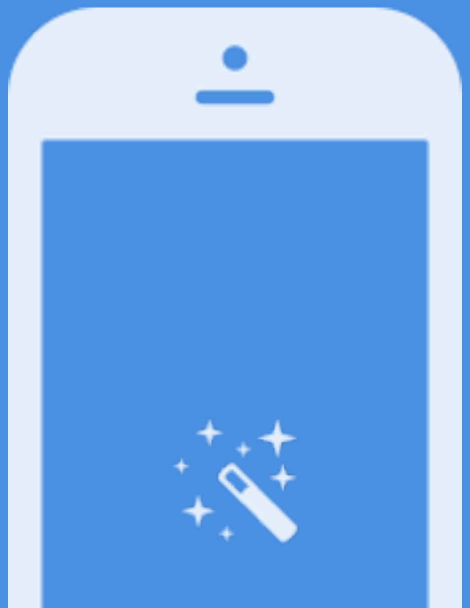
# iOS Crash Course

Session Six

Janum Trivedi

iOSCrashCour.se



# Session 5 Overview

- Earlier this week we kept working on our to-do list application
    - Added more properties to Item (NSDate, bool)
    - Created a custom UITableViewCell subclass
    - Learned UIGestureRecognizer

# Session 6 Overview

- We’re going to take a quick break from working on the project today
- New concepts:
  - Navigation in iOS (how we can switch between view controllers)
  - Objective-C **Blocks** (aka, closures, lambdas)
  - Concurrency with Grand Central Dispatch

# Followup and Questions

- Can everyone take a minute to respond to the office hours poll?
- Office hours will take place at Shift Creator Space (631 Oxford Rd)
- Questions from last session or about today’s?

# Navigation

- All our apps so far (including our to-do app) only contain one "screen"
- In this "screen" or **view controller**, we may have many subviews (**UIViews**), like a UITableView, UILabel, etc.
- Still, those views are contained in **one** view controller

# Navigation

- We haven’t seen how to have multiple screens or pages in our applications
- For very simple apps, a single screen might suffice
- Other apps may be too complex or contain more than what we could cram into one screen

# What is a “screen”?

- In iOS, a “screen” is a view controller (of type UIViewController)

- In our storyboard, we had exactly **one** view controller (of class ViewController)

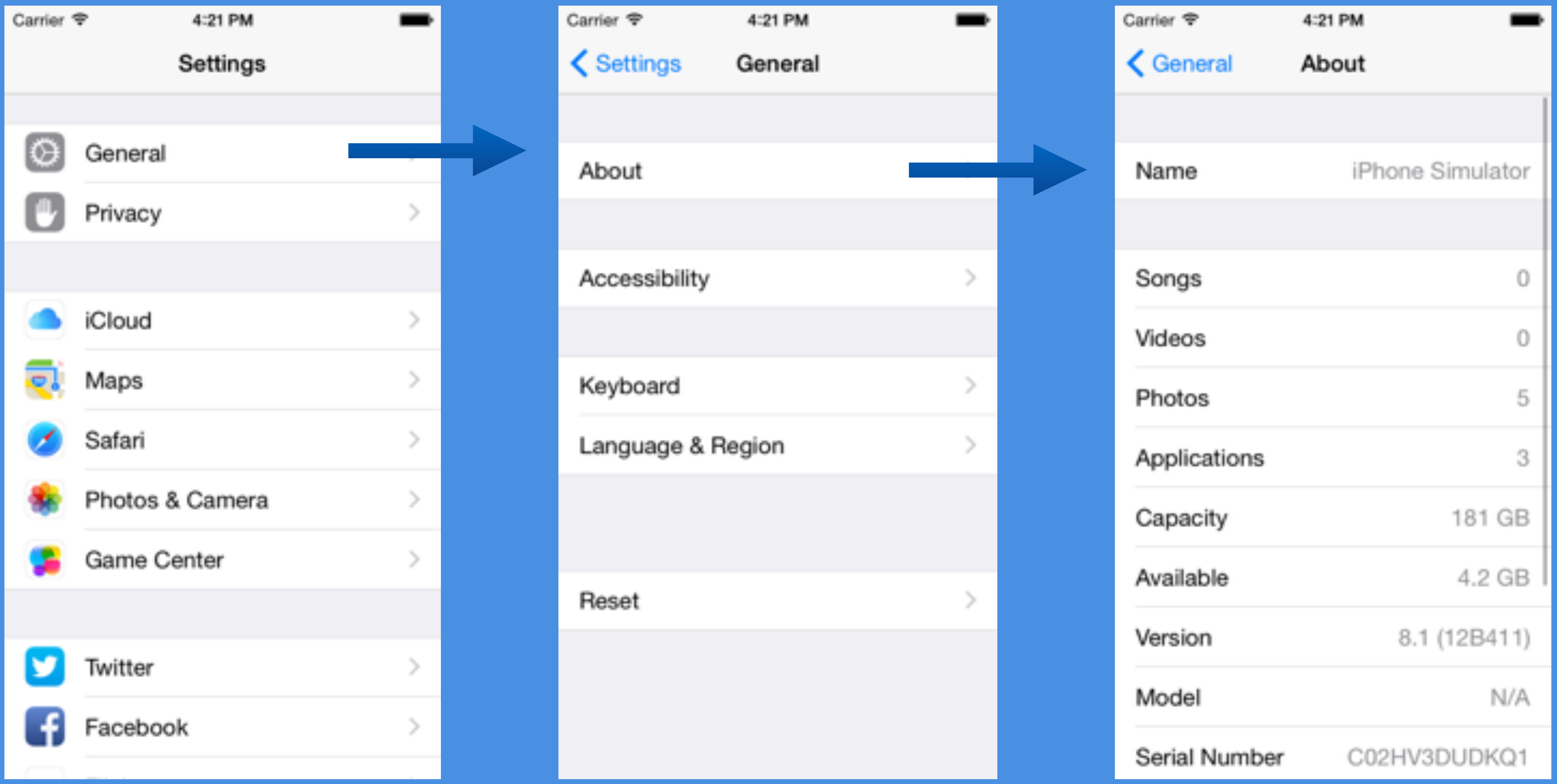
- This makes sense- **each screen must have its own UIViewController subclass**

# What is a “screen”?

- Every time you use an app that takes you to a new screen, it is loading a new UIViewController
- We’re going to focus on navigating linearly through multiple view controllers

# Navigation

- Example: Settings.app



Carrier 4:21 PM

Settings

General

Privacy

iCloud

Maps

Safari

Photos & Camera

Game Center

Twitter

Facebook

Carrier 4:21 PM

‹ Settings General

About

Accessibility

Keyboard

Language & Region

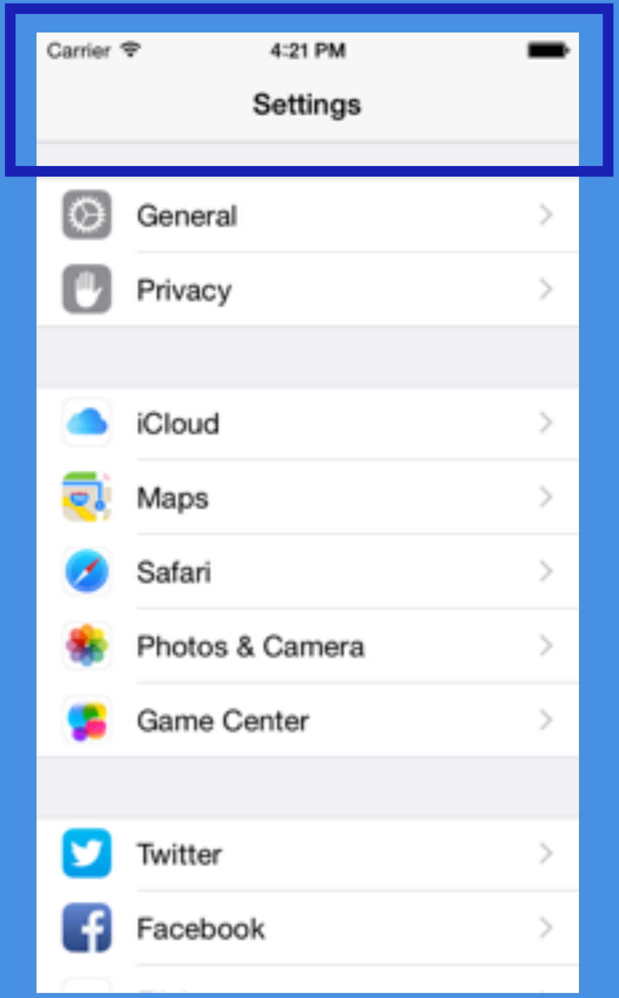
Reset

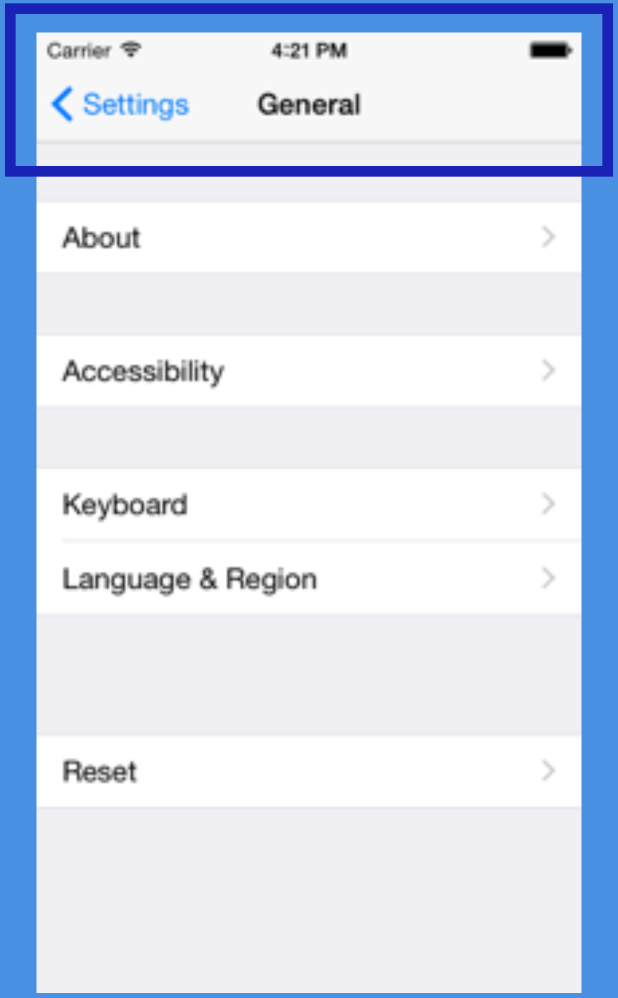
Carrier 4:21 PM

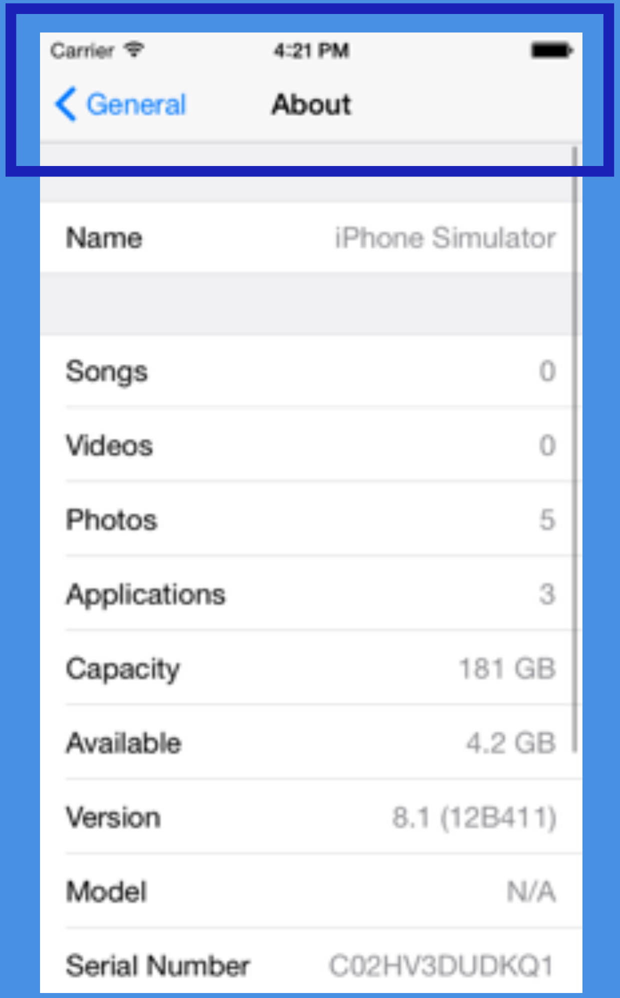
‹ General About

| | |
|---|---|
| Name | iPhone Simulator |
| Songs | 0 |
| Videos | 0 |
| Photos | 5 |
| Applications | 3 |
| Capacity | 181 GB |
| Available | 4.2 GB |
| Version | 8.1 (12B411) |
| Model | N/A |
| Serial Number | C02HV3DUDKQ1 |

- The navigation bar stays while the content changes
- The UINavigationBar is part of the navigation controller (UINavigationController)

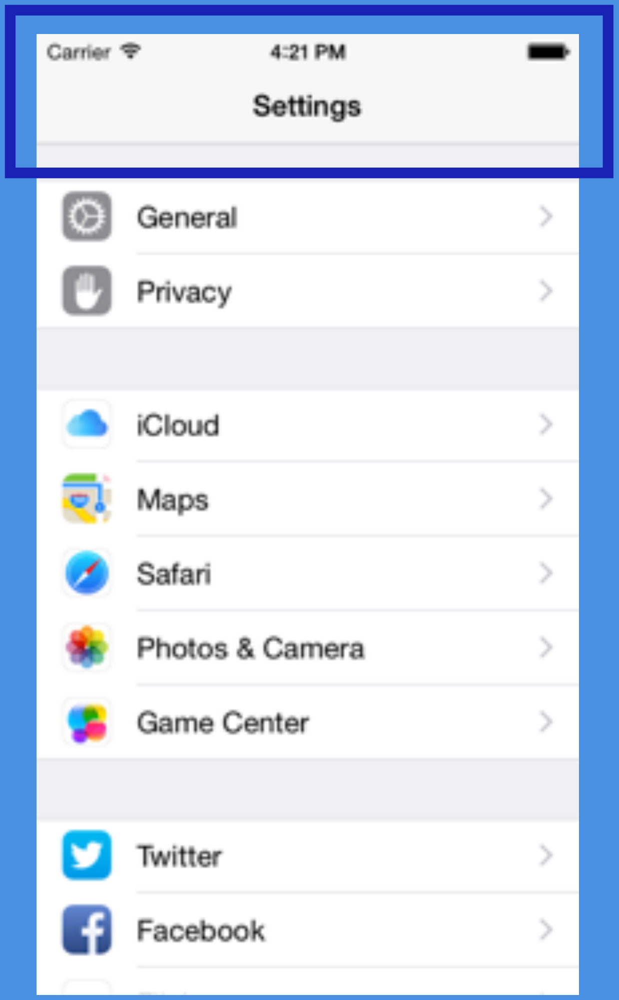


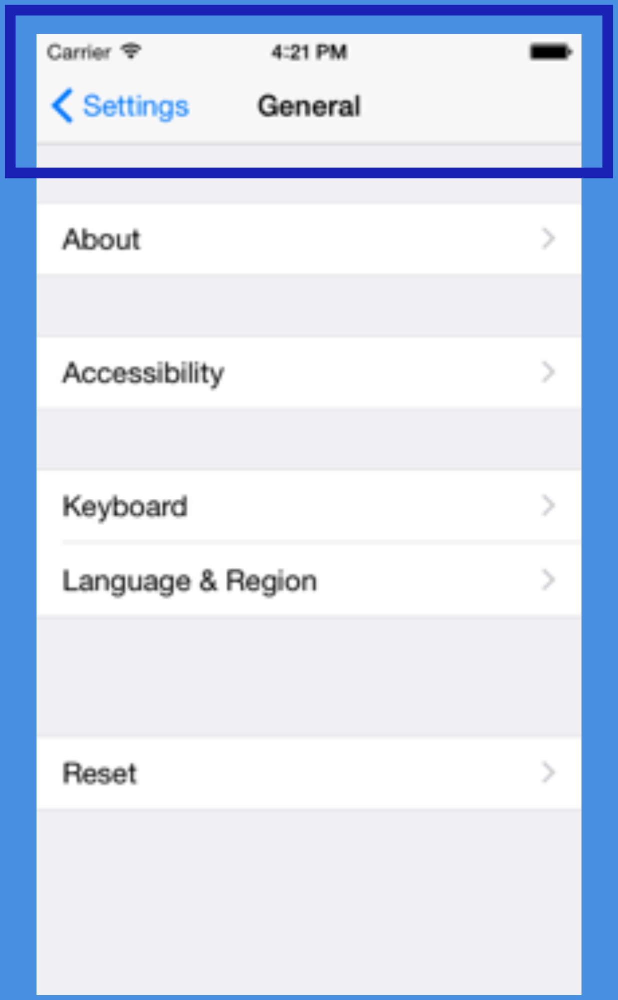


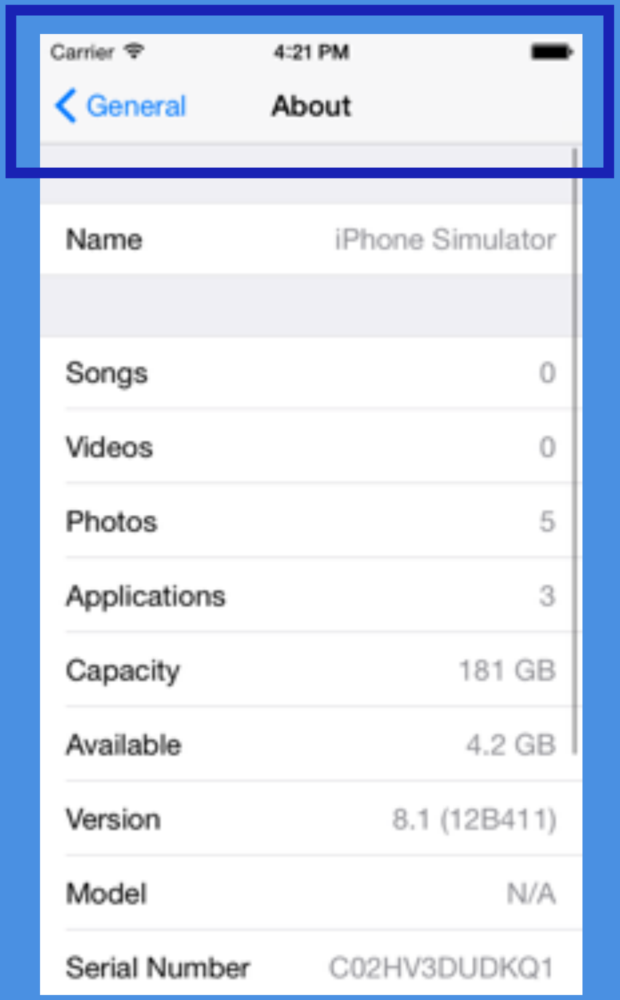


| | |
|---|---|
| Name | iPhone Simulator |
| Songs | 0 |
| Videos | 0 |
| Photos | 5 |
| Applications | 3 |
| Capacity | 181 GB |
| Available | 4.2 GB |
| Version | 8.1 (12B411) |
| Model | N/A |
| Serial Number | C02HV3DUDKQ1 |

- Recall: our "Root View Controller" is a UINavigationController that has our View Controller as a child







| | |
|---|---|
| Name | iPhone Simulator |
| Songs | 0 |
| Videos | 0 |
| Photos | 5 |
| Applications | 3 |
| Capacity | 181 GB |
| Available | 4.2 GB |
| Version | 8.1 (12B411) |
| Model | N/A |
| Serial Number | C02HV3DUDKQ1 |

# UINavigationController

- **UINavigationController** is the class that **manages our view hierarchy**

- It manages transitioning to a new screen (view controller), and when we're done, going back to the previous screen

- The **navigation controller** "holds" all our **view controllers**, and is necessary if we want to navigate between controllers

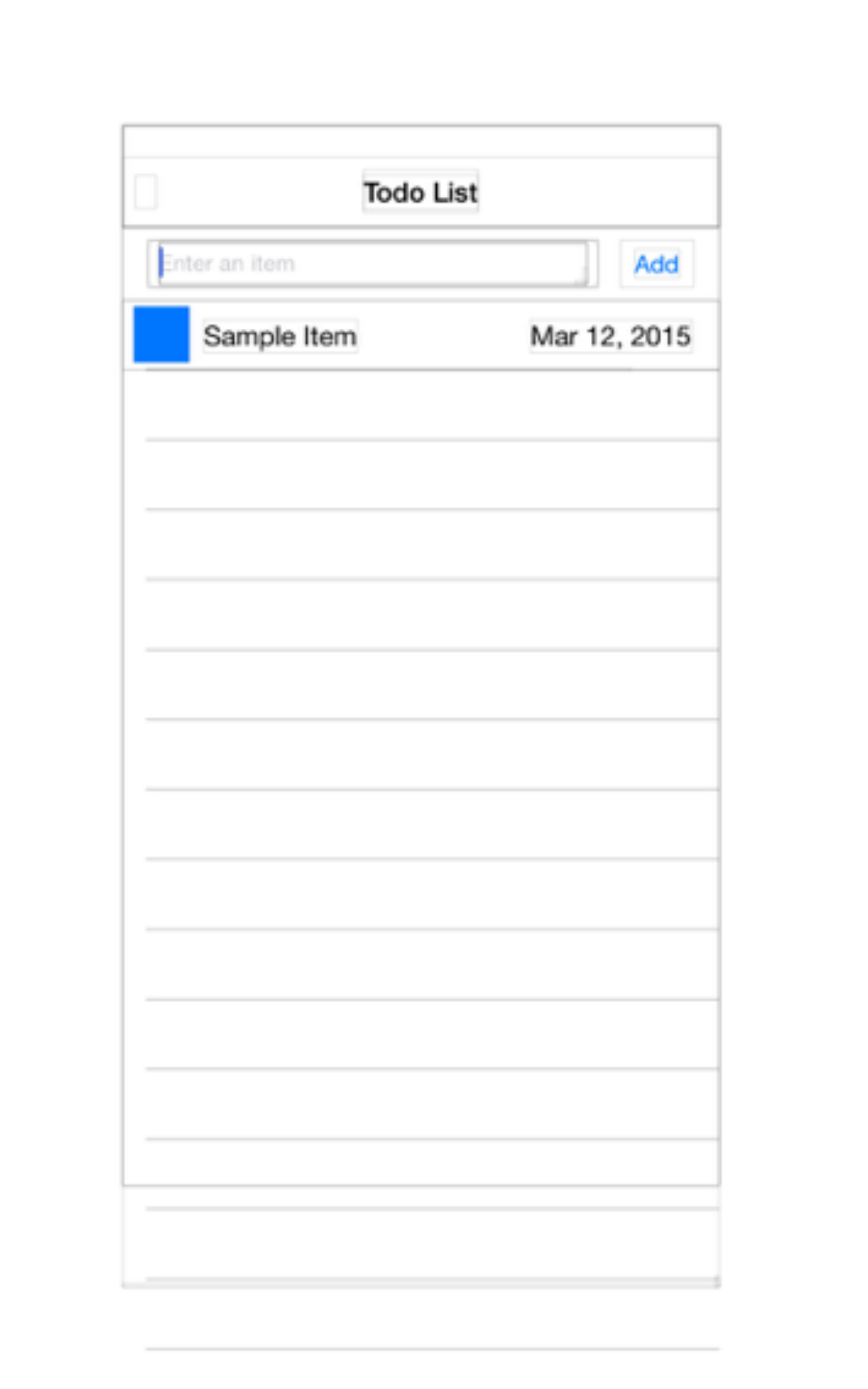
# UINavigationController

- One of the properties of **UINavigationController** is the **UINavigationBar** itself
    - This is the visible bar at the top that shows the title, back button, etc.
    - Even holds **UIBarButtonItems**
        - Special kind of UIButton that [mostly] exist only in navigation bars

# Visualization

- From the front, our app looks like this

- In the background, our app actually layer cake…



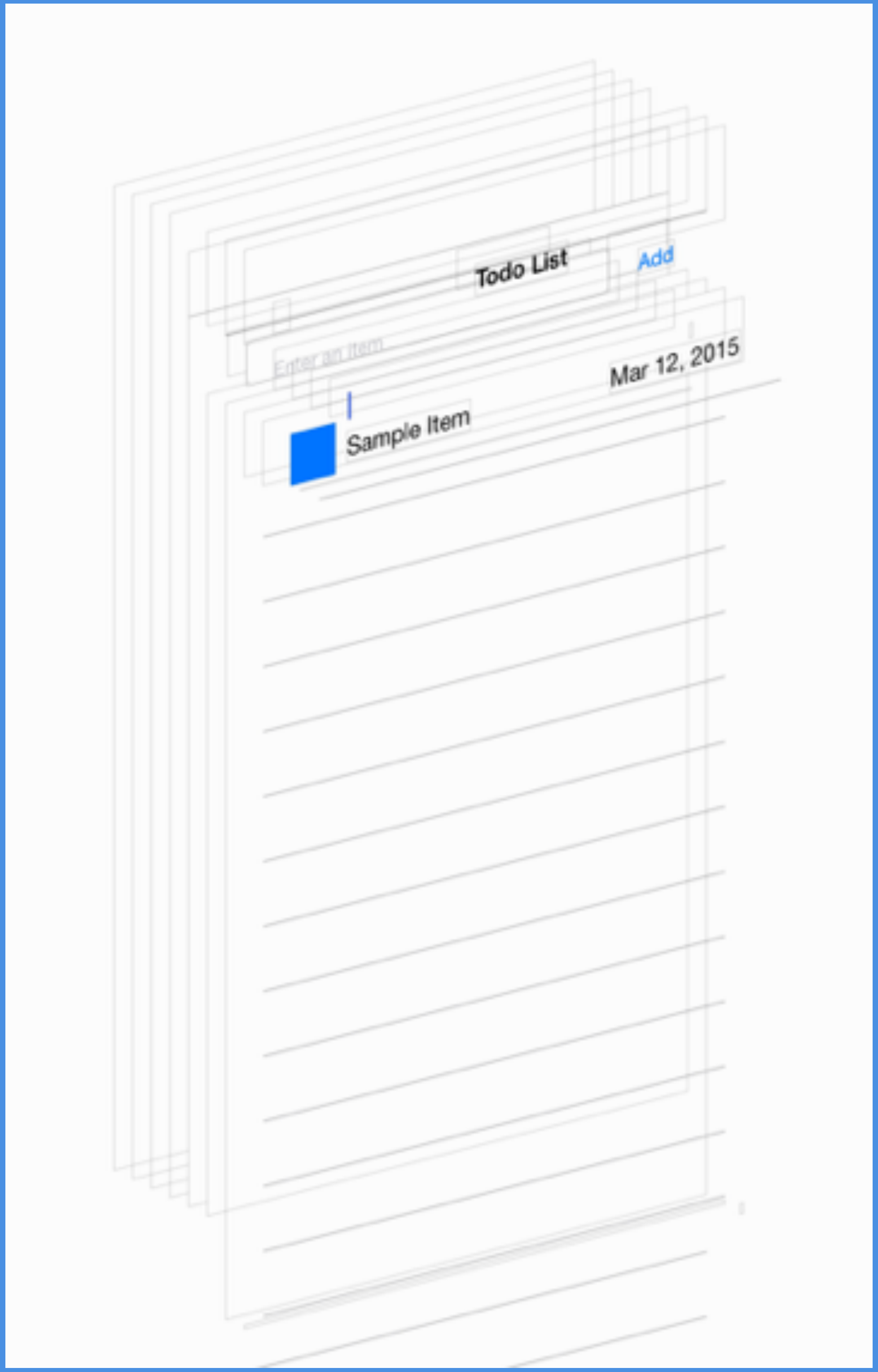
Sample Item Mar 12, 2015

# Visualization

- From the front, our app looks like this

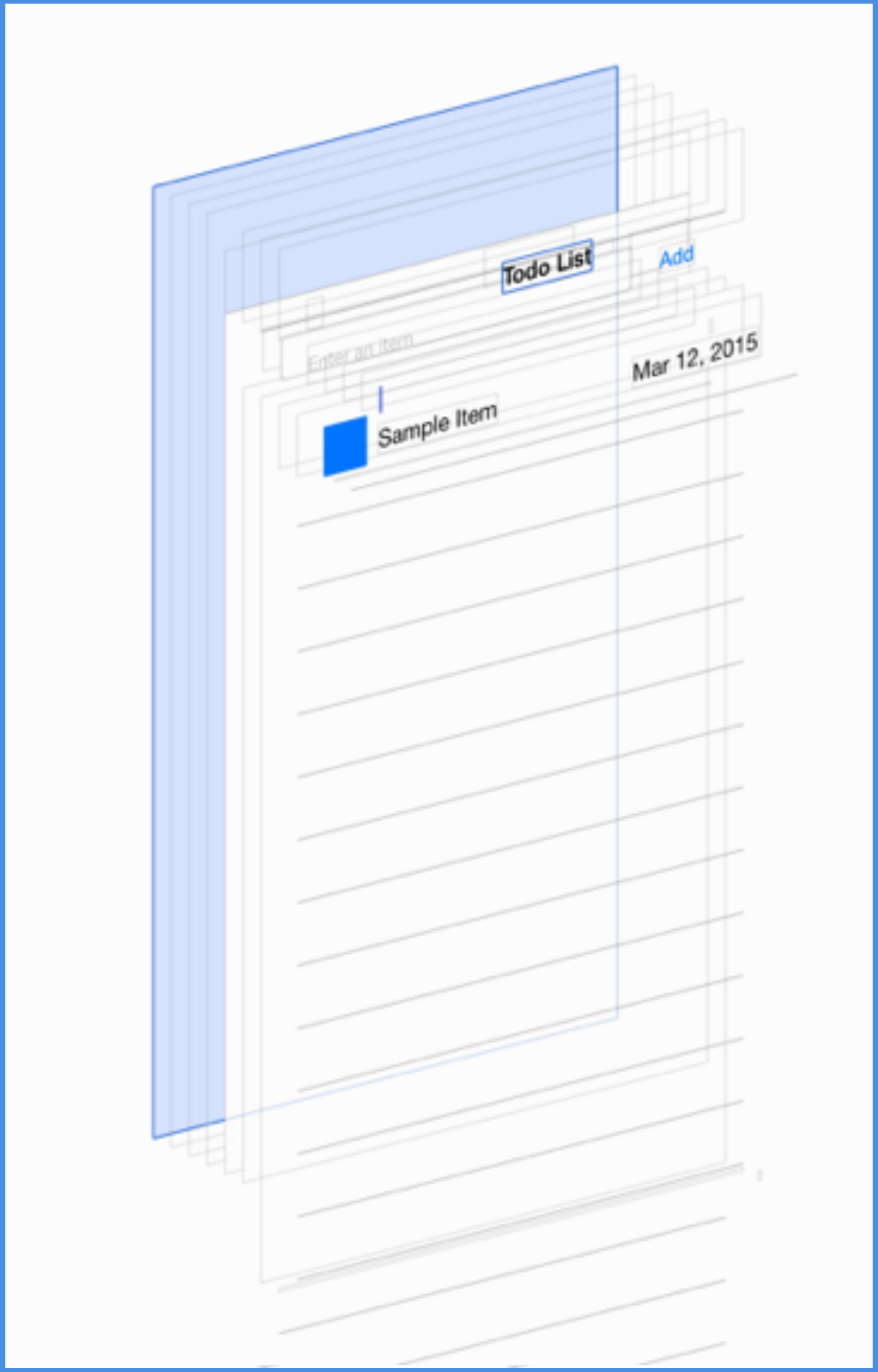
- In the background, our app actually layer cake…

- Like so:



# UIWindow

- And at the base of **everything** we see in our app exists a single **UIWindow**
- This is the one property we saw in our AppDelegate.h file
- Everything else in our UI is added as a “child” of our window

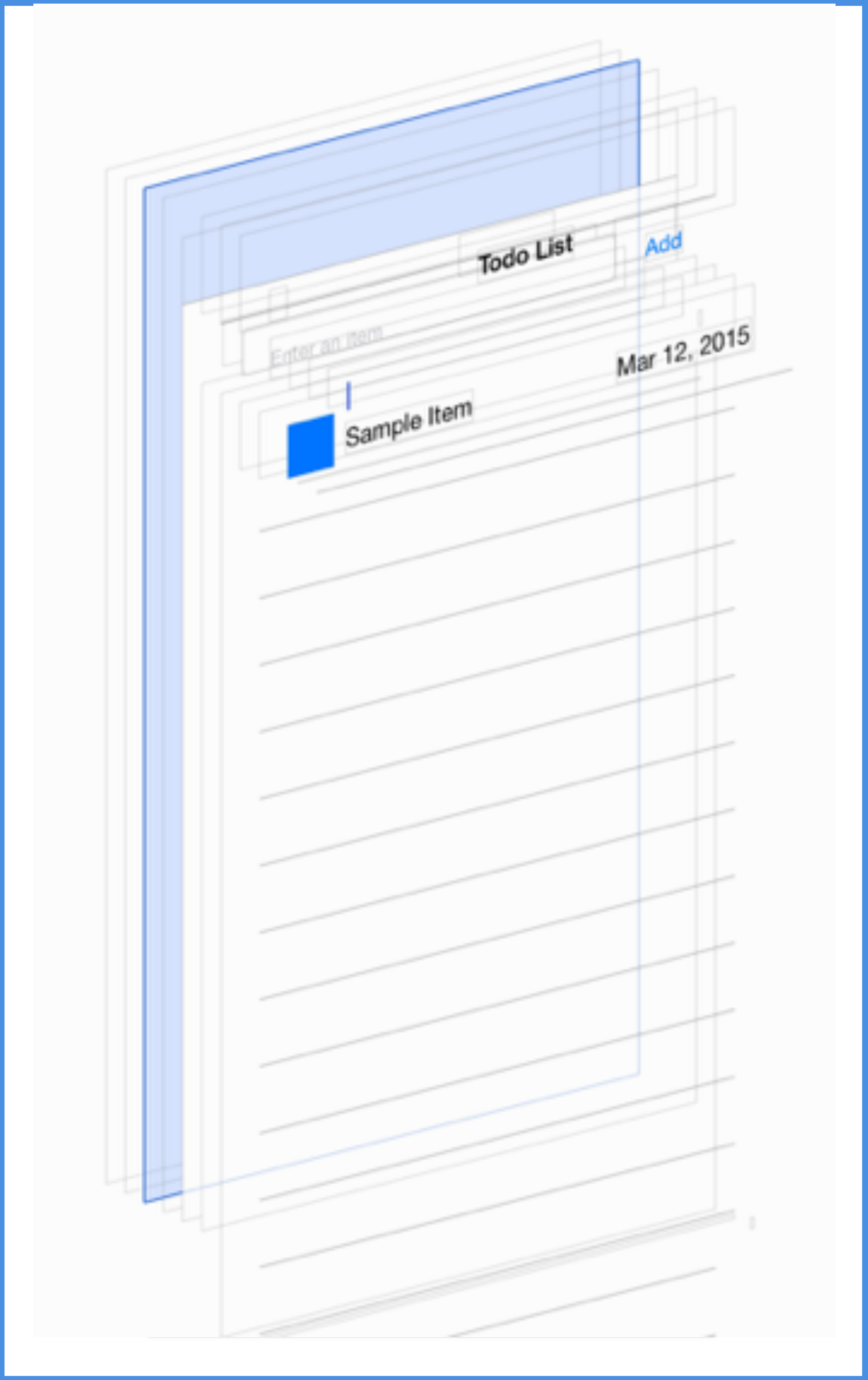


# UINavigationController

- Then, we may have a UINavigationController as the next layer

- Our navigation controller holds our UIViewControllers

    - There may be many

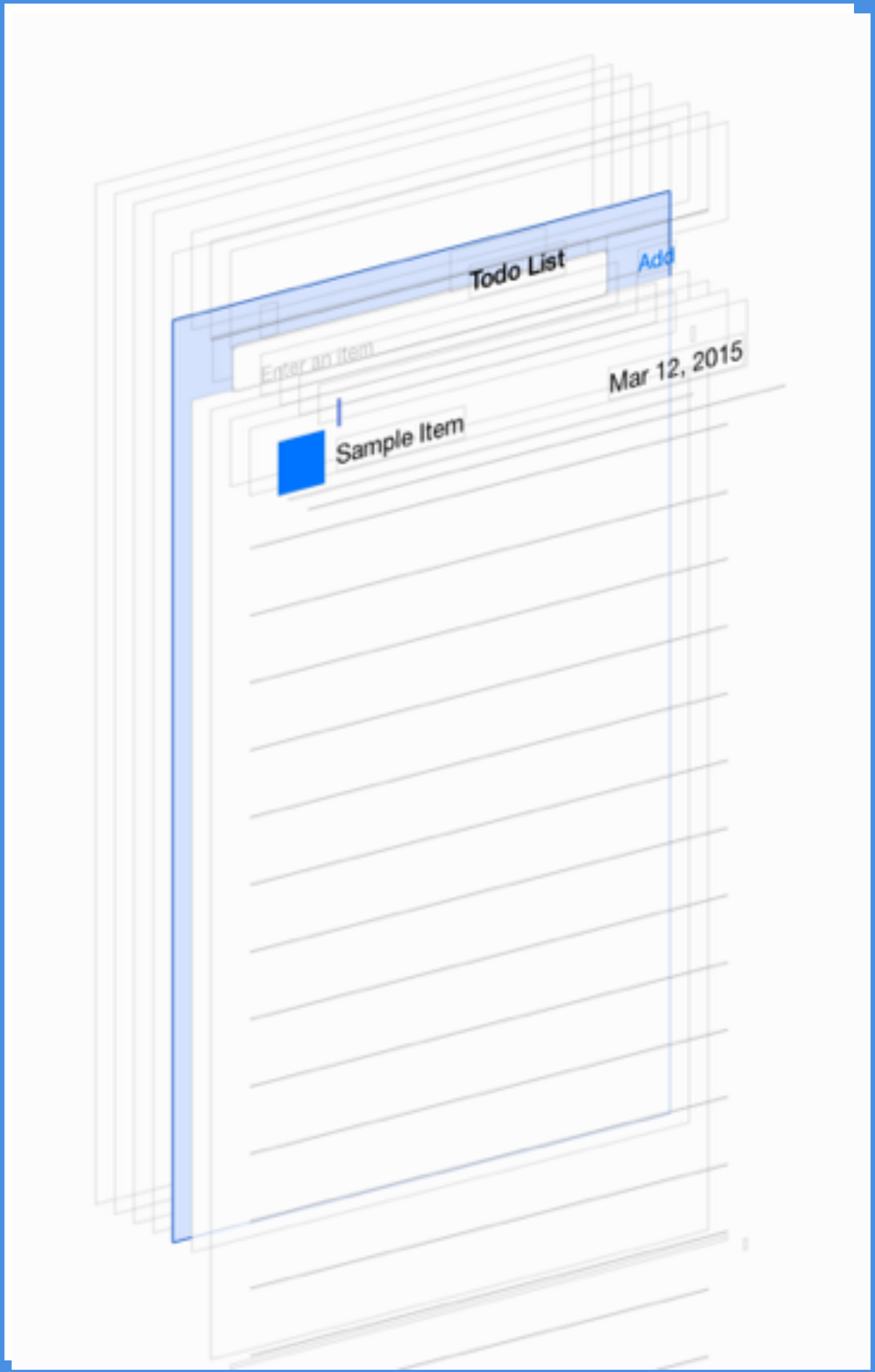


# UIViewController

- Remember: a UIViewController essentially represents a “screen”

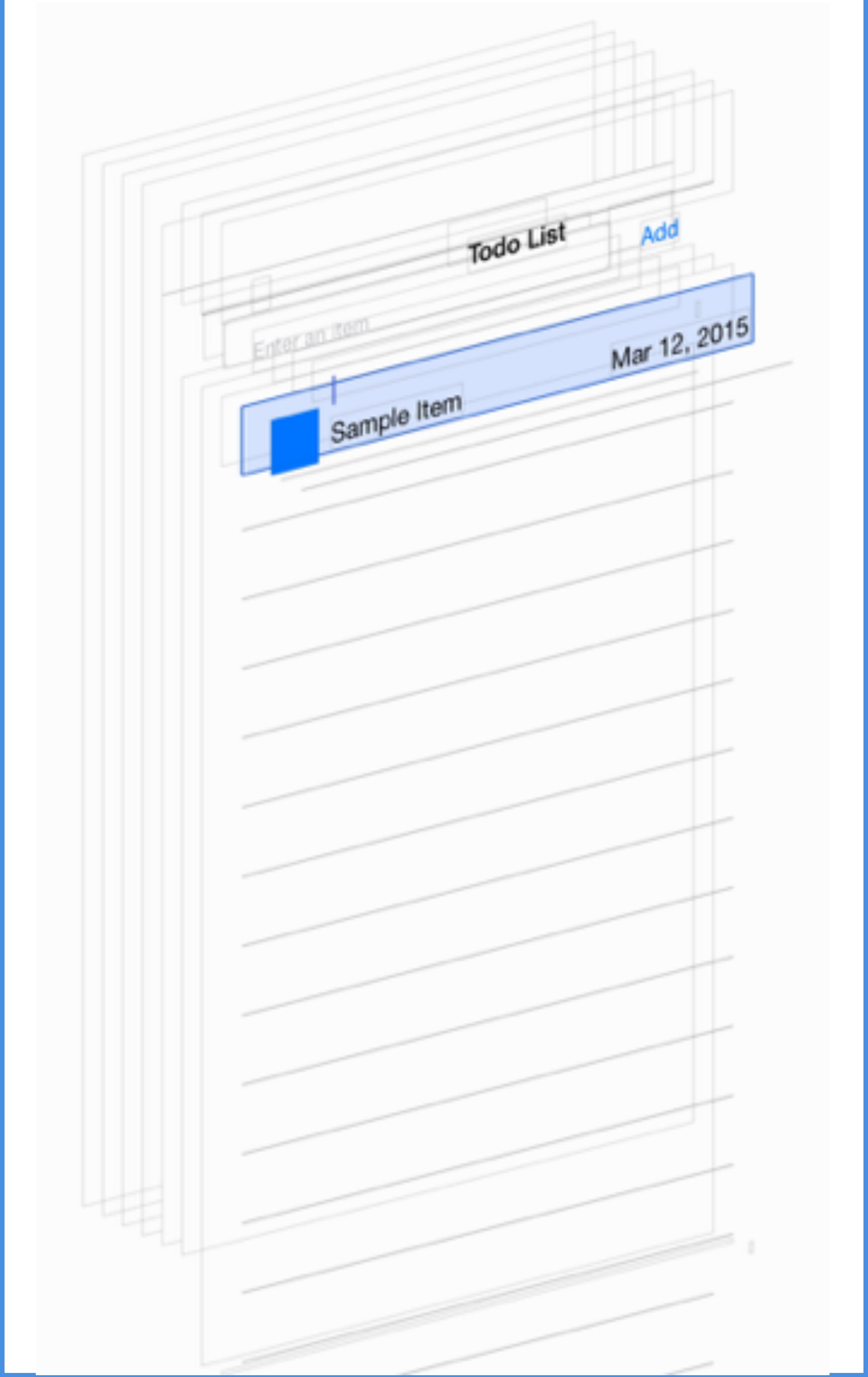
- Our UINavigationController will only show one view controller at a time

- UIViewController has UIView property, and this is the first layer we see in our app



# Visible Subviews

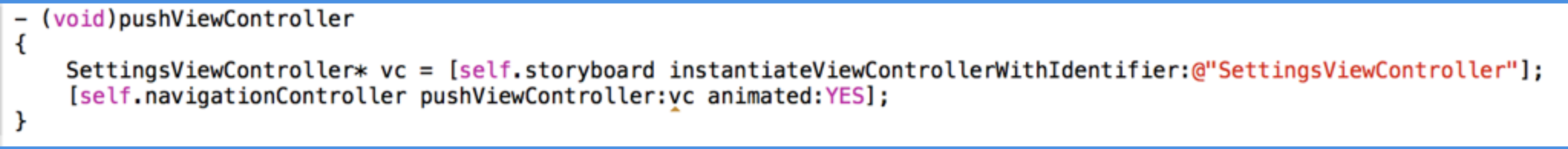
- To our UIViewController's view property (accessed via self.view), we add UIView subclasses
    - ex., UILabel, UIButton, UITableView, etc.
    - Highlighted is a single UITableViewCell



# How do we “push” a new vc?

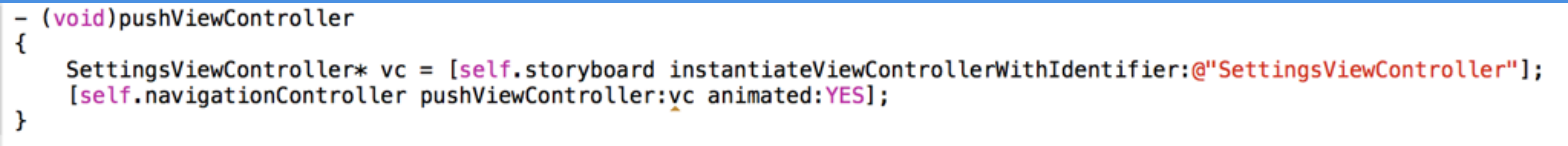
- First, we have to make sure we actually *have* another UIViewController subclass, and a UINavigationController

- Then, assuming we’re using Storyboards, it’s fairly simple:



```
- (void)pushViewController
{
    SettingsViewController* vc = [self.storyboard instantiateViewControllerWithIdentifier:@"SettingsViewController"];
    [self.navigationController pushViewController:vc animated:YES];
}
```

# How do we “push” a new vc?



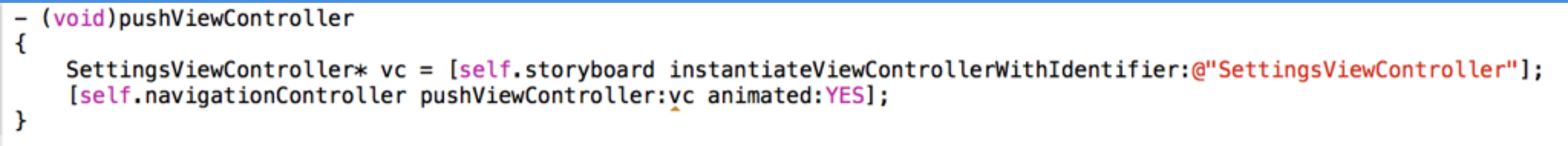
```
- (void)pushViewController
{
    SettingsViewController* vc = [self.storyboard instantiateViewControllerWithIdentifier:@"SettingsViewController"];
    [self.navigationController pushViewController:vc animated:YES];
}
```

- Note: SettingsViewController is an example class name

- We declare a new instance of the view controller, then load it from the Storyboard (using it’s Identifier)

- Similar to how we loaded a custom UITableViewCell from the .xib file

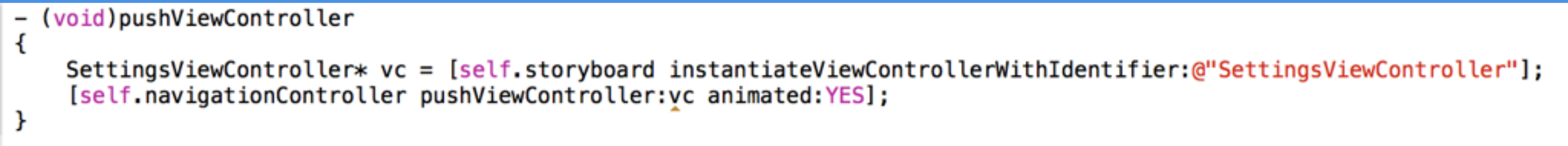
# How do we “push” a new vc?



```
- (void)pushViewController
{
    SettingsViewController* vc = [self.storyboard instantiateViewControllerWithIdentifier:@"SettingsViewController"];
    [self.navigationController pushViewController:vc animated:YES];
}
```

- This code would exist in a UIViewController subclass, so we could access our UINavigationController using:
  - `self.navigationController`
  - Then call the `pushViewController` method

# How do we “push” a new vc?



```
- (void)pushViewController
{
    SettingsViewController* vc = [self.storyboard instantiateViewControllerWithIdentifier:@"SettingsViewController"];
    [self.navigationController pushViewController:vc animated:YES];
}
```

- This code will push a new view controller onto the view controller stack

- When we tap the “back” button, our navigation controller will “pop” the current view controller off the stack. We can manually pop the current vc using:

- `[self.navigationController popViewControllerAnimated:YES];`

# Stacks of Dishes

- The navigation hierarchy is much like a stack of dishes
- I can add a dish to the top of the stack, and that's the dish I see
- I can then take the top dish off, and I'll see the previous dish

Questions?

# Objective-C Blocks

- Like delegations, blocks are a foundational concept in Objective-C
- Prefix: for very simple tasks or programs, you may not see them very often, but when you need a block, you **need** a block

# Objective-C Blocks

- What is a **block**?
- A **block** is a block of code that is contained within curly braces {}
  - A series of statements
- At first, this just sounds like the definition of a method or function

# Objective-C Blocks

- They are similar to functions, **but:**
    - They can be defined **inline** in other method calls

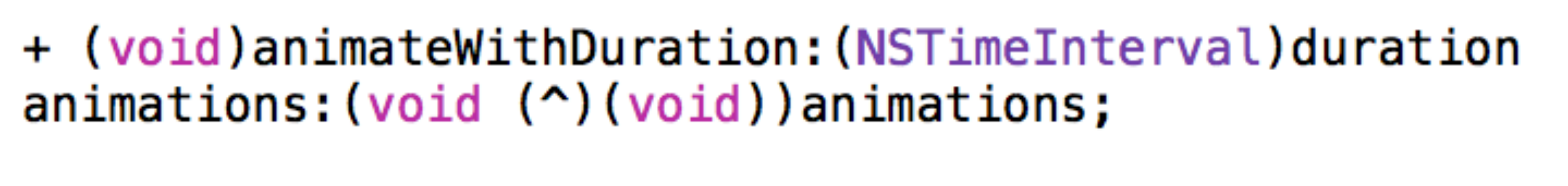
# Block Example

- What does it mean to “define a block of code inline another method”?

- Let’s look at how simple animations work in iOS (with a block)

- We’ll use a UIView method called animateWithDuration

# Block Example

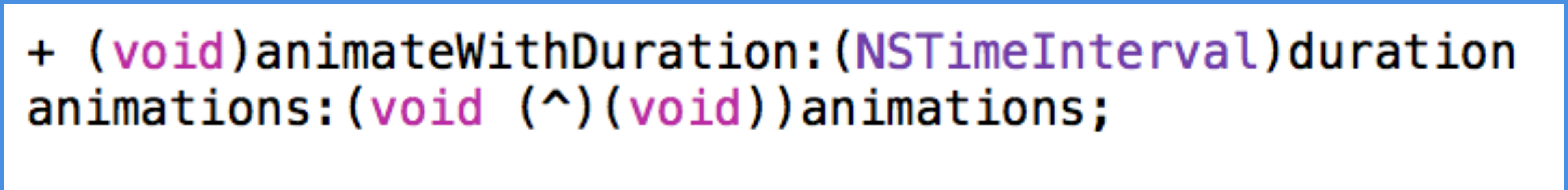


```
+ (void)animateWithDuration:(NSTimeInterval)duration
animations:(void (^)(void))animations;
```

- This is a class method that returns void

- Accepts 2 parameters: an animation duration and a block of code that specifies what we want to animate

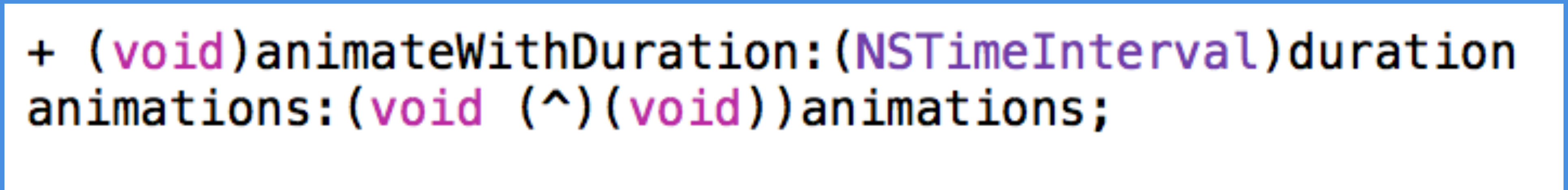
# Block Example



```
+ (void)animateWithDuration:(NSTimeInterval)duration
animations:(void (^)(void))animations;
```

- Imagine we wanted to fade out a label (changing the alpha from 1 to 0)
- If we call self.label.alpha = 0, it would happen instantly (i.e., no fade animation)
- **Instead,** this method takes in a parameter called animations

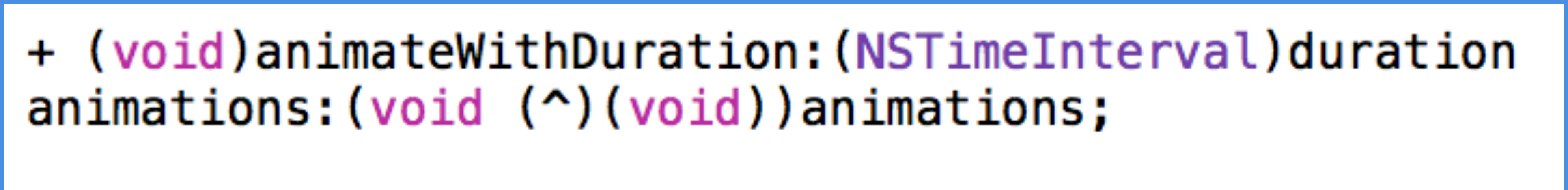
# Block Example



```
+ (void)animateWithDuration:(NSTimeInterval)duration
animations:(void (^)(void))animations;
```

- The animations parameter is a block
- We must pass in a block of code that specifies what changes we want to animate, and animateWithDuration, will execute that code
- Blocks are objects like any other
    - We can define it’s value, pass it around, etc.

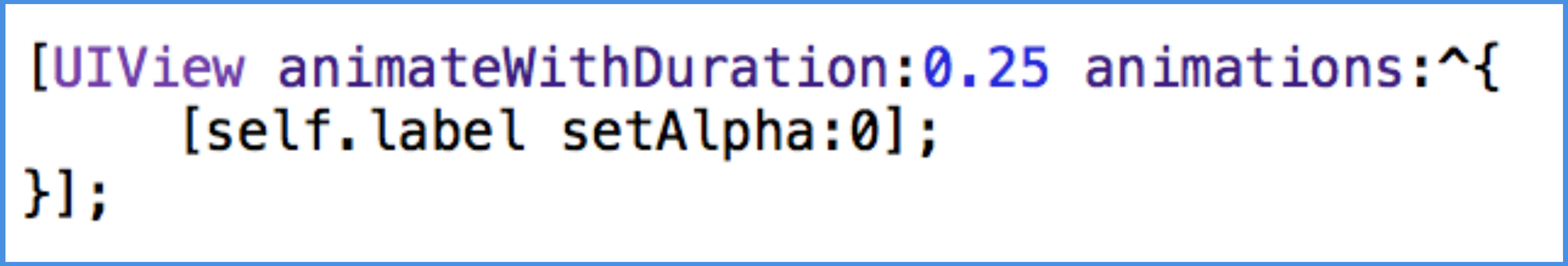
# Block Example



```
+ (void)animateWithDuration:(NSTimeInterval)duration
animations:(void (^)(void))animations;
```

- They are *similar* to function pointers in C++

- How do we **use** this method?

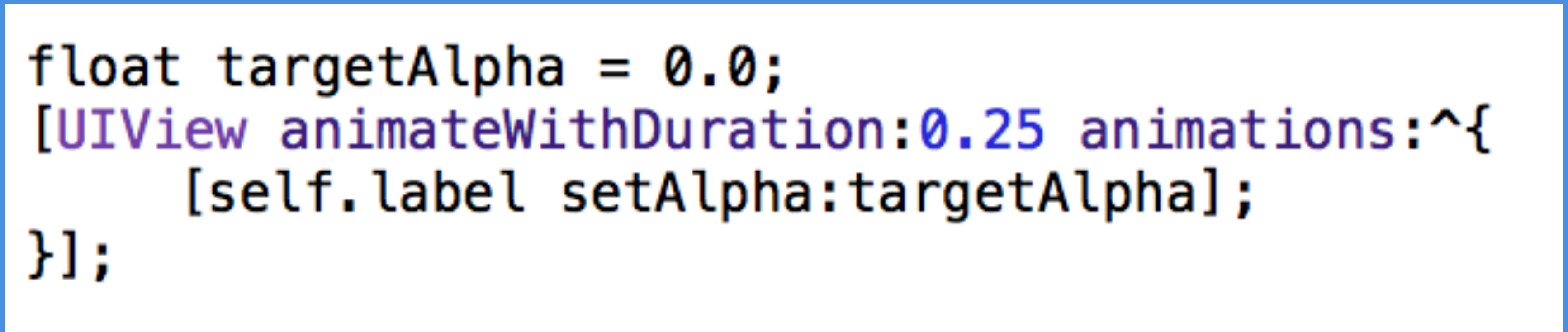


```
[UIView animateWithDuration:0.25 animations:^{
    [self.label setAlpha:0];
}];
```

- Note the ^ (caret)- it indicates a block

# Block Example

- Blocks can also *capture local scope*
- Meaning they can access local variables that are declared *outside* of the block
- So we could do:



```
float targetAlpha = 0.0;
[UIView animateWithDuration:0.25 animations:^{
    [self.label setAlpha:targetAlpha];
}];
```

# Block Example

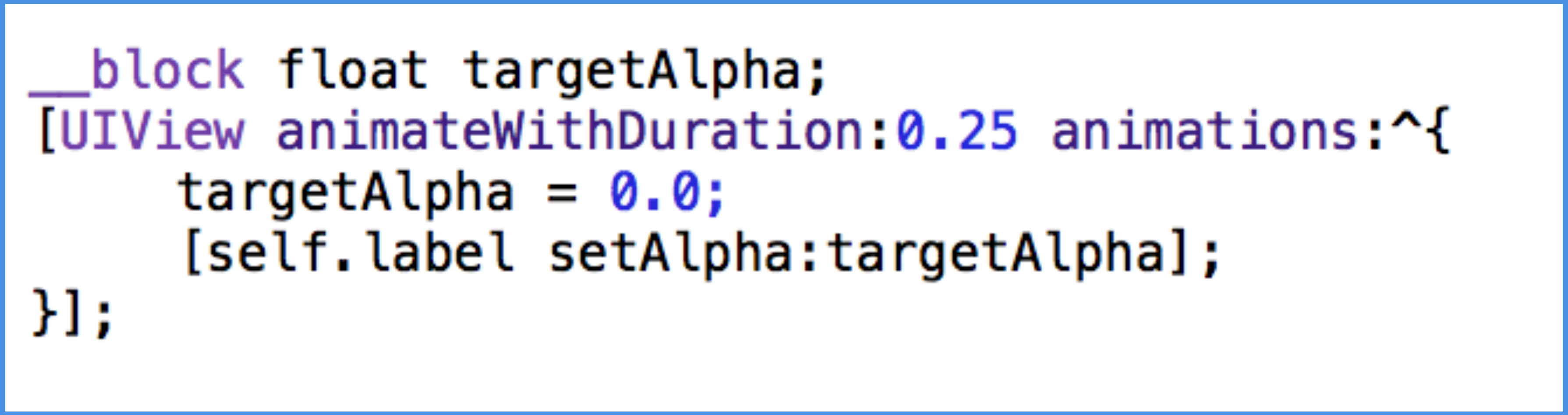
- This will compile and work fine

- However, local variables that are declared outside of the block scope are **read-only** by default

- If we attempted to change the value of targetAlpha in our block, we would get a compile-time error

# Block Example

- However, we can allow a local variable to be writable (i.e., mutable) in a block by using the **`__block`** keyword:



```
__block float targetAlpha;
[UIView animateWithDuration:0.25 animations:^{
    targetAlpha = 0.0;
    [self.label setAlpha:targetAlpha];
}];
```

- This will compile

# Questions?

# Blocks

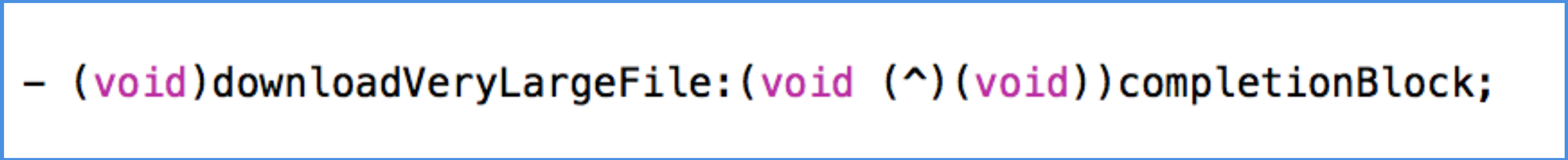
- So far, our block example with animateWithDuration shows us that blocks let us pass in code as a method parameter
- In turn, that method can execute that code in its implementation
- However, there's a much larger reason to use blocks in iOS, and that is to create **callbacks**

# Callbacks with Blocks

- Suppose we wanted to download a very large image from the internet, and alert the user when the download was complete

- Suppose we have a class called DownloadManager that has a method called – downloadVeryLargeFile

# Callbacks with Blocks



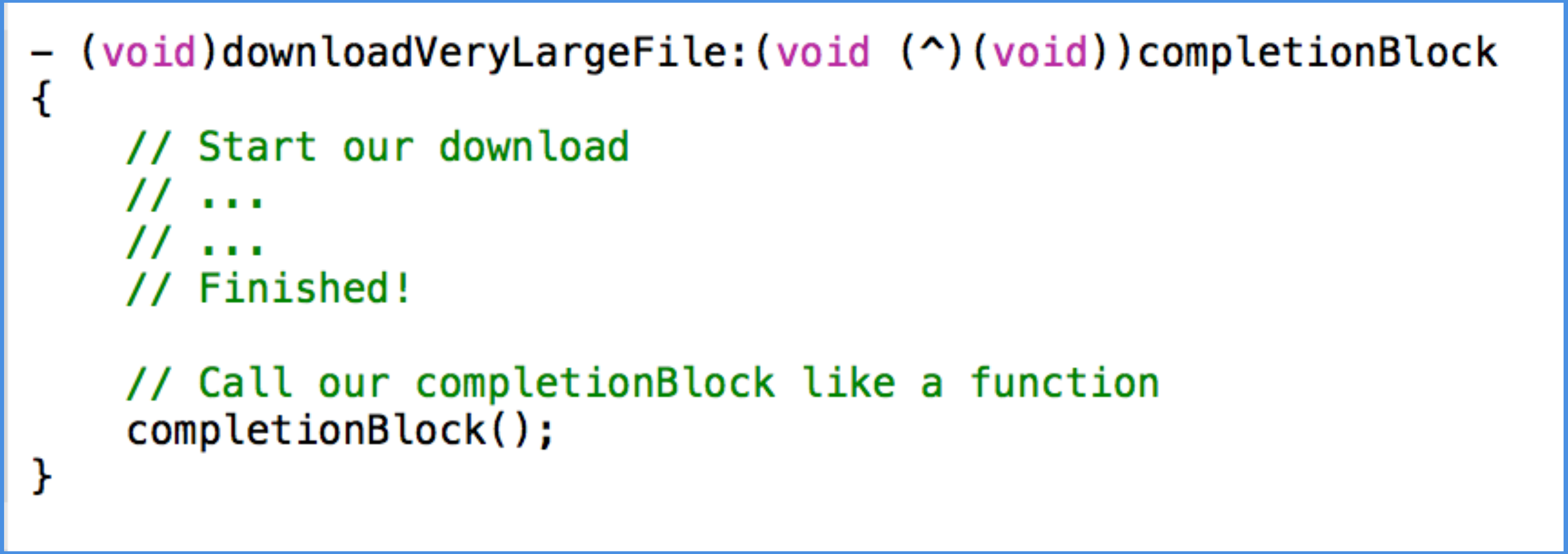
```
- (void)downloadVeryLargeFile:(void (^)(void))completionBlock;
```

- downloadVeryLargeFile has one block parameter (called completionBlock) that doesn’t return anything

- This function would start the download, and when finished, execute the completionBlock parameter

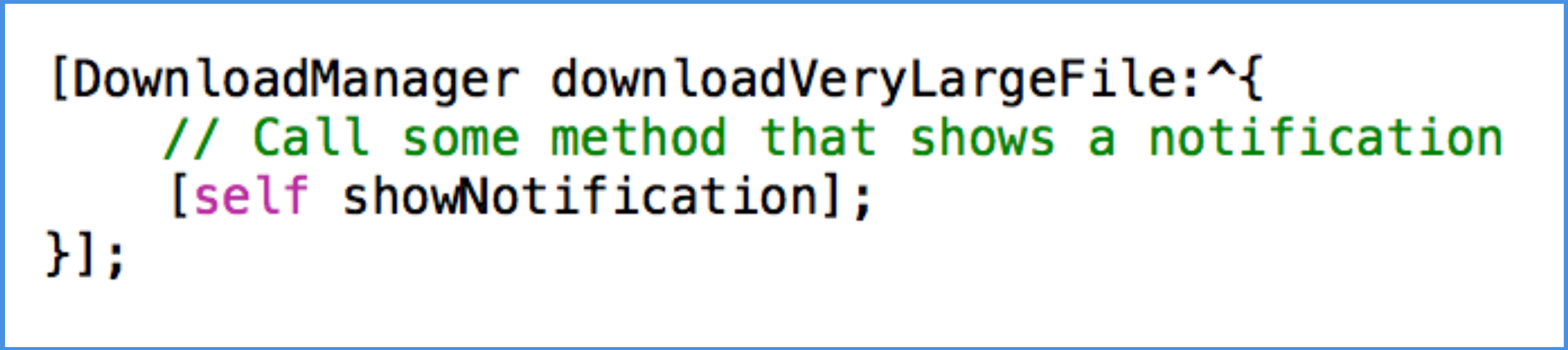
# Callbacks with Blocks

- Our implementation (pseudocode):



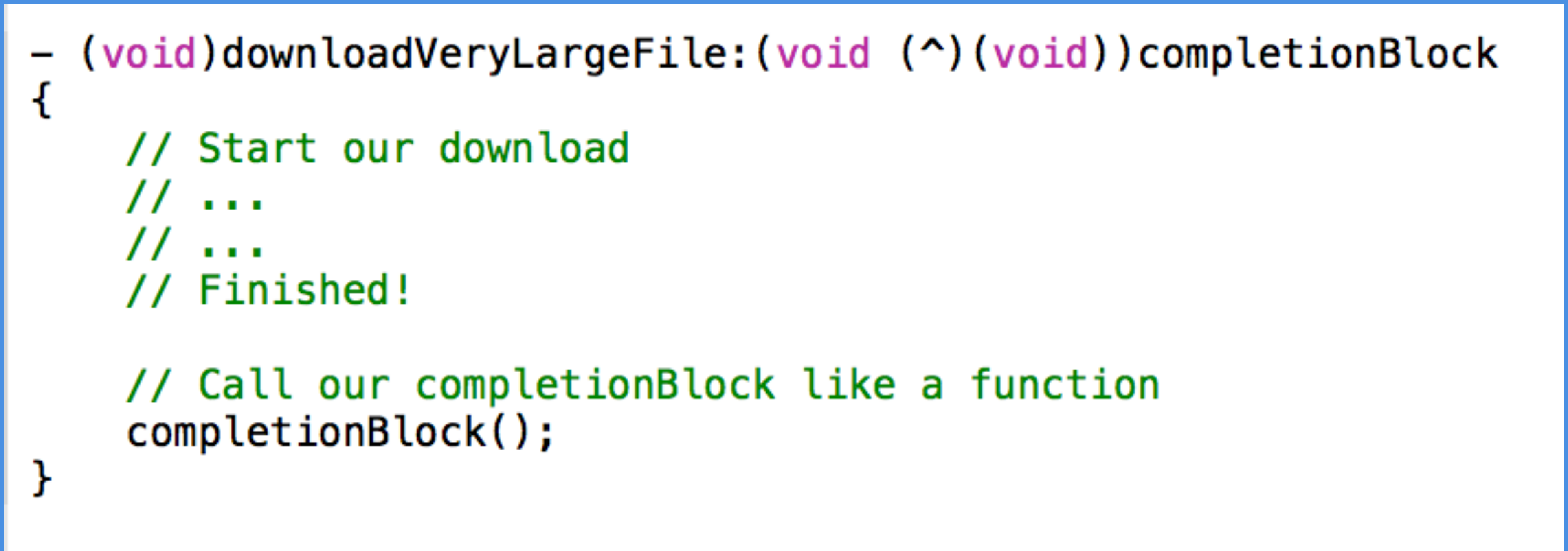
```
- (void)downloadVeryLargeFile:(void (^)(void))completionBlock
{
    // Start our download
    // ...
    // ...
    // Finished!

    // Call our completionBlock like a function
    completionBlock();
}
```

- So if we defined our completionBlock to show the user a notification like this:



```
[DownloadManager downloadVeryLargeFile:^{
    // Call some method that shows a notification
    [self showNotification];
}];
```

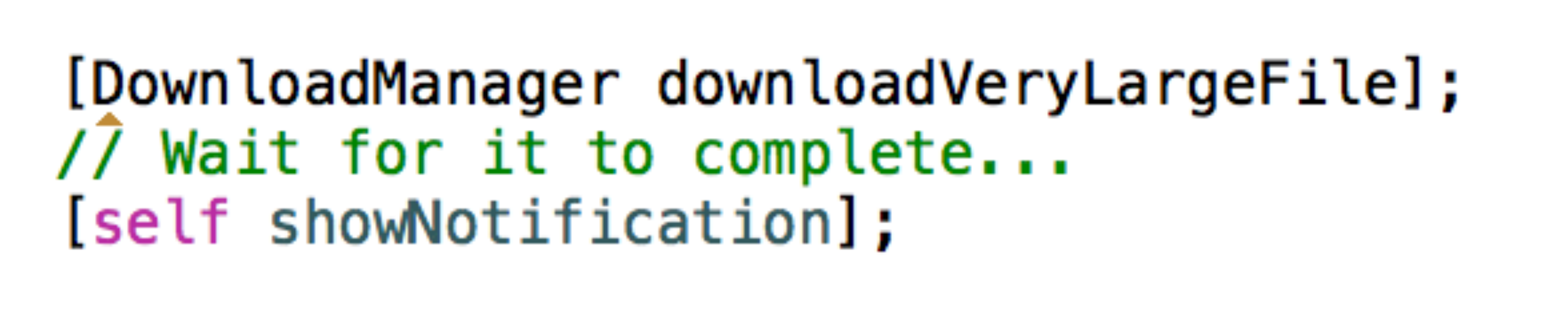
- Then that notification would be called when we call completionBlock() in our method:



```
- (void)downloadVeryLargeFile:(void (^)(void))completionBlock
{
    // Start our download
    // ...
    // ...
    // Finished!

    // Call our completionBlock like a function
    completionBlock();
}
```

# Callbacks with Blocks

- Now, why wouldn’t we just write our code like this?
- It would seem much simpler?



```
[DownloadManager downloadVeryLargeFile];
// Wait for it to complete...
[self showNotification];
```

- To answer this, we must take a step back

# Threading

- The problem has to do with “threading”
- What is a thread?
- A thread represents a single path of execution through our code

# Threading

- A good way to think of a thread is like a grocery store checkout area

- Each “thread” is like an individual employee at their own checkout counter

- If we have one employee working, all the work (i.e., all the customers) must go through that one employee

# Threading

- This could be pretty slow, right?
- If we add more employees and checkout counters, then we can spread customers across them
- They'll all operate at the same time, independent of each other
- We would get the work done much faster

# Threading

- In iOS, when our application launches, we start with one "worker" or **thread**
  - This is the **main thread** or the **UI thread**
  - It is a special thread because it is the only thread allowed to make changes to the user interface
    - i.e., anything with the UI or UIKit
    - ex., setting the text of a cell, changing an image, etc.

# Threading

- This main thread is created in our main() function
- **By default, all code runs on the main thread**
- You could imagine it is very easy and very possible to overwhelm that main thread with too much work
- If we do, we **freeze** the main thread
- That is why our UIs and apps “freeze”

# Solution?

- If we have a computationally expensive task (or one that will take a while) like downloading an image from a remove web server, we **do not** want our main thread to take care of that
- Then our main thread will be busy downloading the image instead of making sure our UI is responsive to touch events, etc.

# Solution?

- We want to delegate that expensive work to another thread (another worker)
- **But remember: these threads are independent of each other, so we are not sure when it will finish**
- This is why we use callbacks

# Solution?

- If we go back to our startDownloadingVeryLargeFile method, our implementation could create a new thread to download the image on

- Then, it will call completionBlock() to alert us the download is complete

- It won’t matter if the download is instantaneous or takes 10 seconds– either way, our main thread won’t be blocked

# Grand Central Dispatch

- So how would we have our method create and run on a new thread?

- Enter Grand Central Dispatch (GCD)

- GCD is a multithreading and concurrency API library written in C

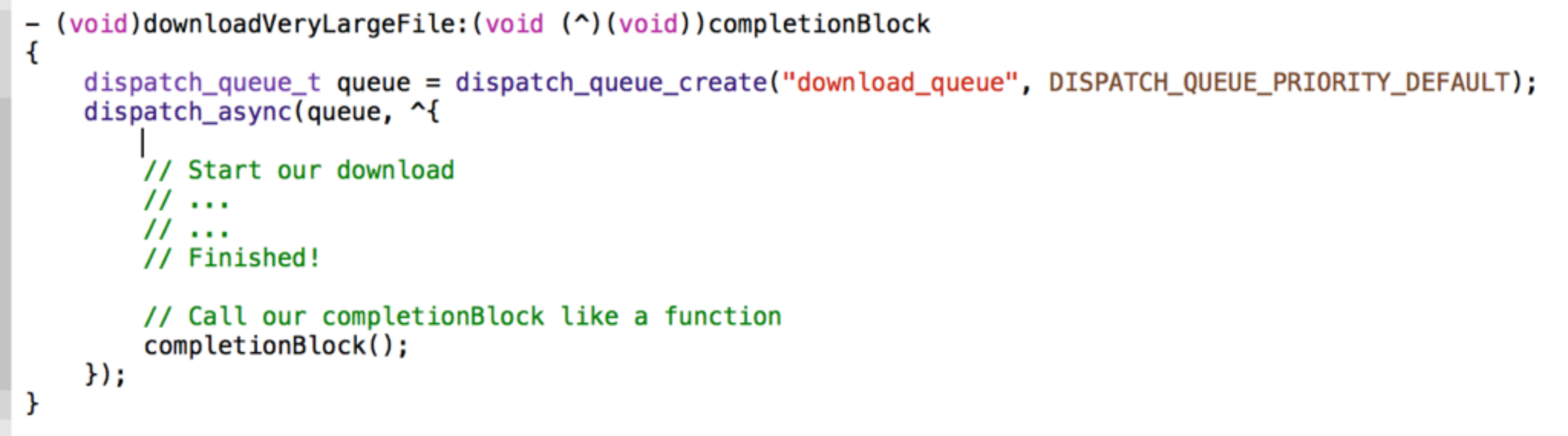
# Grand Central Dispatch

- Sounds scary, but it isn’t
- The idea is every thread has a **queue** of work (much like the queue of a checkout lane)
- Each thread can execute one instruction (i.e., a task) at a time
- When it’s completed one instruction, it moves to the next
- This is called a **serial queue**

# Grand Central Dispatch

- Remember: when we dispatch work to a new thread or queue, we have no guarantees for when that code will finish executing (that’s what callback blocks are for)
- Also, if that thread blocks, it (mostly) won’t affect other threads (i.e., the main thread)
- How do we create a new thread queue?

# Creating a new thread queue

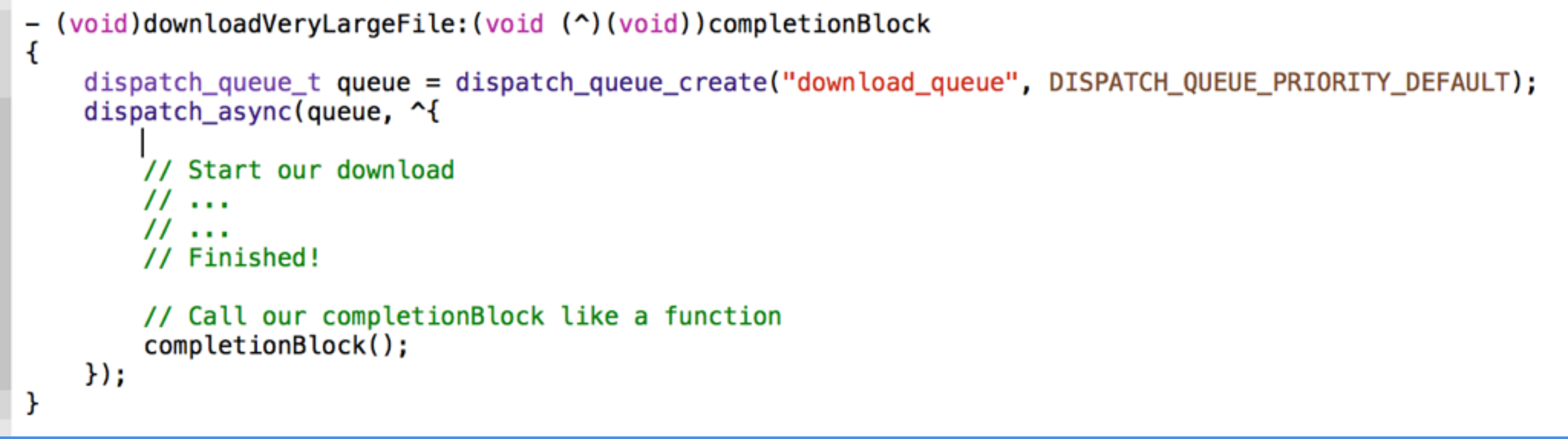


```
- (void)downloadVeryLargeFile:(void (^)(void))completionBlock
{
    dispatch_queue_t queue = dispatch_queue_create("download_queue", DISPATCH_QUEUE_PRIORITY_DEFAULT);
    dispatch_async(queue, ^{

        // Start our download
        // ...
        // ...
        // Finished!

        // Call our completionBlock like a function
        completionBlock();
    });
}
```

- We are first creating a queue variable of type dispatch_queue_t with an identifier and priority

- Then call dispatch_async (dispatch_asynchronous) which adds the block of code to a queue on a newly dispatched thread

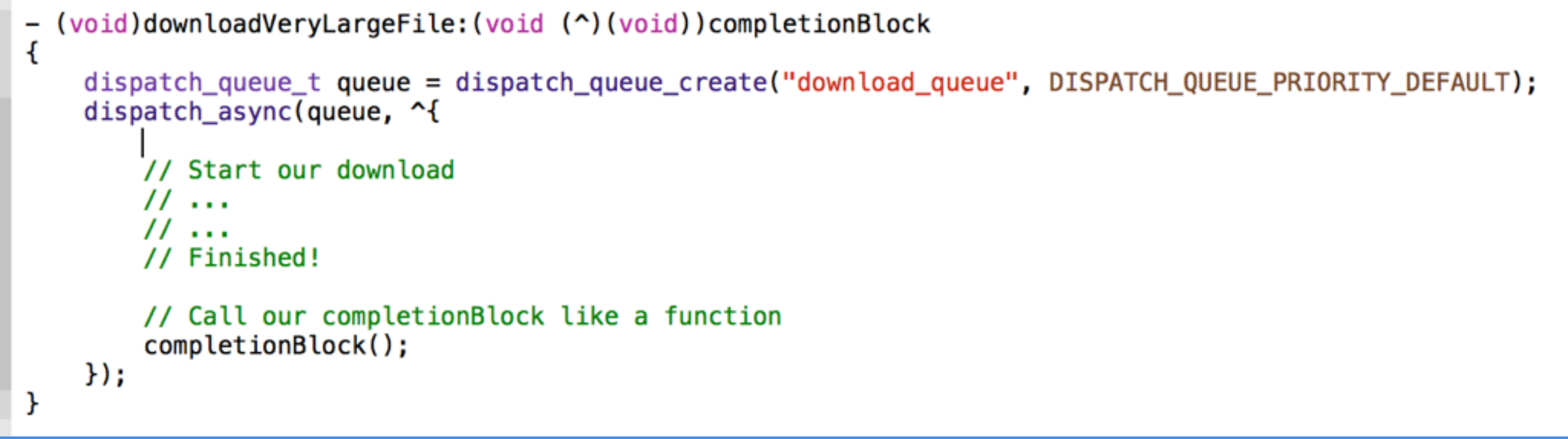
# Creating a new thread queue



```
- (void)downloadVeryLargeFile:(void (^)(void))completionBlock
{
    dispatch_queue_t queue = dispatch_queue_create("download_queue", DISPATCH_QUEUE_PRIORITY_DEFAULT);
    dispatch_async(queue, ^{

        // Start our download
        // ...
        // ...
        // Finished!

        // Call our completionBlock like a function
        completionBlock();
    });
}
```

- You can also access the main thread again using **dispatch_get_main_queue**

# Creating a new thread queue



```
- (void)downloadVeryLargeFile:(void (^)(void))completionBlock
{
    dispatch_queue_t queue = dispatch_queue_create("download_queue", DISPATCH_QUEUE_PRIORITY_DEFAULT);
    dispatch_async(queue, ^{
        
        // Start our download
        // ...
        // ...
        // Finished!
        
        // Call our completionBlock like a function
        completionBlock();
    });
}
```

- There are also higher-level wrappers for GCD in Foundation (i.e., NSOperation), but for most small threading tasks, it’s probably overkill