in a SLServiceType
3. Add whatever image or URL you are going to share if you have them.
4. Add a completionHandler of type (SLComposeViewControllerResult) -> (Void) (this is optional)
5. Present the view controller