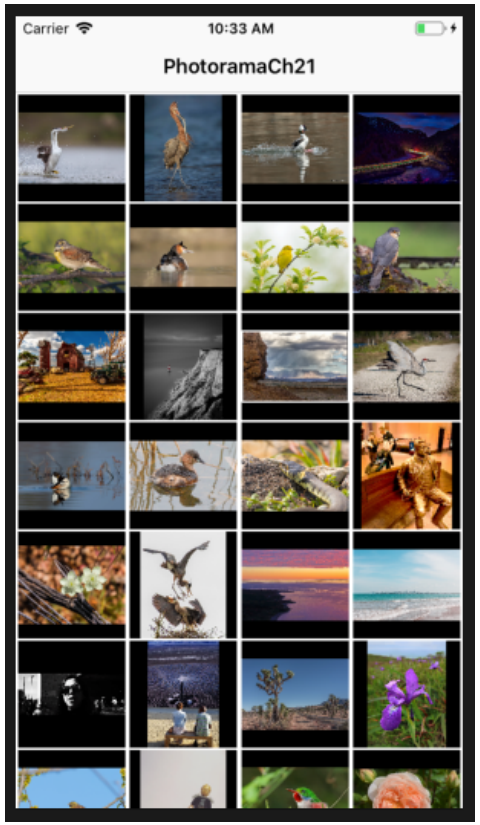
CS275
Fall 2020

iOS: Collection Views

BNRG CHAPTER 21

Photorama: Chapters 20-23

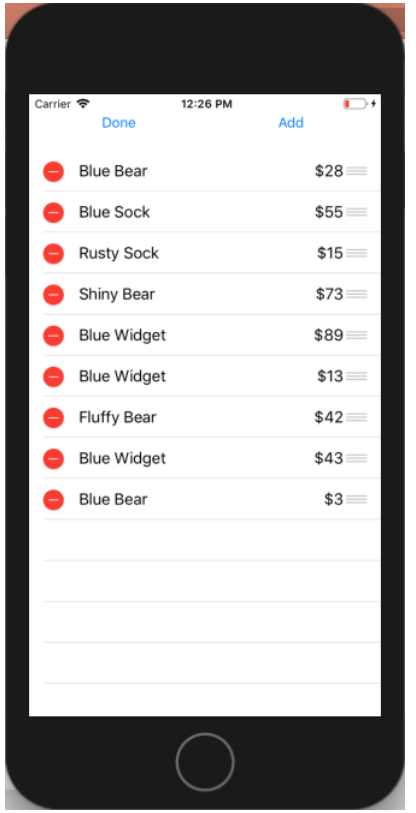
- This is a complex app
- Chapter 20: web services, JSON, threads
- Chapter 21: collection views
- Chapters 22-23: persistence, through Core Data



Collection View

Collection View

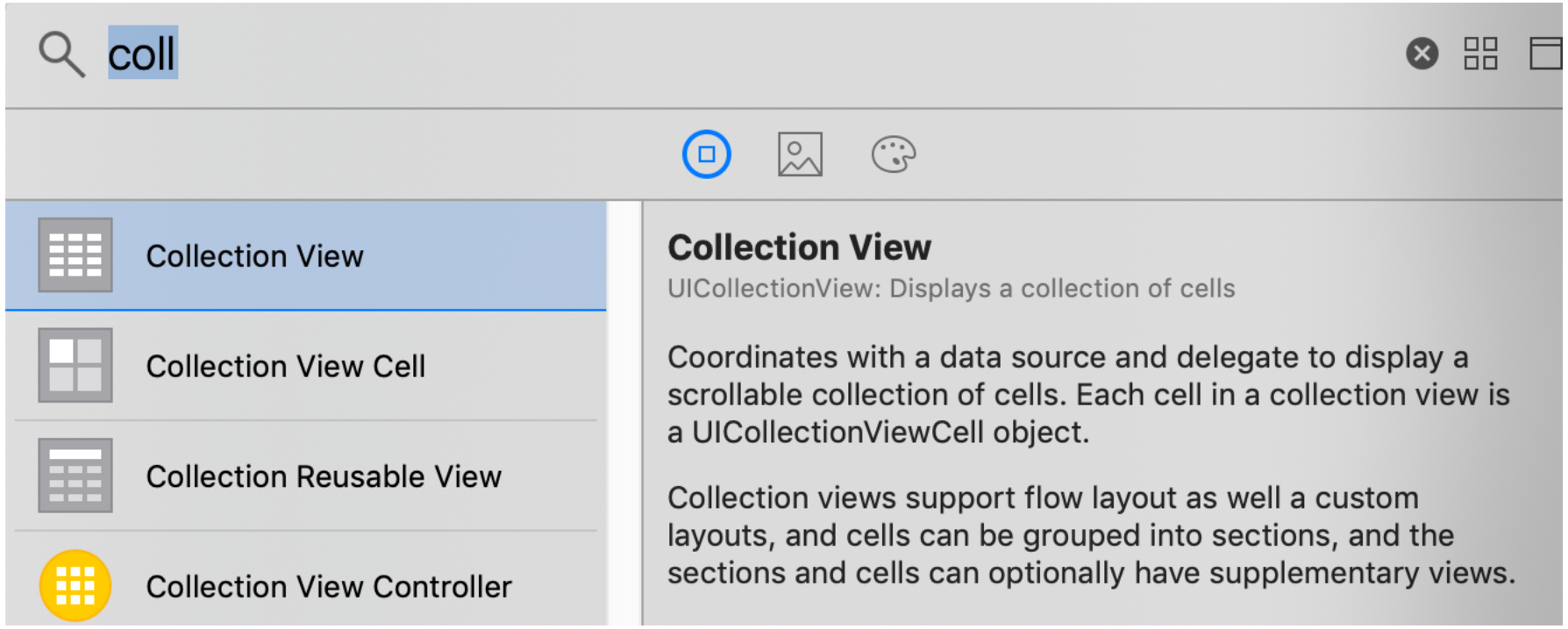
- Table view is one way to display a set of related information
- Collection view is another way
- Instead of a hierarchical list, the collection view uses a layout object to guide the presentation of information
- `UICollectionViewFlowLayout`: a scrollable (two-dimensional) grid



UITableView

Collection View

From the Library in Interface Builder



Collection View Data Source

A Collection View requires a source for the data that will be displayed

- the goal is abstraction: put the data in its own class—not in the view controller

The best way to do this: create a new class to serve as the source of the data

- for PhotoRama, the class will be called `PhotoDataSource`

The class must conform to `UICollectionViewDataSource`

- in order to conform to `UICollectionViewDataSource`, the class must also conform to `NSObjectProtocol`
- and the easiest way to accomplish this is to subclass `NSObject`

`UICollectionViewDataSource` Protocol

This protocol requires us to implement these two methods:

1. `func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int`
   - the view asks how many cells to display

2. `func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell`
   - the view asks for the `UICollectionViewCell` to display for a given index path
   - index path: for hierarchical data—has a section and a row

Recall: `UITableViewDataSource` requires similar methods

Connecting the UI and the Code

An outlet for the collection view:

```
class PhotosViewController: UIViewController {
    @IBOutlet var collectionView: UICollectionView!

    var store: PhotoStore!
    let photoDataSource = PhotoDataSource()

    override func viewDidLoad() {
        super.viewDidLoad()
        collectionView.dataSource = photoDataSource
        ...
    }
    ...
}
```

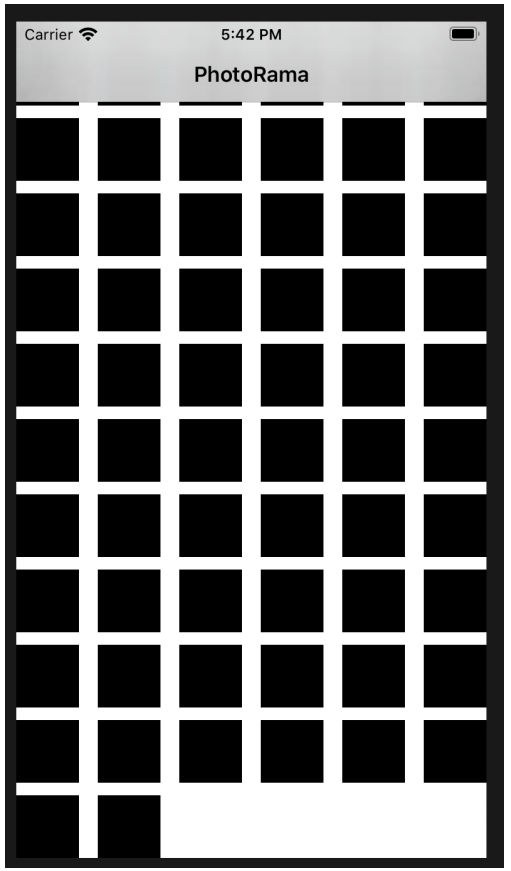
UICollectionViewFlowLayout

Built-in layout that takes views and places them in the collection view by letting them "flow" across the screen and then down to the next row

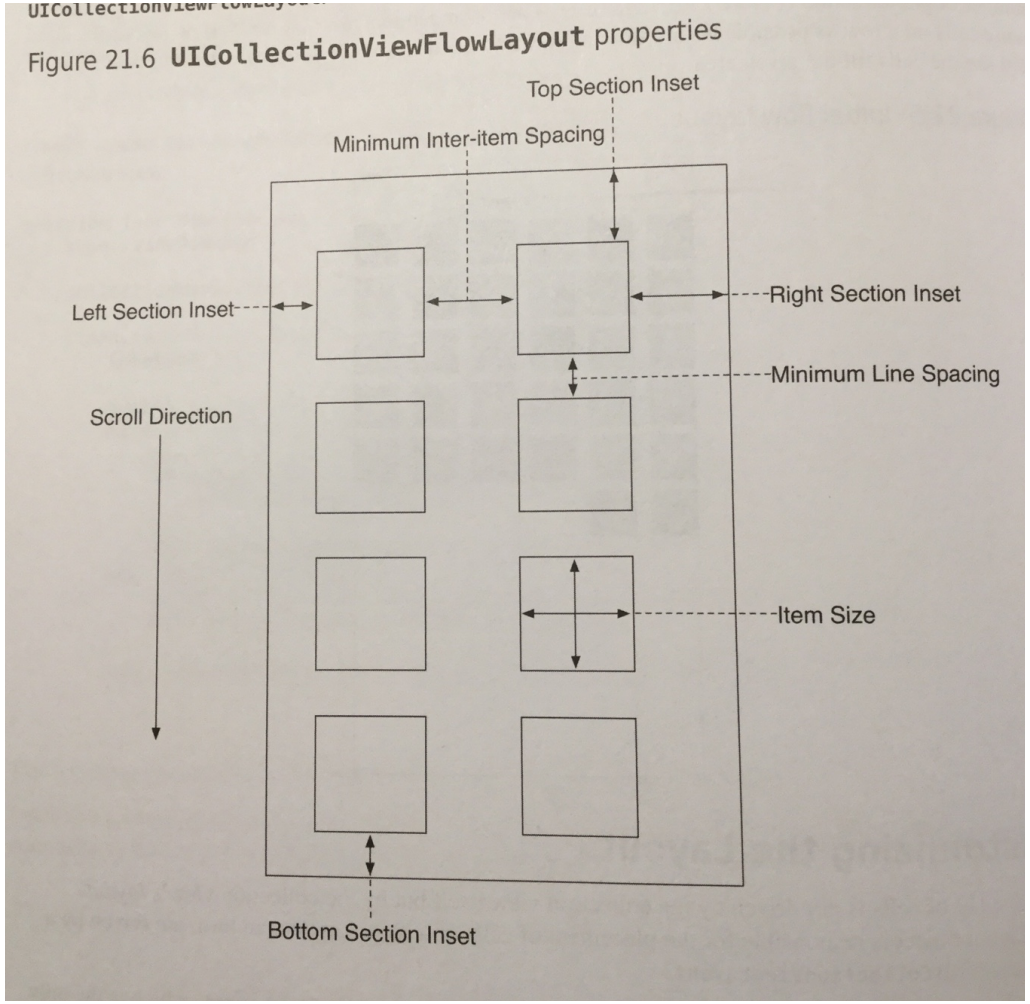
- each view is indexed as 0, 1, 2, ...

Properties we can set:

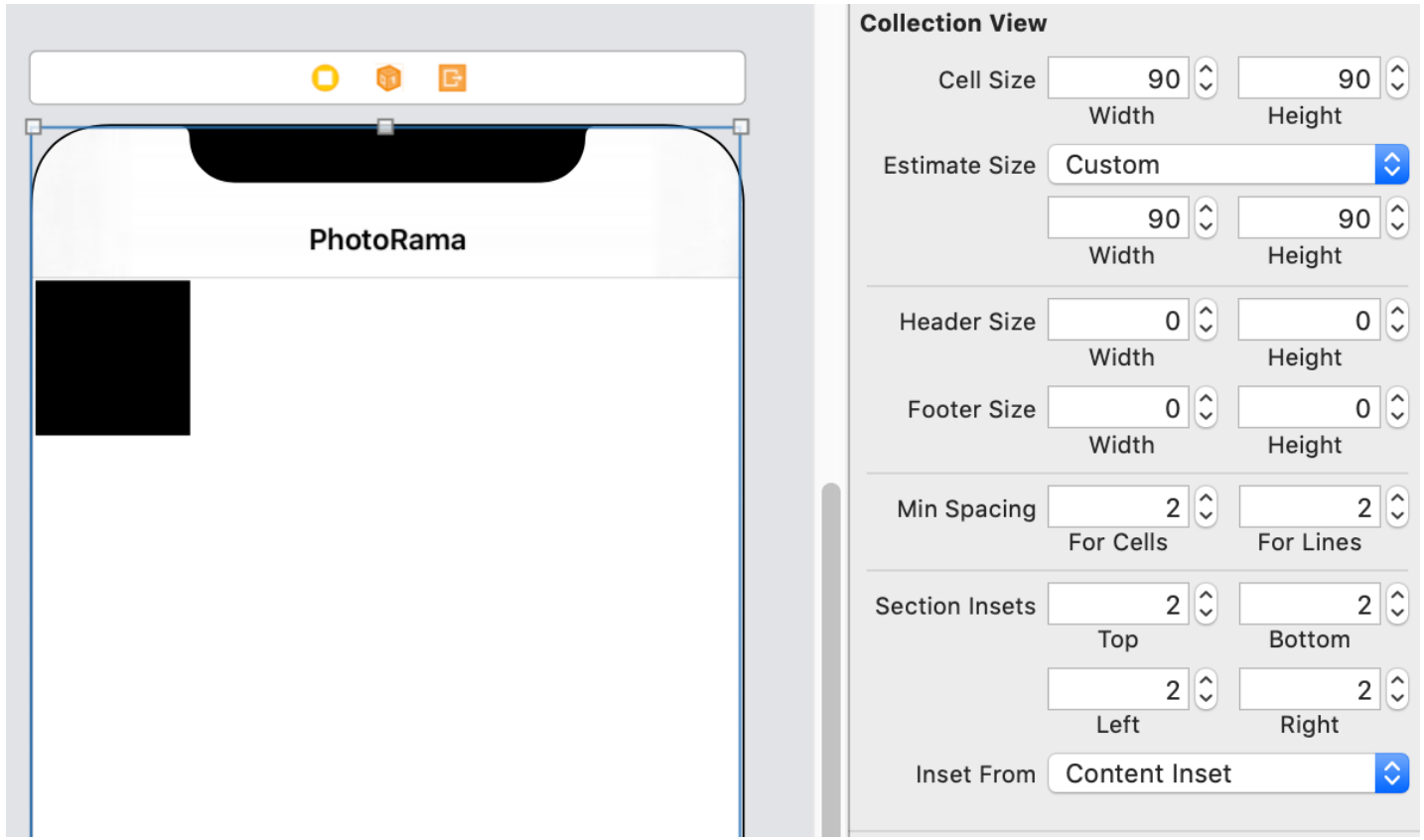
- scrollDirection – scroll horizontally or vertically?
- minimumLineSpacing – the spacing between lines
- itemSize
- + some others

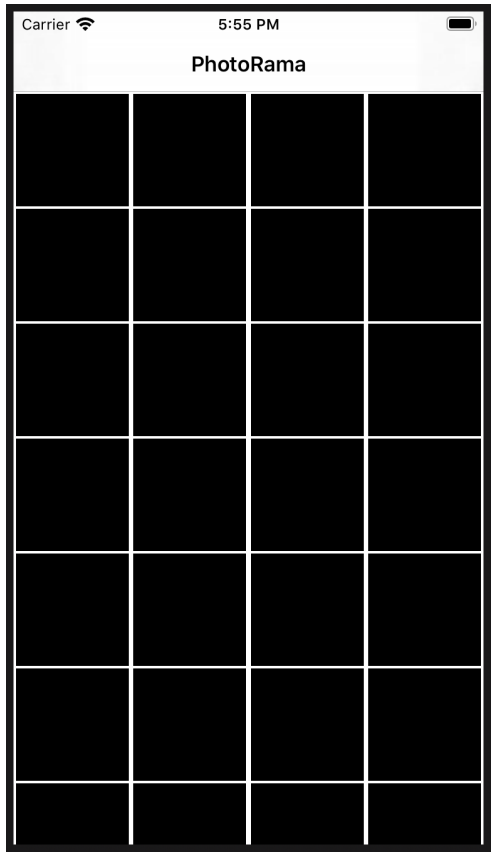


UICollectionViewFlowLayout Properties



Collection View Size Inspector



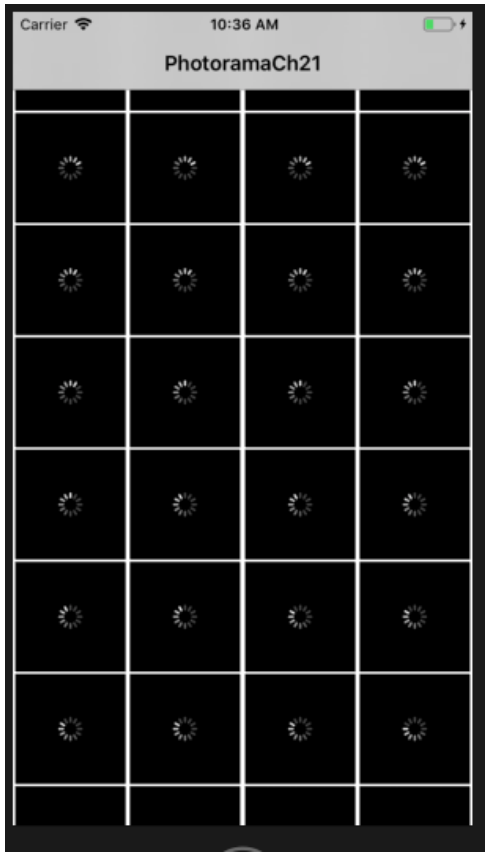


A Custom `UICollectionViewCell`

Eventually, the collection view will display photos that the app downloads

Usability feature: while the downloading is happening, display a spinning indicator

- This spinning indicator will be in a custom `UICollectionViewCell`



Custom `UICollectionViewCell`

A custom `UICollectionViewCell` needs a class to define it

```
import UIKit

class PhotoCollectionViewCell: UICollectionViewCell {
    @IBOutlet var imageView: UIImageView!
    @IBOutlet var spinner: UIActivityIndicatorView!

    func update(with image: UIImage?) {
        if let imageToDisplay = image {
            spinner.stopAnimating()
            imageView.image = imageToDisplay
        } else {
            spinner.startAnimating()
            imageView.image = nil
        }
    }
}
```

The Spinning Indicator

"Turn on" the spinning when the cell is first created

- and when the cell is getting reused

`awakeFromNib()` gives us a hook for the first situation

- `awakeFromNib()` gets called on an object when it is loaded from an archive—in this case, it's from the storyboard file

`prepareForReuse()` gives us a hook for the second situation

- it's called when a cell is about to be reused

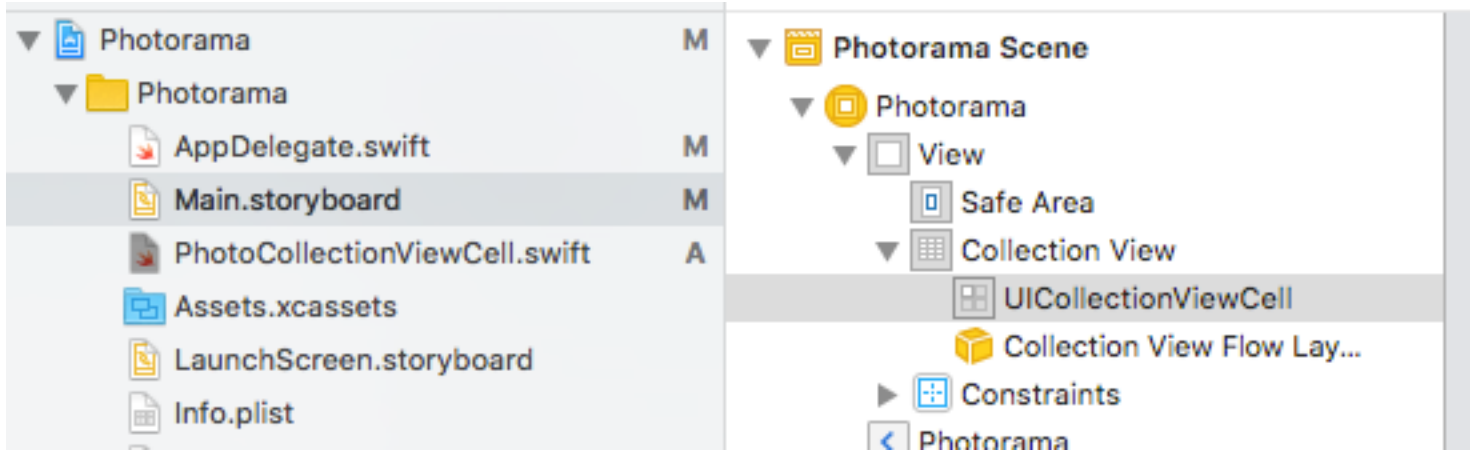
Override these in the new `PhotoCollectionViewCell` class and set the image to nil, which turns on the spinner

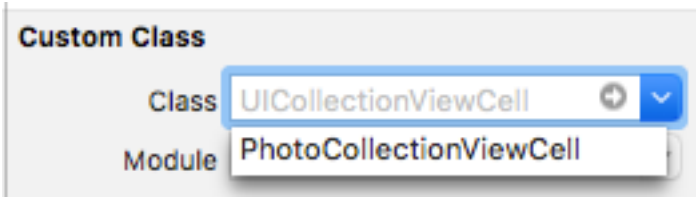
Prototype View Cell

The collection view needs at least prototype view cell

Each different kind of prototype view cell will have a reuse identifier

- by convention, the reuse identifier is just the name of the class that serves as the prototype cell
- this is the same mechanism that is used with `UITableView`

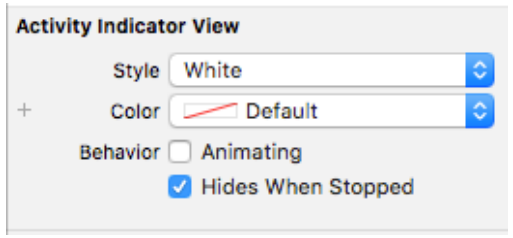


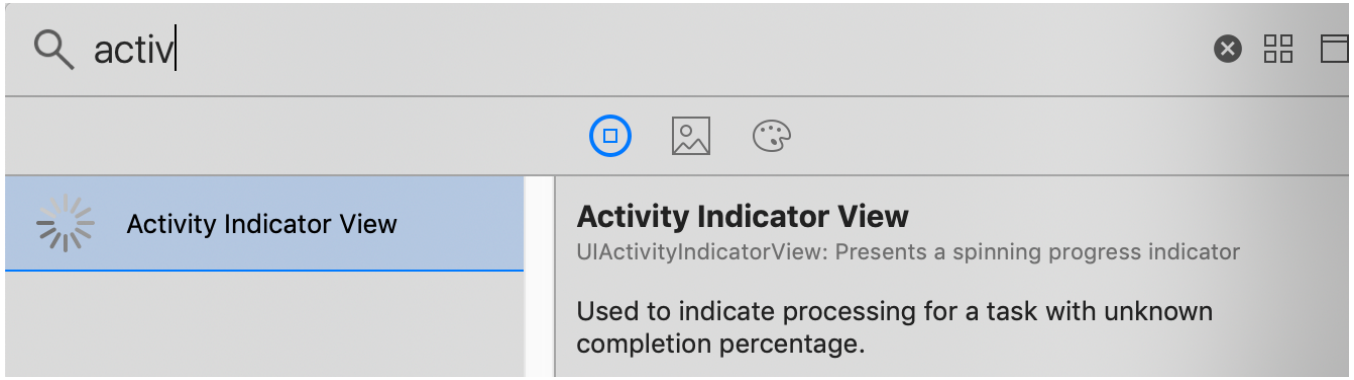


Activity Indicator View

Since each cell in the collection view will display an image, we add an image view to the `UICollectionViewCell`

- then we can drag an activity indicator on to that image view
- and set this attribute:



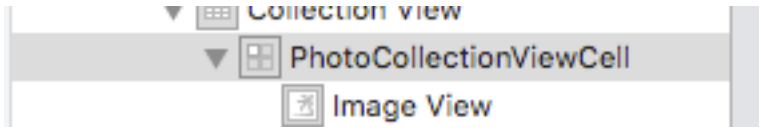


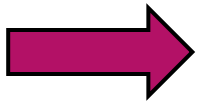
Setting the View Cell Identifier

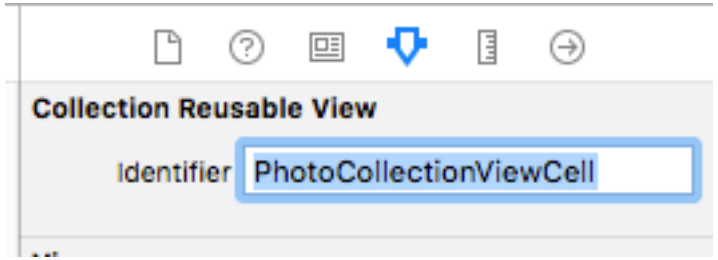
```
class PhotoDataSource: NSObject, UICollectionViewDataSource {

    var photos = [Photo]()

    func collectionView(_ collectionView: UICollectionView,
                        numberOfItemsInSection section: Int) -> Int {
        return photos.count
    }

    func collectionView(_ collectionView: UICollectionView,
                        cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        //let identifier = "UICollectionViewCell"
        let identifier = "PhotoCollectionViewCell"
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: identifier,
                                                      for: indexPath)
        return cell
    }
}
```

Another Required Step

The book doesn't show this, but you must also set the Collection Reusable View Identifier



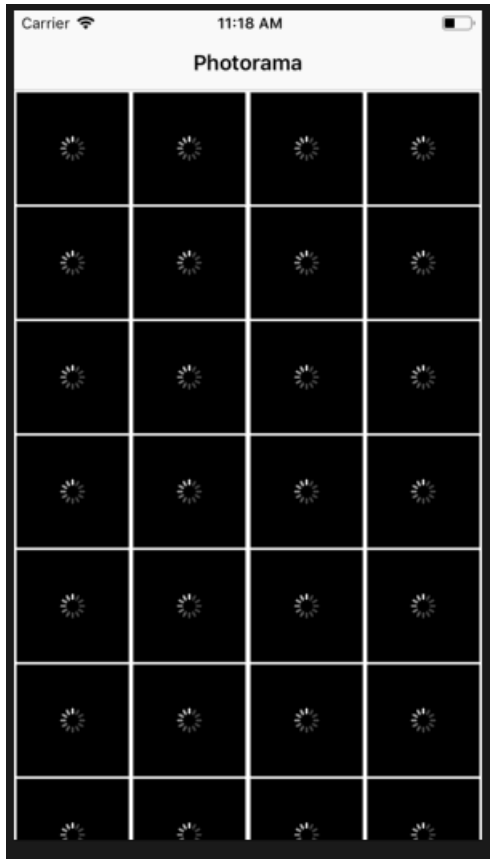




Activity Indicators

You should see this:

- each cell in the collection view has the spinner running



Downloading the Image Data

The initial query returns a list of links

- the query returns metadata about the photos

Each link represents an actual image from our query

Downloading the Image Data

Question: when to download the images?

Poor choice: after the query is returned

- image files are large
- we might end up downloading images that are never actually viewed in the collection view

Better choice: when a particular view in the collection view will appear on screen

On-Demand Loading

Load the image only when its view cell will appear on screen

- `UICollectionView` has a mechanism to support this through its `UICollectionViewDelegate`
- `UICollectionViewDelegate` provides `collectionView(_:willDisplay:forItemAt:)`, which will be called every time a cell is getting displayed onscreen

On-Demand Loading

Have `PhotosViewController` conform to `UICollectionViewDelegate`

- and set the delegate to be the controller itself

```
class PhotosViewController: UIViewController, UICollectionViewDelegate {
    ...
    override func viewDidLoad() {
        super.viewDidLoad()
        collectionView.dataSource = photoDataSource
        collectionView.delegate = self
```

On-Demand Loading

But as is, the application will not cache images

- if a view moves off the screen, the app will reload its image when it moves back on screen

Solution: reuse `ImageStore`, from Chapter 15, in the Homepwner project

- save an image in secondary storage when it loads
- will require us to ask whether an image is in in the `ImageStore`
- and for that, we need to be able to compare two Photo instances: “does photo1 == photo2?”

Checking to See Whether Two Photos Are Equal

To determine whether two Photo instances are equal: make the Photo class conform to Equatable

This way, we can call `index(of:)` on the Photos array with Photo

```
class Photo: Equatable {
   // other stuff we've already defined
    static func == (lhs: Photo, rhs: Photo) -> Bool {
        // two photos are the same if the have the same photoID
        return lhs.photoID == rhs.photoID
    }

}
```

autocorrect wants to turn this into “Equitable”

my favorite autocorrect is still “Hashable” => “Washable”

Extensions

Extensions in Swift allow us to add additional behavior to an existing type

- the type can be a built-in type, such as Int, or a user-defined type

Extensions allow us to group functionality into logical units

- they also allow us to add new function to a type
- even if we don't have the source code of the definition of the type

```
let fourteen = 7.doubled
// enable this function by extending Int

extension Int {
    var doubled: Int {
        return self * 2
    }
}
```

Extensions

Adding a new == as a class method would be awkward

- since we would have to change the original class definition

A more elegant solution:

```
class Photo {
  // the original class is unchanged
}

extension Photo: Equatable {
    static func == (lhs: Photo, rhs: Photo) -> Bool {
        // two photos are the same if the have the same photoID
        return lhs.photoID == rhs.photoID
    }
}
```

Navigating to a Photo

Basic UI behavior that we want for the Collection VIew:

- when the user taps on a photo in the collection view, that photo should be shown in a UIImage in its own view controller

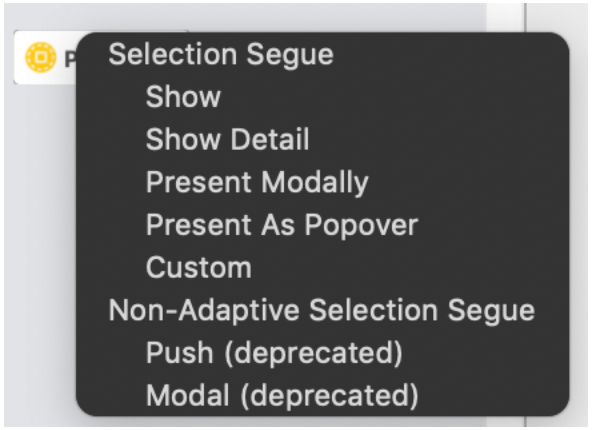
New view controller:

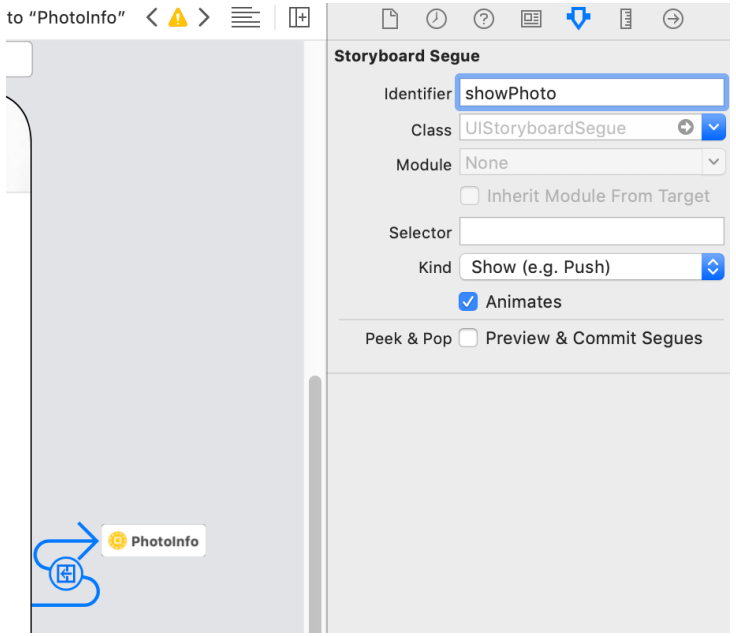
```
class PhotoInfoViewController: UIViewController {
    @IBOutlet var imageView: UIImageView!
}
```

I did this on a second storyboard, to show an example of using a storyboard reference

Create the Segue

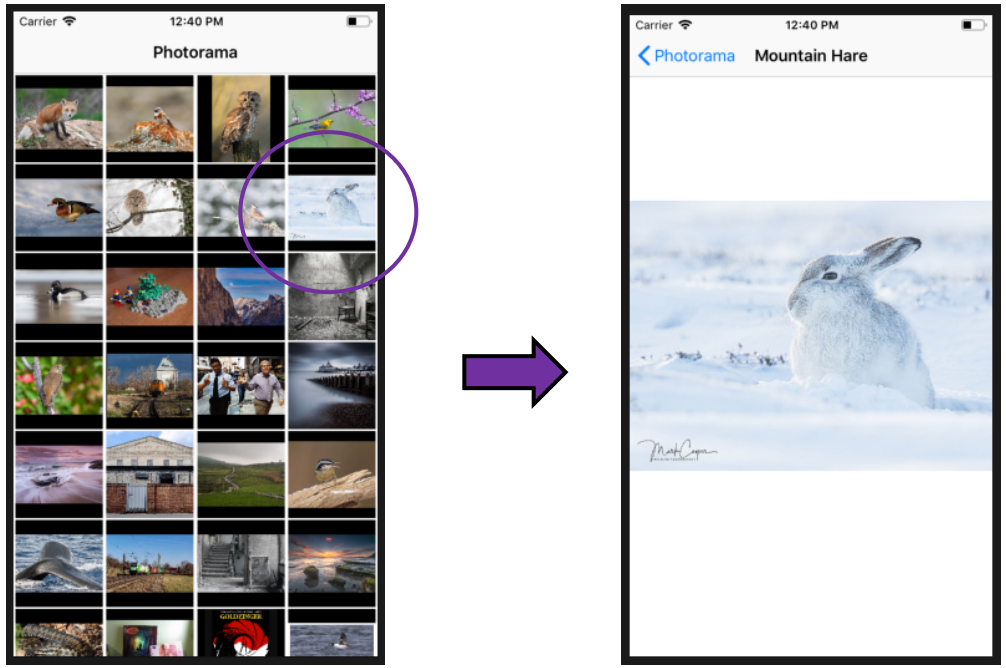
- Control-drag from the PhotoCollectionViewCell to the new controller
- Select "Show"
- Give the segue an identifier: showPhoto





Overview-Detail View

- When the user taps a photo, the segue will be triggered
- The sending view controller needs to pass the Photo and the PhotoStore to the destination view controller



Segue Action

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    switch segue.identifier {
    case "showPhoto"?:
        if let selectedIndexPath = collectionView.indexPathsForSelectedItems?.first {
            let photo = photoDataSource.photos[selectedIndexPath.row]
            let destinationVC = segue.destination as! PhotoInfoViewController
            destinationVC.photo = photo
            destinationVC.store = store
        }
    default:
        preconditionFailure("Unexpected segue identifier.")
    }
}
```