iOS Dev Accelerator Week 2 Day 3

- Homework Review
- Animating Constraints
- Magic Numbers
- CollectionViews
- Functional Programming
- DRY
- Property Observers

Homework Review

Animating Constraints

Constraints and animation

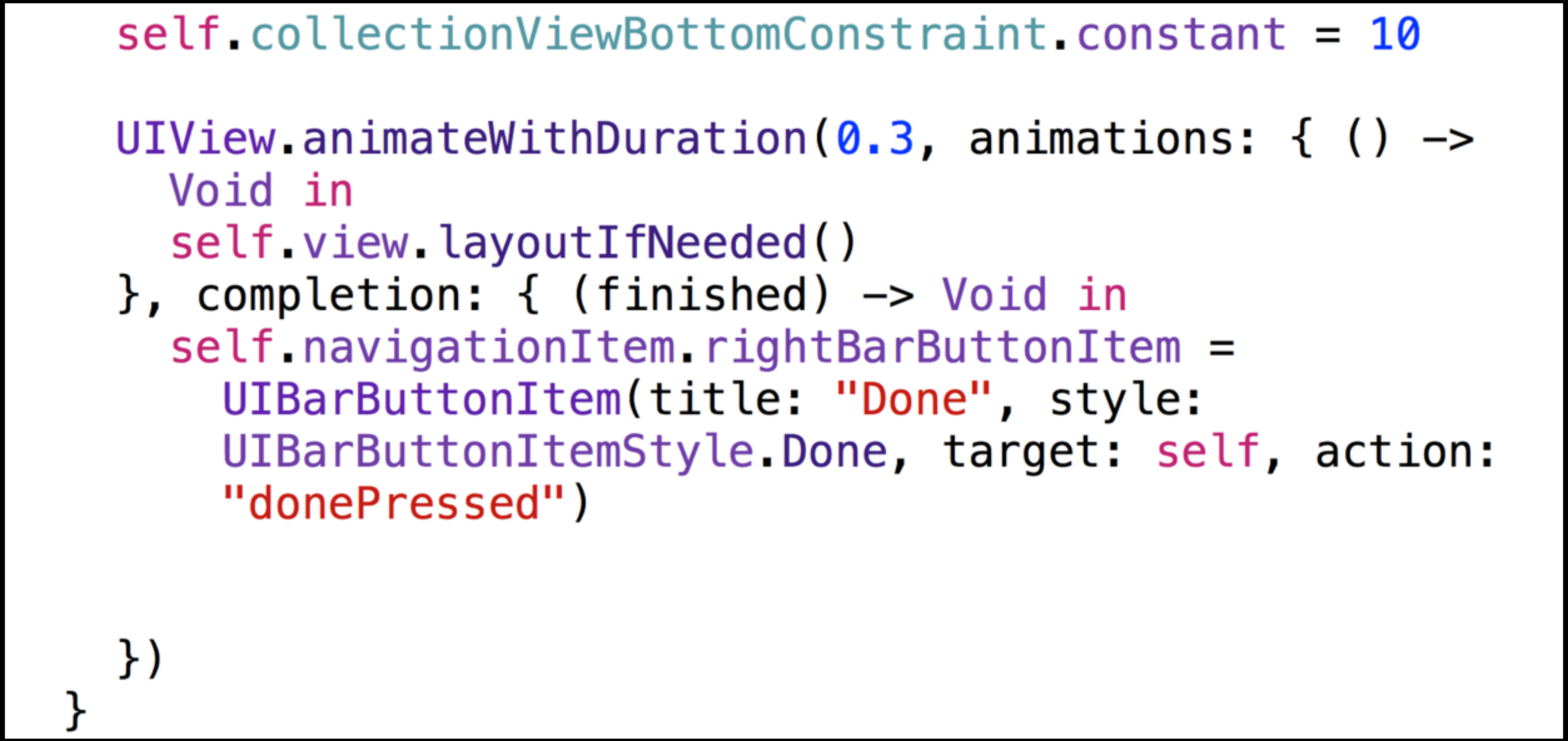
- Constraints can be animated, meaning you can smoothly move or change the size of views that are managed by autolayout
- It can be tough at first to figure out. “Well how do I get to the constraints of each view?”
- In code its a bit of a chore, but luckily, we can create IBOutlets to individual constraints from the storyboard. Hooray!

Demo

Animating Autolayout changes

1. Change the constant of the constraint

2. In your animation block, call layoutIfNeeded() on the view controller’s view

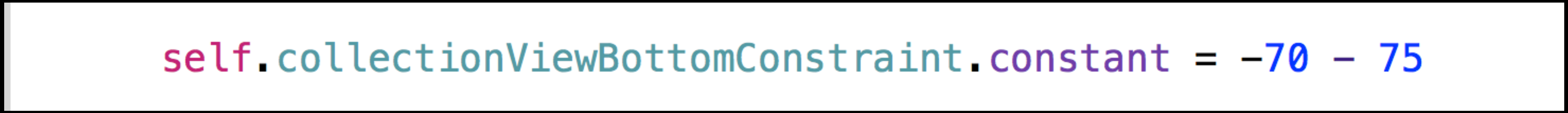


```
    self.collectionViewBottomConstraint.constant = 10

    UIView.animateWithDuration(0.3, animations: { () ->
      Void in
      self.view.layoutIfNeeded()
    }, completion: { (finished) -> Void in
      self.navigationItem.rightBarButtonItem =
        UIBarButtonItem(title: "Done", style:
        UIBarButtonItemStyle.Done, target: self, action:
        "donePressed")


    })
  }
```

Demo

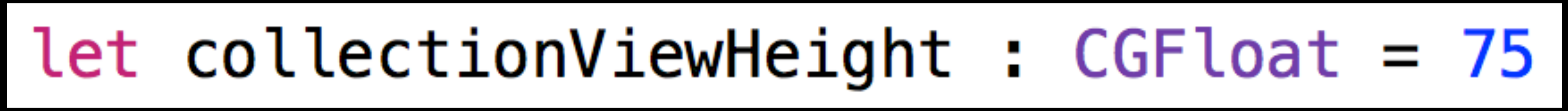
Magic Numbers

- A Magic Number is a direct usage of numbers in code:



```
self.collectionViewBottomConstraint.constant = -70 - 75
```

- This should always be refactored to use variables instead:



```
let collectionViewHeight : CGFloat = 75
```

```
self.collectionViewBottomConstraint.constant = -self.tabBarController!.tabBar.frame.height
  - self.collectionViewHeight
```

- **0 and 1 are excused from the magic number rule**

Magic Numbers

- Benefits of Eliminating Magic Numbers:
    - Greatly improves readability: If you run across a magic number in someone else’s code, or even your own, it is very hard to figure out why that number was chosen, and what is being used for. (eg: why is this number 8? Why did they choose 8? What happens if I change it?)
    - Easier to maintain: often times numbers you use will need to be used in multiple places. Instead of having to change each magic number, just change the variable that stores the number (eg: collection view height property, only change it once when it is declared)

Demo

UICollectionView

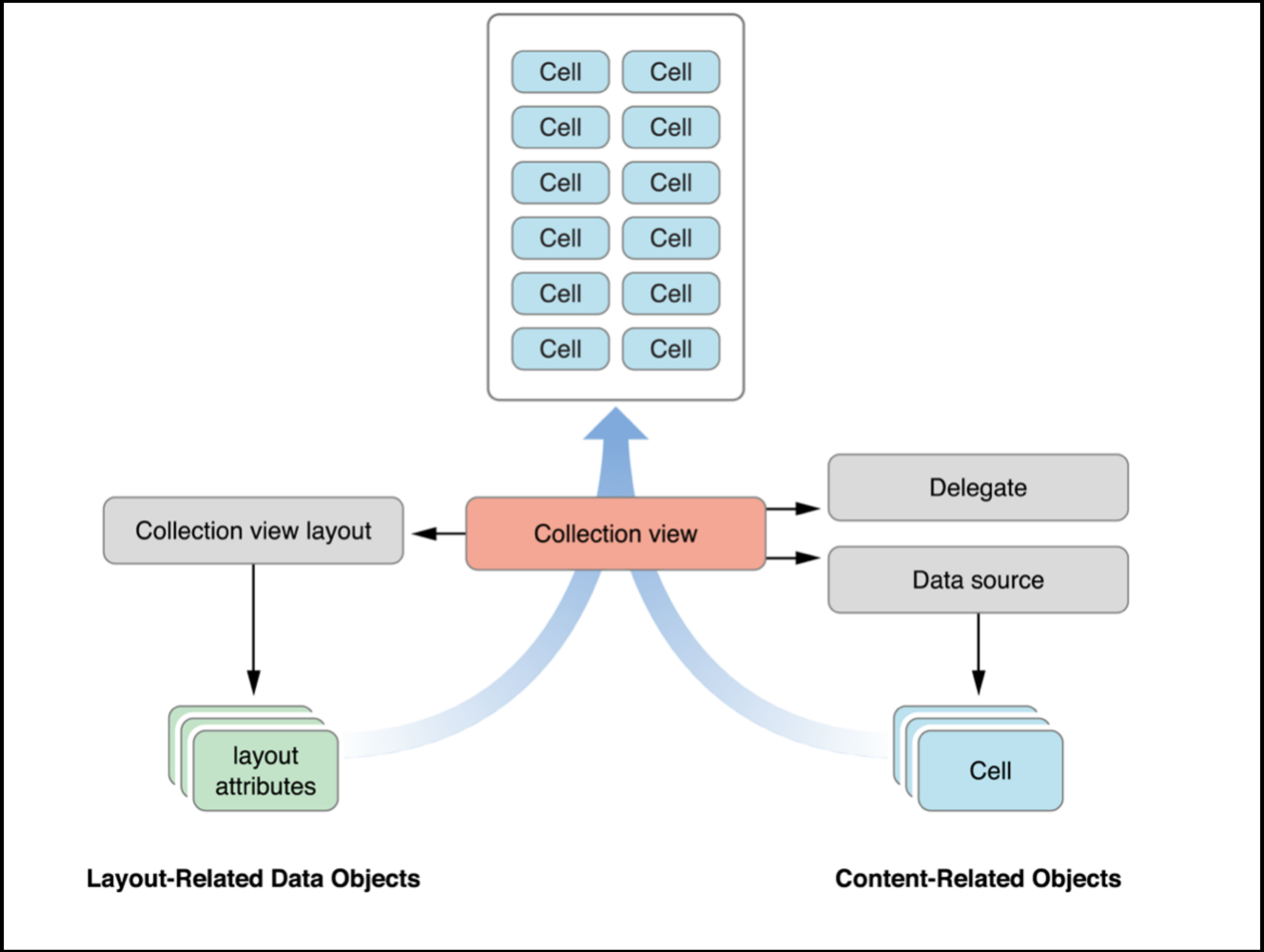
`UICollectionView`

- “A collection view is a way to present an ordered set of data items using flexible and changeable layout”
- Most commonly used to present items in a **grid-like** arrangement. (Items to collection views are as rows to table views)
- Creating custom layouts allow the possibility of many different layouts (grids, circular layouts, stacks, dynamic,etc)

Internal WorkFlow

- You provide the data (datasource pattern!)
- The layout object provides the placement information
- The collection view merges the two pieces together to achieve the final appearance.

Collection View Objects



Reusable Views

- Collection views employ the same recycle program that table views do. (Same Queue data structure)
- 3 reusable views involved with collection views:
    1. Cells : Presents the content of a single item
    2. Supplementary views : headers and footers
    3. Decoration views : wholly owned by the layout object and not tied to any data from your data source. Ex : Custom background appearance.
- Unlike table views, collection view imposes no styles on your reusable views, they are for the most part blank canvases for you to work with.

`CollectionViewDataSource`

- Very similar to tableview’s datasource. It is required.
- Must answer these questions:
    - For a given section, how many items does a section contain?
    - For a given section or item, what views should be used to display the corresponding content? (just like cellForRow)

Demo

Functional Programming

First class

- Functions in Swift are considered first-class citizens
- This just means functions can be treated like just like regular variables
- They can be passed as arguments to other functions, and functions can return other functions
- The key fact of functional programming is that functions are values, no different from structs, integers, booleans.

Using functional programming in our app

- We need to lazily generate a bunch of filtered thumbnails in our collection view
- We could create a thumbnail wrapper object that contains a reference the original and an optional property for the filtered thumbnail, but then we still need a way to figure out which thumbnail object will show which filter.
- Instead, lets create an array of functions that each perform a different filter
- We will use that array to back our collection view
- Heres how an array of filter functions would look like as a property:

```
let filters : [(UIImage, CIContext)->(UIImage)]!
```

Demo

DRY – Don’t Repeat Yourself

- DRY is another principle of software development, which aims at reducing repetitive code/tasks.
- It has a counterpart, called the rule of 3, which provides a nice balance.
- The rule of 3 is a refactoring rule of thumb that states a piece of code can be copied once, but once that code is used 3 times, its time to extract a new procedure instead of having all these duplicates.

Why Duplication sucks

- Duplication in programming is bad because it makes code harder to maintain. If you make a change in one place in duplicated code,you need to change it in all the other places
- Also duplication is just doing something over that you already have done, which isn't efficient!
- Keep in mind, retroactively applying DRY and the rule of 3 also takes time, so you should to keep these things in mind as you write code, not after you write the code.

Demo

Property Observers

Property Observers

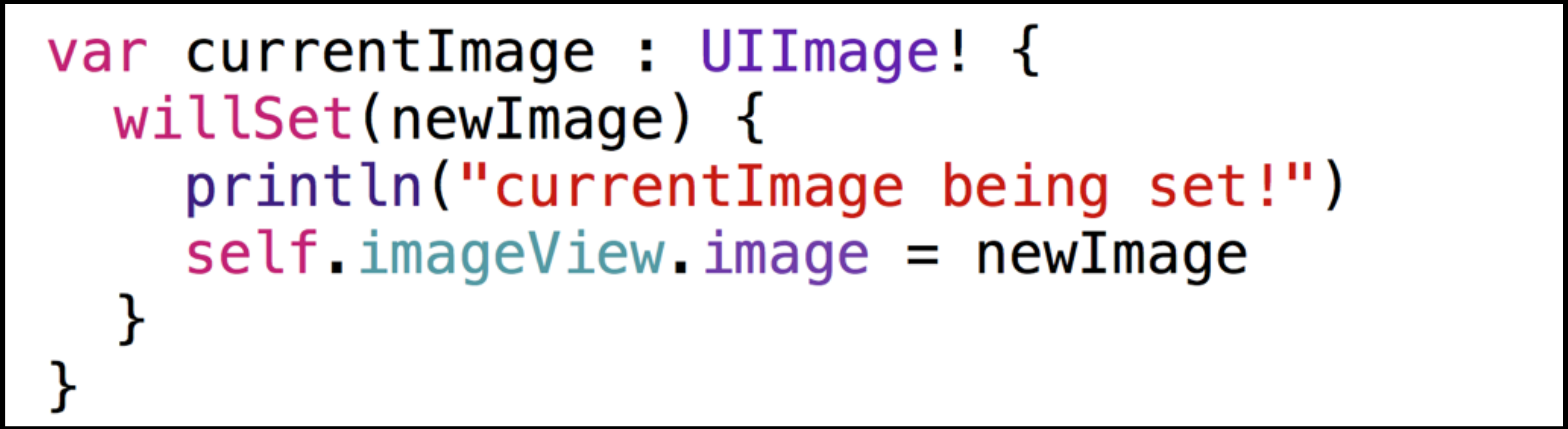
- Property Observers observe and respond to changes in a property value
- Property Observers are called every time a property value is set, even if the value is the same as the property’s current value
- You can even add Property Observers to inherited properties!

2 different observer options

- willSet: is called just before the value is stored in the property
- didSet: is called immediately after the new value is stored

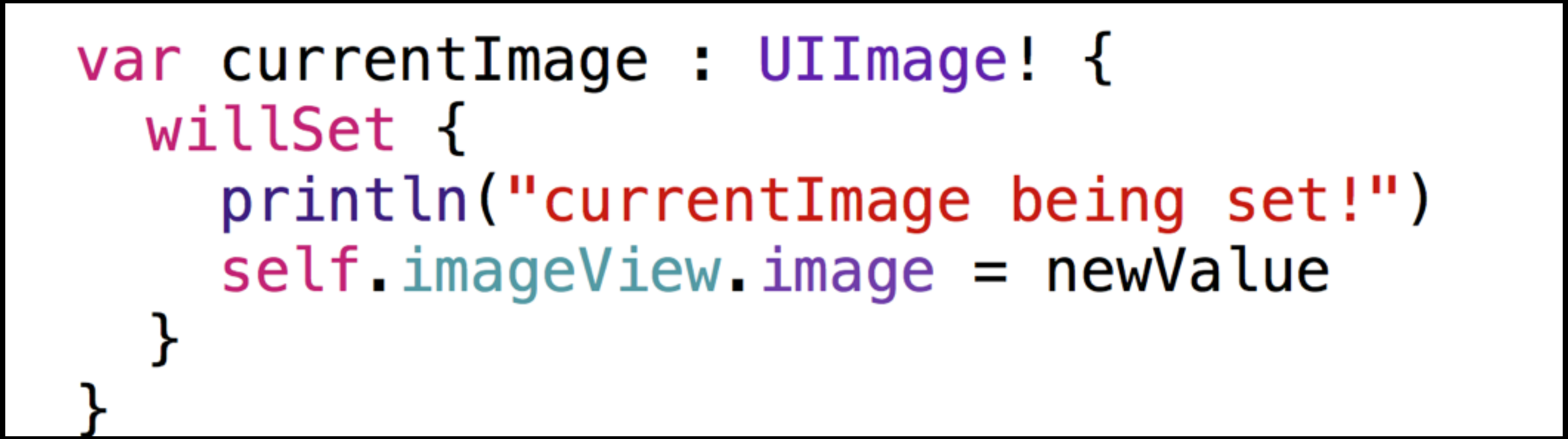
willSet

- If you implement the willSet observer, it is passed the new property value as a constant parameter



```swift
var currentImage : UIImage! {
  willSet(newImage) {
    println("currentImage being set!")
    self.imageView.image = newImage
  }
}
```

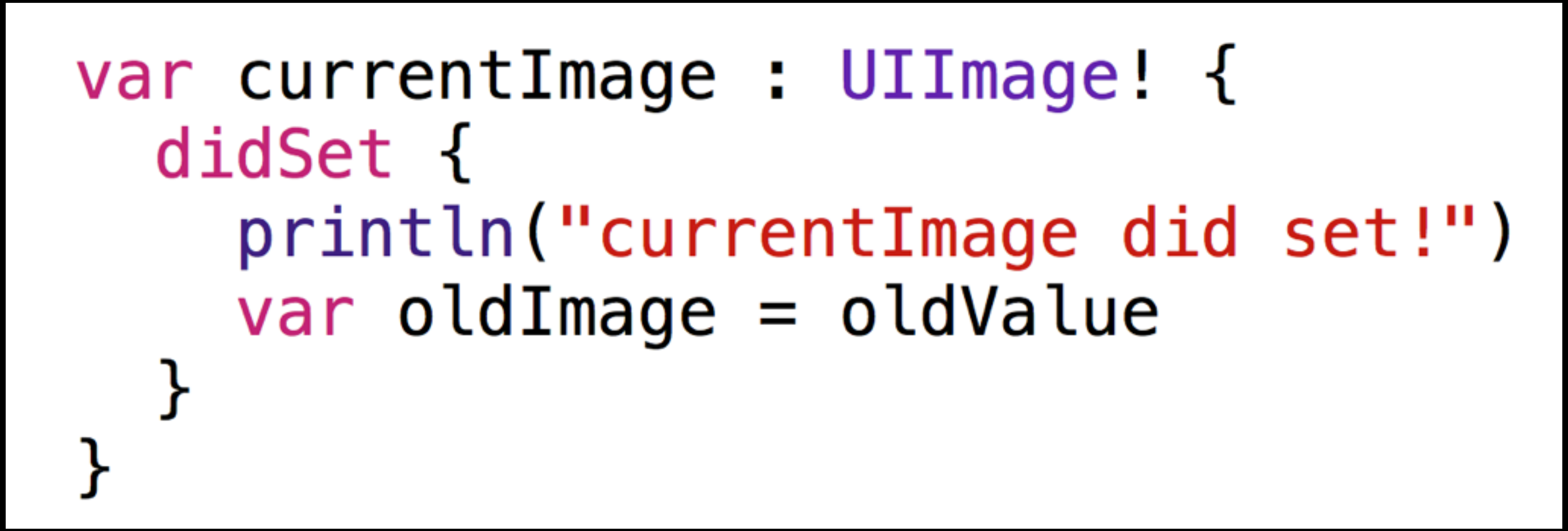
- You can specify your own name for this parameter as part of your willSet implementation. If you choose not to, a default name of newValue is provided



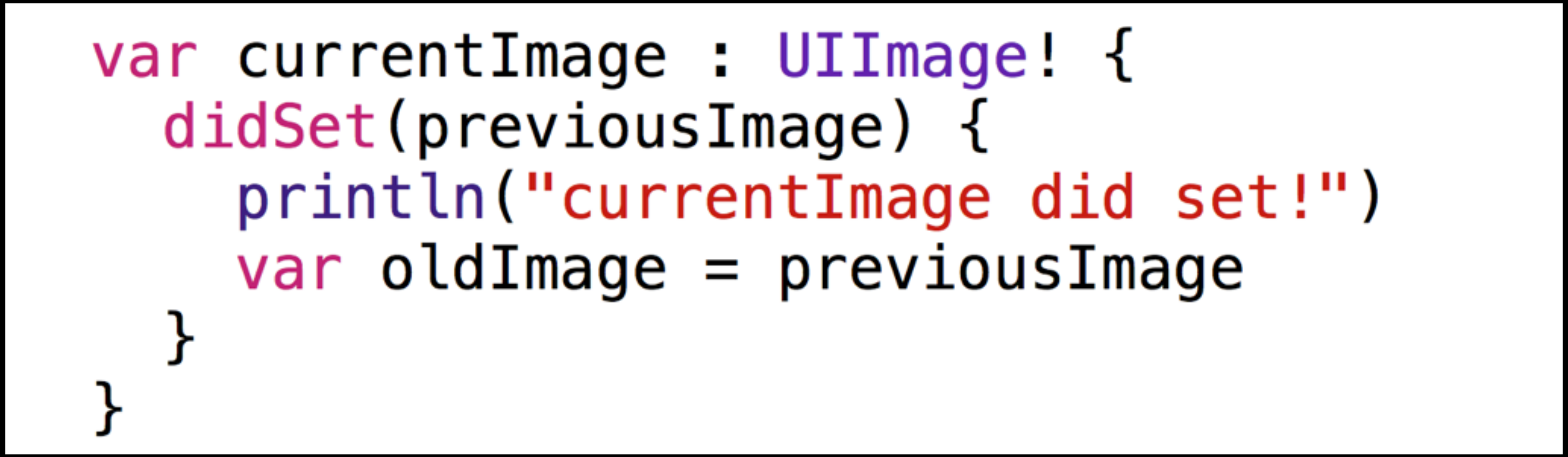
```swift
var currentImage : UIImage! {
  willSet {
    println("currentImage being set!")
    self.imageView.image = newValue
  }
}
```

didSet

- Same with the didSet observer:



```swift
var currentImage : UIImage! {
  didSet {
    println("currentImage did set!")
    var oldImage = oldValue
  }
}
```



```swift
var currentImage : UIImage! {
  didSet(previousImage) {
    println("currentImage did set!")
    var oldImage = previousImage
  }
}
```

Important note

- Property Observers are not called during initialization of the class they belong to, or when the property is given a default value

Demo