



Mobile Application Development



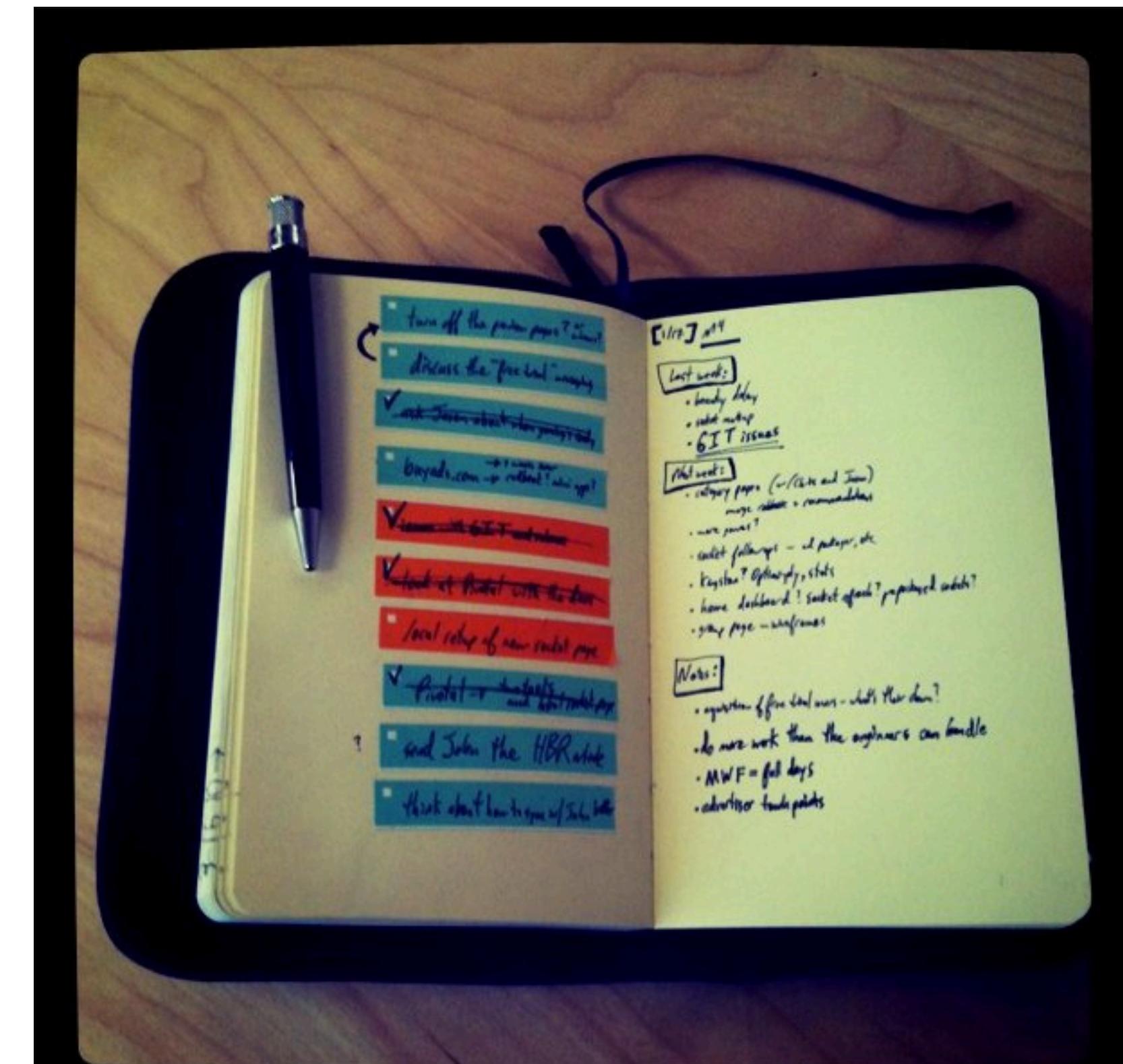
Lecture 1
Introduction to Objective-C



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).

Lecture Summary

- The principles of OOP
- Objective-C language basics
- Classes and objects
- Methods
- Instance variables and properties
- Dynamic binding and introspection
- Foundation framework





References

7

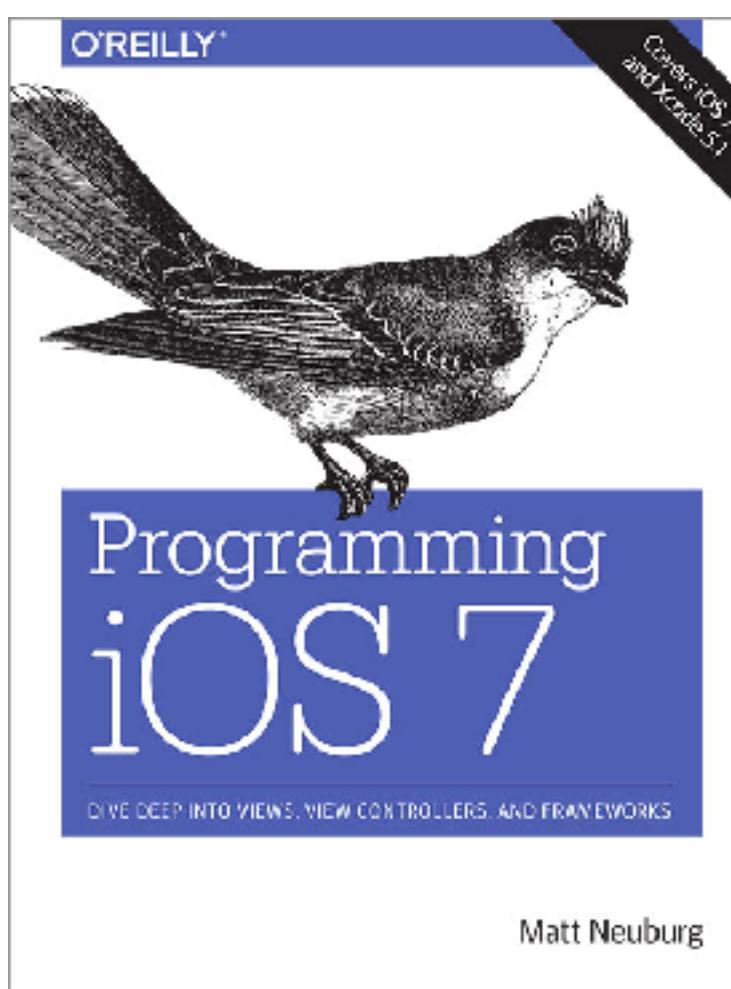
iOS Developers Library

<https://developer.apple.com/library/ios/documentation/cocoa/conceptual/ProgrammingWithObjectiveC>
<https://developer.apple.com/library/ios/navigation/>



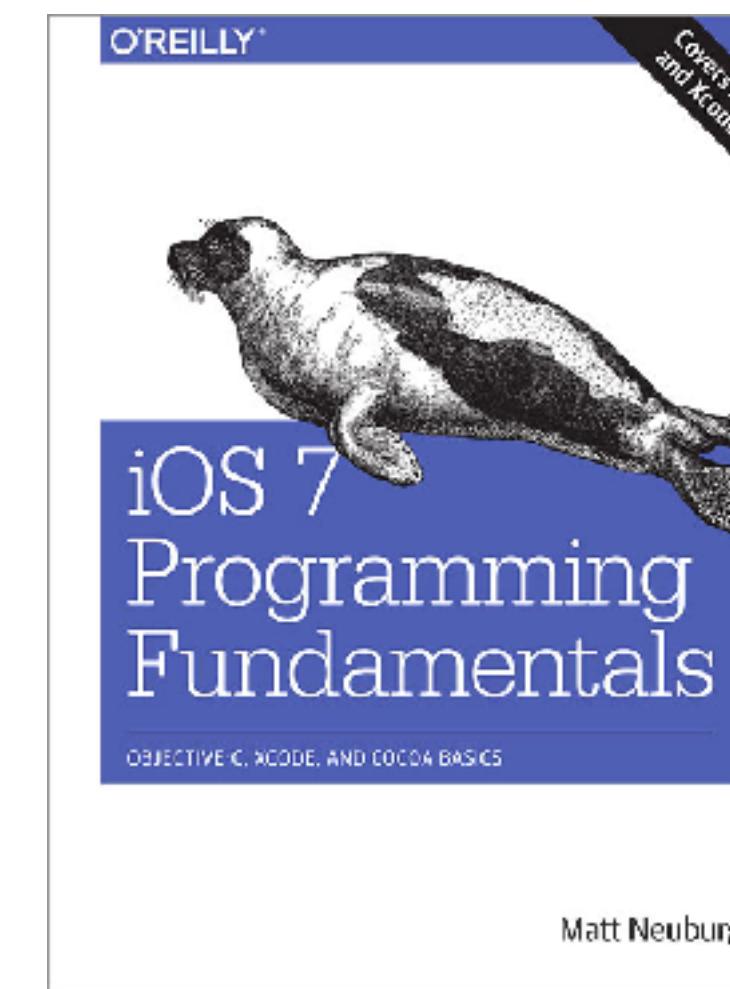
CS193P - Stanford class on iOS

<http://www.stanford.edu/class/cs193p/cgi-bin/drupal/>
<https://itunes.apple.com/us/course/coding-together-developing/id593208016>



Programming iOS 7

by Matt Neuburg
O'Reilly Media - December 2013
<http://shop.oreilly.com/product/0636920031017.do>



iOS 7 Programming Fundamentals

by Matt Neuburg
O'Reilly Media - October 2013
<http://shop.oreilly.com/product/0636920032465.do>



Objective-C

- Objective-C is the primary programming language you use when writing software for OS X and iOS
- Objective-C is a strict superset of the C programming language and provides object-oriented capabilities and a dynamic runtime
- it is possible to compile any C program with the Objective-C compiler
- it is possible to include any C statement within an Objective-C program
- it is even possible to include C++ code inside Objective-C, but the compiler must properly instructed to process C++ code
- Objective-C inherits the syntax, primitive types, and flow control statements of C and adds syntax for defining classes and methods
- All the syntax that is not related to object-oriented features of the language is the same as classical C syntax

Swift

- Since iOS8, Apple introduced a new programming language, named Swift
- Concise and expressive syntax
- Swift introduces safe programming patterns and does not rely on C compatibility
- Swift code can work side-by-side with Objective-C, so it can be used to extend existing apps
- Available since XCode 6



```
var square = 0
var diceRoll = 0
while square < finalSquare {
    // roll the dice
    if ++diceRoll == 7 { diceRoll = 1 }
    // move by the rolled amount
    square += diceRoll
    if square < board.count {
        square += board[square]
    }
}
println("Game over!")
```

<https://developer.apple.com/swift/>

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/



Object-Oriented Programming

- OOP is a programming paradigm that aims at building programs that are:
 - **highly modular**
 - **well-structured**
 - **easy to maintain**
 - **easy to extend**
- OOP focuses also on code re-usability



Object-Oriented Programming

- In OOP, a program is composed of a set (graph) of interacting **objects**, representing concepts related to the specific scenario (domain)
- An object represents is a data structure which is composed by the aggregation of:
 - **state**: the actual data that the object operates on (**attributes**)
 - **behavior**: the functionalities that allow to operate with the object and its data (**methods**)
- An object can interact with another one by invoking one of its methods
- Objects have distinct responsibilities and are to be considered as independent “black-boxes”
- OOP aims at writing complex programs that are easy to manage, maintain, test, and debug
 - ... but to reach this goal, a good design is needed
 - ... and good design comes with experience!



Classes and objects

- **Classes are blueprints (or prototypes) for objects**
- **Objects are instances of a class**
- A class defines the structure and behavior that all the instances of that class share
- A class should be highly specialized in order to provide some behavior:
 - representing data
 - controlling a user interface
 - performing network activity
 - working with a database
 - ...



The 3 principles of OOP

- In order to be effective, OOP defines 3 basic principles that must be followed:
 - **Inheritance**
 - **Encapsulation**
 - **Polymorphism**



Inheritance

- **Inheritance** is a fundamental principle of Object-Oriented Programming
- Inheritance allows to define new functionalities of a class by **extending** it
- The class that is being extended is called superclass
- The process of extending a class is called **subclassing**
- A class inherits all the variables and methods defined in its superclass
- All classes therefore define a hierarchy which ultimately defines which functionalities the instances of each class have
- Inheritance creates a semantic “is-a” relationship
- Anything a superclass can do, its subclasses can do



Encapsulation

- **Encapsulation** is a fundamental principle of Object-Oriented Programming
- Encapsulation means that data and the methods to operate on data are packed into the same component (a class)
- According to the encapsulation principles, only the interface (i.e., the methods to operate on data) should be accessible and visible from outside an object, while the implementation (i.e., the details of how the manipulation is performed) should be hidden
- Encapsulation allows to treat objects as black-boxes, which provide some functionality but hide the details of their operations
- Different levels of visibility can be defined by marking methods as **private**, **protected**, or **public**



Polymorphism

- **Polymorphism** is the third fundamental principle of Object-Oriented Programming
- With inheritance, all the subclasses inherit the same methods of their superclass
- It is possible for subclasses to override the methods of their superclasses
- Since we can always treat an object as a member of one of its superclasses, the actual behavior that it will perform when one of its methods is invoked depends on its concrete type
- The ability to adapt the behavior to the concrete type is called polymorphism

Good practices

- A good principle for better OOP code is to define classes in such a way that:
 - classes are very specialized in doing a few things
 - classes expose only the methods that must be used in order to interact with them
 - classes can be treated as “black-boxes” that provide some functionality
- Avoid classes that do too many things: they can be probably split into more classes!
- Use accessor methods (getters and setters) to interact with the data
- Always try to define interfaces that are useful for your problem: it is much easier to **think about what objects can do**, rather than having to think about the functionalities that they inherit



Objective-C programs

- Objective-C programs are split into two main types of files:
- **.h** files: define the interfaces, that is which functionalities are exposed to other parts of the code; classes, data structures, and methods are defined in the interface file
- **.m** files: implement the interfaces defined in the corresponding .h file
- In case an implementation file contains C++ code, it must be named using the **.mm** extension
- Only the interface can be accessed from other parts of your code; the implementation file is opaque!

```
//  
// MDPoi.h  
// iMobdev  
//  
// Created by Simone Cirani on 30/09/13.  
// Copyright (c) 2013 Università degli Studi  
di Parma. All rights reserved.  
  
#import <Foundation/Foundation.h>  
  
@interface MDPoi : NSObject{  
    NSString *_name;  
    double _latitude;  
    double _longitude;  
}  
  
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude;  
- (double) latitude;  
- (double) longitude;  
  
@property (nonatomic,copy) NSString *name;  
- (void) log;  
@end
```

.h

```
//  
// MDPoi.m  
// iMobdev  
//  
// Created by Simone Cirani on 30/09/13.  
// Copyright (c) 2013 Università degli Studi  
di Parma. All rights reserved.  
  
#import "MDPoi.h"  
  
@implementation MDPoi  
  
@synthesize name = _name;  
  
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{  
    NSLog(@"%@", __FUNCTION__);  
    if(self = [super init]) {  
        _name = [name copy];  
        _latitude = latitude;  
        _longitude = longitude;  
    }  
    return self;  
}  
  
- (double) latitude{  
    return _latitude;  
}  
  
- (double) longitude{  
    return _longitude;  
}  
- (void) log{
```

.m



Defining classes

- Classes are defined in the interface file, between the `@interface` directive and the corresponding `@end`
- Everything inside the interface block defines the class' structure (instance variables) and functionalities

```
#import <Foundation/Foundation.h>

@interface MDPOI : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (void)setLatitude:(double)lat;

@end
```

MDPOI.h



Defining classes

- Let's define a new class called MDPOI, used to hold a point of interest, with name, latitude, and longitude

```
#import <Foundation/Foundation.h>
```

```
@interface MDPOI : NSObject{
```

```
NSString *_name;  
double _latitude;  
double _longitude;
```

```
}
```

```
- (void)setLatitude:(double)lat;
```

```
@end
```



Defining classes

- Let's define a new class called MDPOI, used to hold a point of interest, with name, latitude, and longitude

```
#import <Foundation/Foundation.h>
```

← Import the header for **NSObject** class

```
@interface MDPOI : NSObject{
```



```
NSString *_name;
```

```
double _latitude;
```

```
double _longitude;
```


}


```
- (void)setLatitude:(double)lat;
```



```
@end
```



Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

```
#import <Foundation/Foundation.h>
```

```
@interface MDPoi : NSObject{
```

```
NSString *_name;  
double _latitude;  
double _longitude;
```

```
}
```

```
- (void)setLatitude:(double)lat;
```

```
@end
```

Declare the **MDPoi** class

Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

```
#import <Foundation/Foundation.h>
```

```
@interface MDPoi : NSObject{
```

```
NSString *_name;  
double _latitude;  
double _longitude;
```

```
}
```

```
- (void)setLatitude:(double)lat;
```

```
@end
```



Extends the NSObject class



Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

```
#import <Foundation/Foundation.h>
```

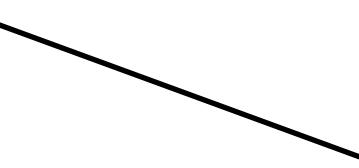
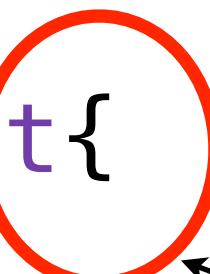
```
@interface MDPoi : NSObject{
```

```
NSString *_name;  
double _latitude;  
double _longitude;
```

```
}
```

```
- (void)setLatitude:(double)lat;
```

```
@end
```



Instance variables must be declared between curly braces

Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

```
#import <Foundation/Foundation.h>
```

```
@interface MDPOI : NS0bject{
```

```
NSString *_name;  
double _latitude;  
double _longitude;
```

```
}
```

Instance variables must be declared between curly braces

```
- (void)setLatitude:(double)lat;
```

```
@end
```



Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

```
#import <Foundation/Foundation.h>

@interface MDPoi : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (void)setLatitude:(double)lat;
```

← Methods must be defined outside the curly braces

```
@end
```



Implementing classes

- Now, let's implement the MDPOI class defined in the MDPOI.h interface file

```
#import "MDPoi.h"

@implementation MDPoi

- (void)setLatitude:(double)lat{
    _latitude = lat;
}

@end
```

MDPoi.m

Implementing classes

- Now, let's implement the MDPOI class defined in the MDPOI.h interface file

```
#import "MDPOI.h" ← Import the interface header file
```

```
@implementation MDPOI
```

```
- (void)setLatitude:(double)lat{  
    _latitude = lat;  
}
```

```
@end
```

Implementing classes

- Now, let's implement the MDPoi class defined in the MDPoi.h interface file

```
#import "MDPoi.h"
@implementation MDPoi
- (void)setLatitude:(double)lat{
    _latitude = lat;
}
@end
```

The implementation of the interface must be between the **@implementation** directive and the corresponding **@end**



Implementing classes

- Now, let's implement the MDPoi class defined in the MDPoi.h interface file

```
#import "MDPoi.h"

@implementation MDPoi

- (void)setLatitude:(double)lat{
    _latitude = lat;
}

@end
```

The declared methods must be implemented
(a warning is issued if not, but no error - only at runtime)



Special keywords: **id**, **nil**, **BOOL**, and **self**

- In Objective-C, all objects are allocated in the heap, so to access them we always use a pointer to the object
- **id** means a pointer to an object of any kind (similar **void*** in C)
- **nil** is the value of a pointer that points to nothing (**NULL** in C)
- **BOOL** is the type defined (**typedef** in file **objc.h**) by Objective-C for boolean values
 - **YES** == 1 (true)
 - **NO** == 0 (false)
- **self** means a pointer to the current object (similar **this** in Java)

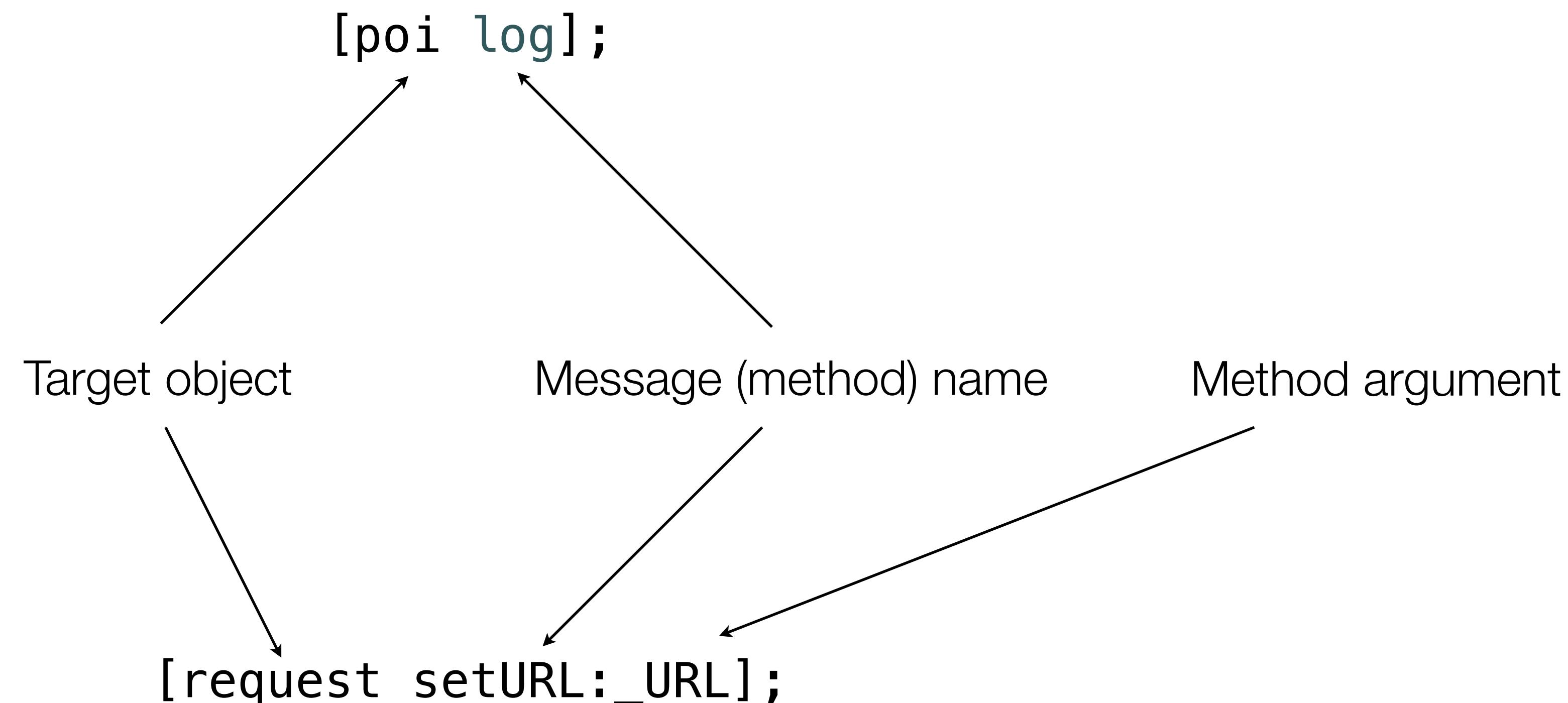


Objective-C methods

- Objective-C objects talk to each other by sending messages
- In Objective-C, method invocation is based on **message passing** to objects
- Message passing differs from classical method calls because the method to be executed is not bound to a specific section of code (in most cases at compile-time) but **the target of the message is dynamically resolved at runtime**, so it might happen that the receiver won't respond to that message (no type checking can be performed at compile-time)
- If the receiver of a message does not respond to the message, an exception is thrown
- It is possible to send a message to **nil**: no NullPointerException is thrown (0 is returned)

Sending messages

- Square brackets are used to send messages





Method syntax

- `(NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;`

Method syntax

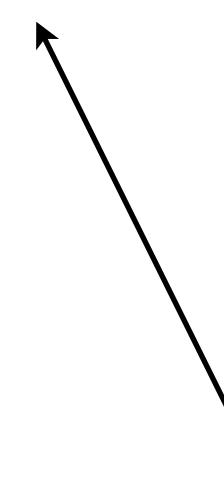
- `(NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;`

- **Instance methods** (relative to the particular object that is the target of the message) begin with a minus sign;
 - they can access instance variables
- **Class methods** (relative to the class itself) begin with a plus sign;
 - accessible through the class name (no need for an instance)
 - they cannot access instance variables
 - utility methods and allocation methods are typically class methods



Method syntax

- **(NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;**



Return type (between parentheses)

Method syntax

- `(NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;`



First part of
method name



Second part of
method name

This method's name is `pointsWithinRange:fromPoi:`

Method syntax

- `(NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;`



First argument of type
double



Second argument of type
MDPoi*



Methods in Objective-C

- Method names in Objective-C sound like sentences natural language
 - pointsWithinRange:fromPoi:
- Some methods have very long names:
 - `(BOOL)tableView:(UITableView *)tableView canPerformAction:(SEL)action forRowAtIndexPath:(NSIndexPath *)indexPath withSender:(id)sender;`
- `BOOL x = [tableView:myTable canPerformAction:myAction forRowAtIndexPath:myIP withSender:s];`
- In order to keep the code readable, it is better to align colons



Invoking Methods in Objective-C

- Suppose a method is defined as follows:
 - `(NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)obj;`
- It can be invoked in the following way:

```
NSArray *list = [self pointsWithinRange:10.0f
                                    fromPoi:poi];
```



Instance variables

- By default, instance variables are **@protected**
- **@private**: can be accessed only within the class itself
- **@protected**: can be accessed directly within the class itself and its subclasses
- **@public**: can be accessed anywhere in the code (not a good OOP practice!)
- Good practice in OOP is to mark all instance variables as private and use getter and setter methods to access them

MDPoi class revised

```
#import <Foundation/Foundation.h>
```

```
@interface MDPoi : NSObject{
```

```
    @private
```

```
        NSString *_name;
```

```
        double _latitude;
```

```
        double _longitude;
```

```
}
```

```
    - (double)latitude;
```

```
    - (void)setLatitude:(double)lat;
```

```
    - (double)longitude;
```

```
    - (void)setLongitude:(double)lon;
```

```
@end
```



MDPoi class revised

```
#import <Foundation/Foundation.h>
```

```
@interface MDPoi : NSObject{
```

```
    @private
```

```
        NSString *_name;  
        double _latitude;  
        double _longitude;
```

```
}
```

- (double)latitude;
- (void)setLatitude:(double)lat;

- (double)longitude;
- (void)setLongitude:(double)lon;

```
@end
```



MDPoi class revised

```
#import <Foundation/Foundation.h>
```

```
@interface MDPoi : NSObject{
```

```
    @private
```

```
        NSString *_name;
```

```
        double _latitude;
```

```
        double _longitude;
```

```
}
```

```
- (double)latitude; ← getter method for _latitude ivar
```

```
- (void)setLatitude:(double)lat;
```

```
- (double)longitude;
```

```
- (void)setLongitude:(double)lon;
```

```
@end
```

MDPoi class revised

```
#import <Foundation/Foundation.h>
```

```
@interface MDPoi : NSObject{
```

```
    @private
```

```
        NSString *_name;
```

```
        double _latitude;
```

```
        double _longitude;
```

```
}
```

```
        - (double)latitude;
```

```
        - (void)setLatitude:(double)lat;
```



setter method for _latitude ivar

```
        - (double)longitude;
```

```
        - (void)setLongitude:(double)lon;
```

```
@end
```



MDPoi class revised

```
#import <Foundation/Foundation.h>

@interface MDPoi : NSObject{

@private
    NSString *_name;
    double _latitude;
    double _longitude;

}

- (double)latitude;
- (void)setLatitude:(double)lat;

- (double)longitude;
- (void)setLongitude:(double)lon;

@end
```

MDPoi.h

```
...

- (double) latitude{
    return _latitude;
}

- (void) setLatitude:(double)lat{
    _latitude = lat;
}

- (double) longitude{
    return _longitude;
}

- (void) setLongitude:(double)lon{
    _longitude = lon;
}

...
```

MDPoi.m



Accessing ivar with getters and setters

```
// get the value of _latitude
double lat = [poi latitude];

// set the value of _latitude
[poi setLatitude:12.0f];
```

Properties

- Properties are a convenient way to get/set the value of instance variables using dot notation
- The main advantage is to avoid the congestion of brackets in the code and to allow for more readable concatenated method invocations
- It is VERY IMPORTANT to pay attention to the lowercase and uppercase letters in the method name:
the compiler can understand the dot notation only if the names of the getter and setter methods are correct!

```
double lat = [poi latitude]; ←→ double lat = poi.latitude;  
[poi setLatitude:12.0f]; ←→ poi.latitude = 12.0f;
```

Compiler-generated properties

- It is possible to define all the getters and setters by hand, but let the compiler help you do the job
- The **@property** keyword instructs the compiler to generate the getter and setter definitions for you
- `(double)latitude;` \longleftrightarrow **@property double latitude;**
`(void)setLatitude:(double)lat;`
- If only a getter method is needed it is possible to specify it to the compiler by adding the “readonly” keyword after **@property**
 - `(double)latitude;` \longleftrightarrow **@property (readonly) double latitude;**



Properties

- Properties can also be defined without an instance variable that matches it

```
@property (readonly) NSString *detail;
```

.h

```
- (NSString *)detail{
    return [NSString stringWithFormat:@"%@ - (%f,%f)",
           _name, _latitude, _longitude];
}
```

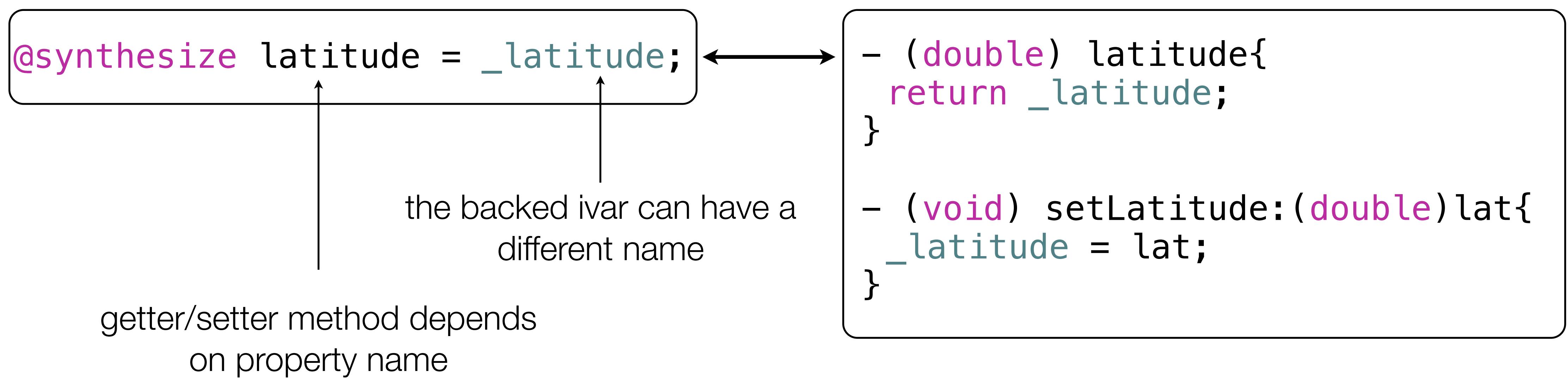
.m

```
NSString *str = poi.detail;
```

usage

Compiler-generated property implementation

- It is possible to let the compiler implement (synthesize) a property matching an instance variable by using the **@synthesize** directive within the implementation of the class
- Using **@synthesize** doesn't forbid to implement either property directly





Dynamic binding

- Objective-C is based on message passing for method invocation
- The resolution of the target of a message is done at runtime: no type checking can be performed at compile-time
- Specifying the type of an object when writing code does not let the compiler perform any type checking, but can only help to highlight possible bugs
- If an object receives a message which it cannot respond to (namely, the class or one of its superclasses does not implement that method) the application will crash
- Casting pointers is a way to “force” the execution of a method, but this should be done carefully (use **introspection**)
- Make sure you always send messages to objects that can respond!**



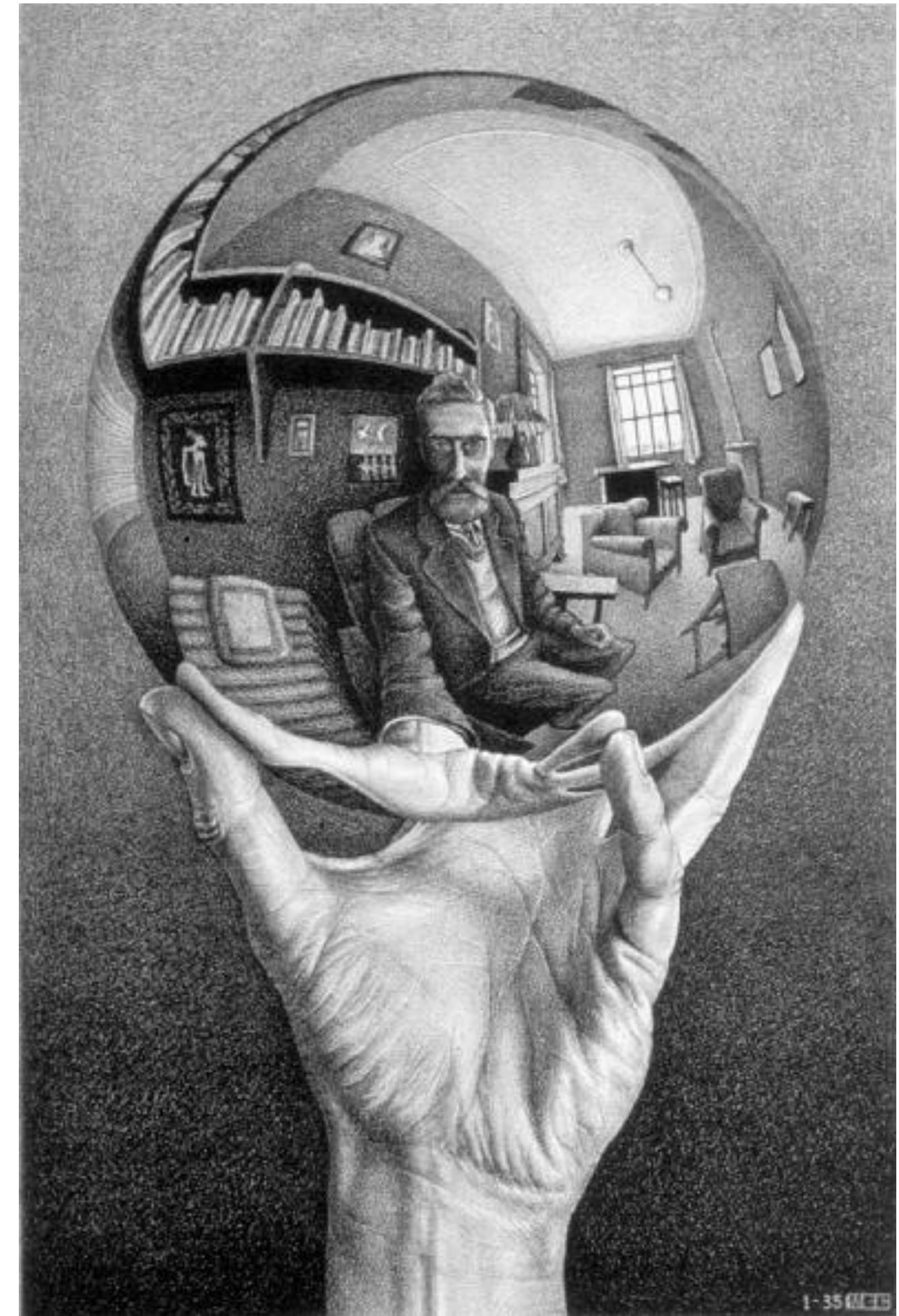
Introspection

- Introspection allows to discover at runtime whether an object can respond or not to a message
- This is especially useful when all we can get are objects of type **id**, such as those that we can get from a collection
- How can we do these test? Any object that is a subclass of NSObject has the following methods:
 - **isKindOfClass:** can be used to check if the object's type matches a class or one of its subclasses
 - **isMemberOfClass:** can be used to check if the object's type matches a class, strictly
 - **respondsToSelector:** can be used to check whether the object can respond to a given selector (a selector is a special type that indicates a method name)

Introspection

- `isKindOfClass:` and `isMemberOfClass:` take a `Class` object as an argument
- A `Class` object can be obtained by using the `class` method which can be invoked on any Objective-C class

```
if( [obj isKindOfClass:[NSString class]]){  
    NSString *str = (NSString *)obj;  
    // do something with string  
    ...  
}
```



Introspection

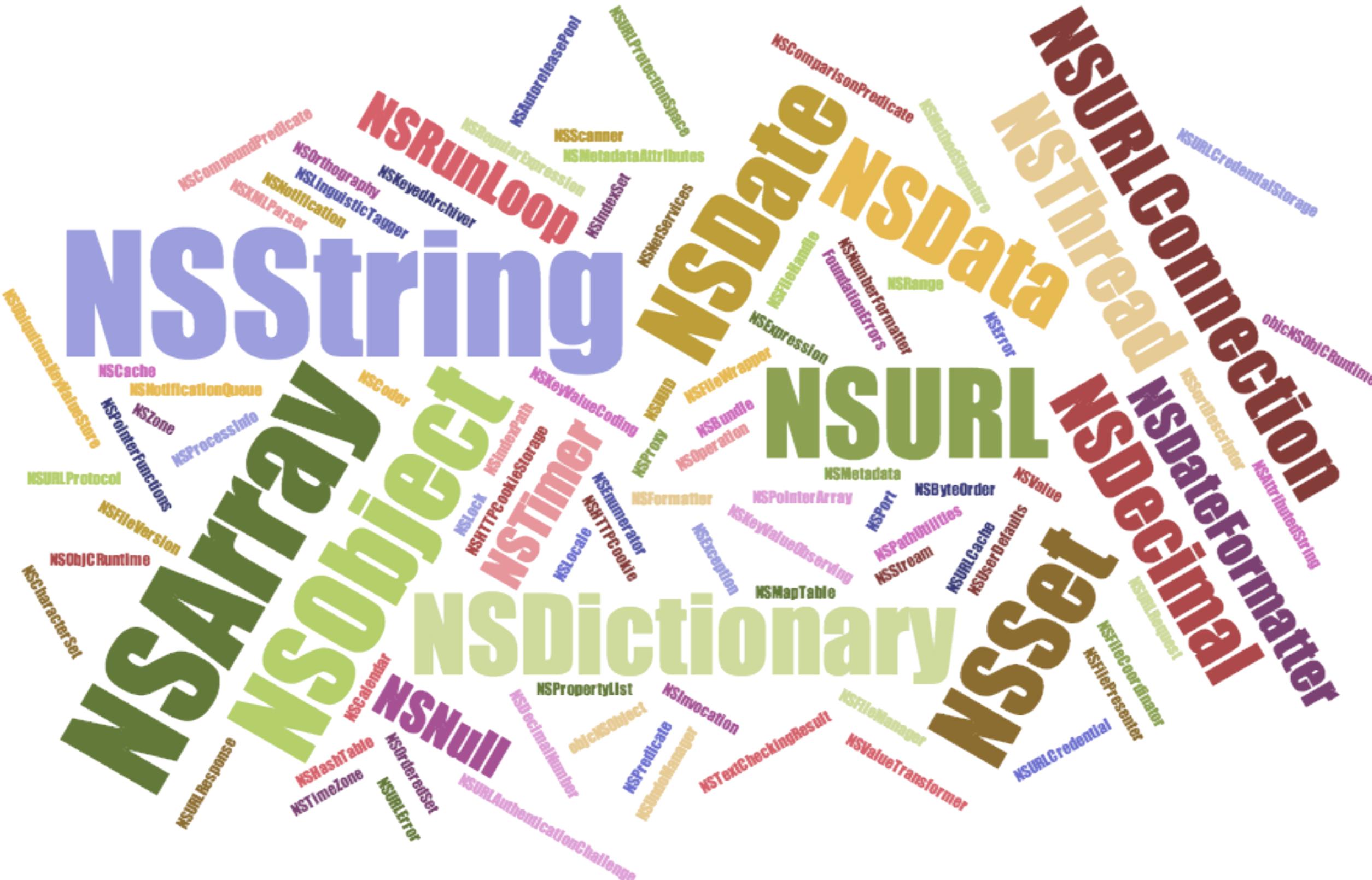
- `respondsToSelector:` takes a **selector** (SEL) as an argument
- A selector can be obtained with the `@selector` keyword:

```
if([obj respondsToSelector:@selector(lowercaseString)]){
    NSString *str = [obj lowercaseString];
    ...
}
```

- It is possible to ask an object to execute a selector:

```
SEL selector = @selector(lowercaseString);
if([obj respondsToSelector:selector]){
    NSString * str = [obj performSelector:selector];
    ...
}
```

Foundation framework





NSObject

- NSObject is the base class for almost anything in iOS (similar to Object class in Java)
- Everything is an (NS)Object
- Provides class and instance methods for memory management
- Provides methods for introspection (`isKindOfClass:` and `isMemberOfClass:` and `respondsToSelector:`)
- Basically, any custom class you will create will be directly or indirectly a subclass of NSObject

NSString & NSMutableString

- **NSString** is a class for managing Unicode strings in Objective-C instead C strings (`char*`)
- Provides useful methods to manipulate strings
- `NSString *str = @"This is an Objective-C string";`
- Note the @ symbol at the beginning: it tells the compiler to handle it as **NSString**
- **NSStrings** are immutable: typically, you get a new **NSString** from an existing one using **NSString** methods
- **NSMutableString** is a mutable **NSString** meaning that it can be manipulated at runtime
- **NSMutableString** is a **NSString**: all methods of **NSString** are available to **NSMutableString**
- With **NSMutableString** you can work with the same object without dealing with new ones



NSNumber

- **NSNumber** wraps primitive types into an object (**int**, **double**, **float**, ...)
- **NSNumber** objects can be used for storing them into collections, which require their elements to be objects

```
NSNumber *number = [NSNumber numberWithInt:2];
int i = [number intValue];
```

- An alternative (faster) method to create instances of **NSNumber**:

```
NSNumber *n = @(10);
```



NSData & NSMutableData

- **NSData** (and its mutable version **NSMutableData**) are used to store bytes
- Typically, **NSData** objects are returned by remote connections since iOS cannot determine in advance what type of data is being transferred (binary or textual)



NSDate

- **NSDate** allows to perform operations on dates:
 - calculations
 - formatting
- **NSTimeInterval (double)** is also used with **NSDate** to manage operations on dates

```
NSDate *now = [NSDate date];
NSDate *nextHour = [NSDate dateWithTimeIntervalSinceNow:3600];
NSTimeInterval delta = [now timeIntervalSinceDate:nextHour];
```



NSArray & NSMutableArray

- NSArray (and its mutable version **NSMutableArray**) provides an ordered list of objects
- Objects of different types can be stored in **NSArray**: introspection should be used to determine the type of each object stored

```
NSArray *array = [NSArray arrayWithObjects:@"string", [NSNumber numberWithInt:10], nil];
int size = [array count];
id first = [array objectAtIndex:0];
id last = [array lastObject];
```

```
NSMutableArray *array2 = [NSMutableArray arrayWithObjects:@"str1", @"str2", nil];
[array2 addObject:@"str3"];
[array2 insertObject:@"first" atIndex:0];
[array2 removeObjectAtIndex:1];
```



NSArray & NSMutableArray

- An alternative (faster) method to create instances of **NSArray**:

```
NSString *obj1 = @""1";
NSString *obj2 = @""2";
NSString *obj3 = @""3";
```

```
...
```

```
NSArray *array = @[obj1,obj2,obj3,@""1",@"(10)];
```

```
NSArray *array = @[@"a",@"b",@"(2)];
NSMutableArray *marray = [NSMutableArray arrayWithArray:array];
[marray removeLastObject];
```



NSDictionary & NSMutableDictionary

- **NSDictionary** (and its mutable version **NSMutableDictionary**) provides a map of key/value pairs
- Both keys and values are objects
- Keys must belong to a class that implements the **hash** and **isEqual:** methods
- Usually, keys are **NSString** objects
- Objects of different types can be stored in **NSDictionary**: introspection should be used to determine the type of each object stored

```
NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:  
                     @"value1", @"key1", @"value2", @"key2", nil];  
int size = [dict count];  
id val = [dict objectForKey:@"key1"];
```

```
NSMutableDictionary *dict2 = [NSMutableDictionary dictionary];  
[dict2 setObject:@"value" forKey:@"key"];  
[dict2 removeObjectForKey:@"key"];
```



NSDictionary & NSMutableDictionary

- An alternative (faster) method to create instances of **NSDictionary**:

```
NSDictionary *dict = @{
    @"key1":@"value1",
    @"key2":@"value2"
};
```



NSSet & NSMutableSet

- NSSet (and its mutable version NSMutableSet) provides an unordered list of objects
- Objects of different types can be stored in NSSet: introspection should be used to determine the type of each object stored

```
NSSet *set = [NSSet setWithObjects:@"obj1", @"obj2", @"obj3", nil];
int size = [set count];
id random = [set anyObject];
```

```
NSMutableSet *teachers = [NSMutableSet set];
[teachers addObject:@"Simone"];
[teachers addObject:@"Marco"];
[teachers removeObject:@"Simone"];
```



Iterating over collections

- It is possible to iterate over a collection as if it were a classical array
- Since collections can store objects of any type, it is a good practice to check against types before sending messages in order to avoid crashes

```
for(int i = 0; i < [alphabet count]; i++){
    id object = [alphabet objectAtIndex:i];
    // do something with object
    ...

    if([object isKindOfClass:[NSString class]]){
        // do something with string
        ...
    }
}
```

Enumeration

- A more convenient way to iterate over a collection is through **enumeration**
- Enumeration is performed through **for-in** statements

```
for(id object in alphabet){  
    if( [object isKindOfClass:[NSString class]]){  
        NSString *string = (NSString *)object;  
        // do something with string  
        ...  
    }  
}
```

- It is possible to avoid explicit casting ONLY IF YOU KNOW WHAT YOU ARE DOING!

```
for(NSString *string in alphabet){  
    // do something with string  
    ...  
}
```



Mobile Application Development



Lecture 1
Introduction to Objective-C



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).



Mobile Application Development



Lecture 2
Introduction to Objective-C (Part II)



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).

Lecture Summary

- Object creation
- Memory management
- Automatic Reference Counting
- Protocols
- Categories





Object creation

- The procedure for creating an object in Objective-C requires two steps:
 1. memory allocation
 2. object instantiation and initialization
- Memory allocation is performed by invoking the `alloc` method of **NSObject**
- Object initialization occurs through an initializer method, which always begins with `init`



Memory allocation

- The **alloc** method asks the operating system to reserve a memory area to keep the object
- **alloc** is inherited by all classes that extend the **NSObject** class
- **alloc** also takes care of setting all instance variables to **nil**



Object initialization

- The **alloc** method returns allocates an object with **nil** instance variables
- Instance variables are properly set using an **initializer** method
- The name of an initializer always begins with **init**
- Every class has a so-called **designated initializer**, which is responsible for setting all the instance variables in such a way that the returned object is in a consistent state
- Any initializer must first invoke the designated initializer of the superclass and check that it did not return a **nil** object
- Initializers can take arguments, typically those that must be passed to properly set instance variables
- It is possible to have more than one initializer, but all other initializers should always call the designated initializer



Structure of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude;
```

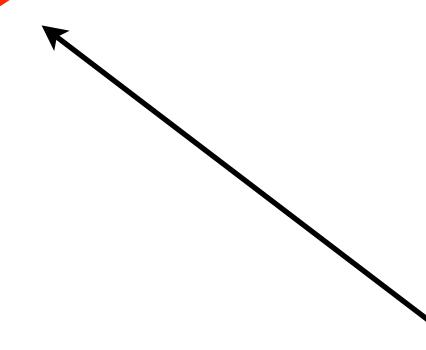
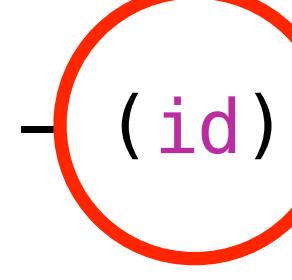
MDPoi.h

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
    if(self = [super init]){
        _name = [name copy];
        _latitude = latitude;
        _longitude = longitude;
    }
    return self;
}
```

MDPoi.m

Declaration of initializers

- `(id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude;`



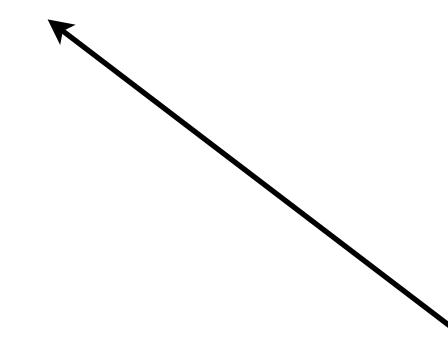
in iOS6 and before, initializers
should always return **id**

in iOS7 a new special keyword
instancetype has been
introduced to explicitly say that the
method returns an object of the
same type of the object it has been
sent the message



Structure of initializers

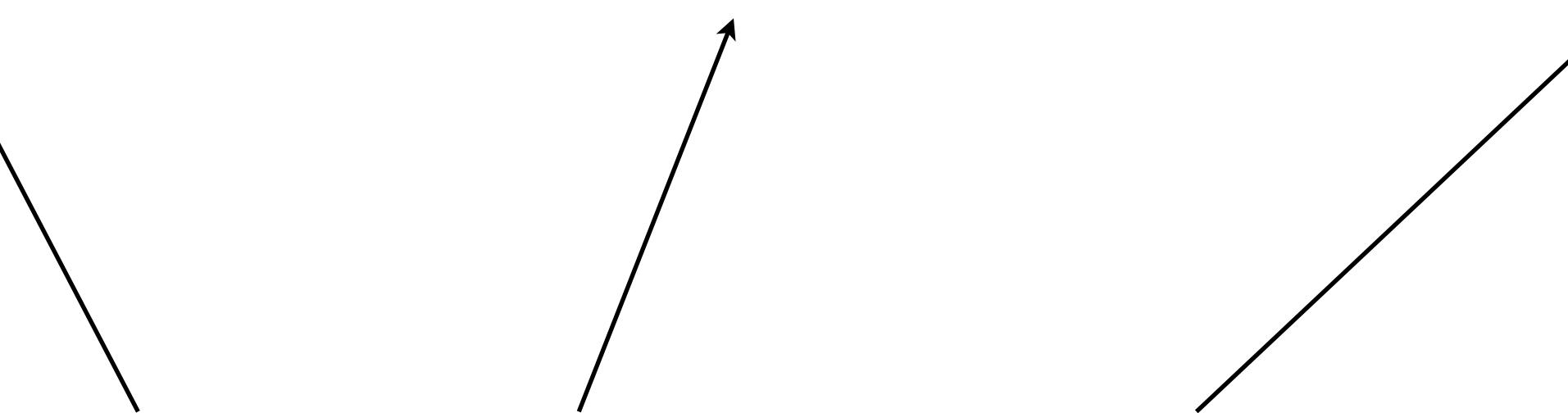
- (**id**) **initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude;**



the name of an initializer starts with
“init” by convention

Structure of initializers

- (**id**) **initWithName:**(**NSString** *)name**latitude:**(**double**)latitude**longitude:**(**double**)longitude;





Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
    if(self = [super init]){
        _name = [name copy];
        _latitude = latitude;
        _longitude = longitude;
    }
    return self;
}
```

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
    if(self = [super init]){
        _name = [name copy];
        _latitude = latitude;
        _longitude = longitude;
    }
    return self;
}
```

call the designated initializer of the
superclass

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
    if(self = [super init]){
        _name = [name copy];
        _latitude = latitude;
        _longitude = longitude;
    }
    return self;
}
```

assign the reference returned by the
designated initializer of the
superclass to **self**



Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
    if(self = [super init]){
        _name = [name copy];
        _latitude = latitude;
        _longitude = longitude;
    }
    return self;
}
```

check if the reference to the returned object is not **nil**

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
    if(self = [super init]){
        _name = [name copy];
        _latitude = latitude;
        _longitude = longitude;
    }
    return self;
}
```

set all the instance variables



Implementation of initializers

```
- (id) initWithFrame:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
    if(self = [super init]){
        _name = [name copy];
        _latitude = latitude;
        _longitude = longitude;
    }
    return self;
}
```

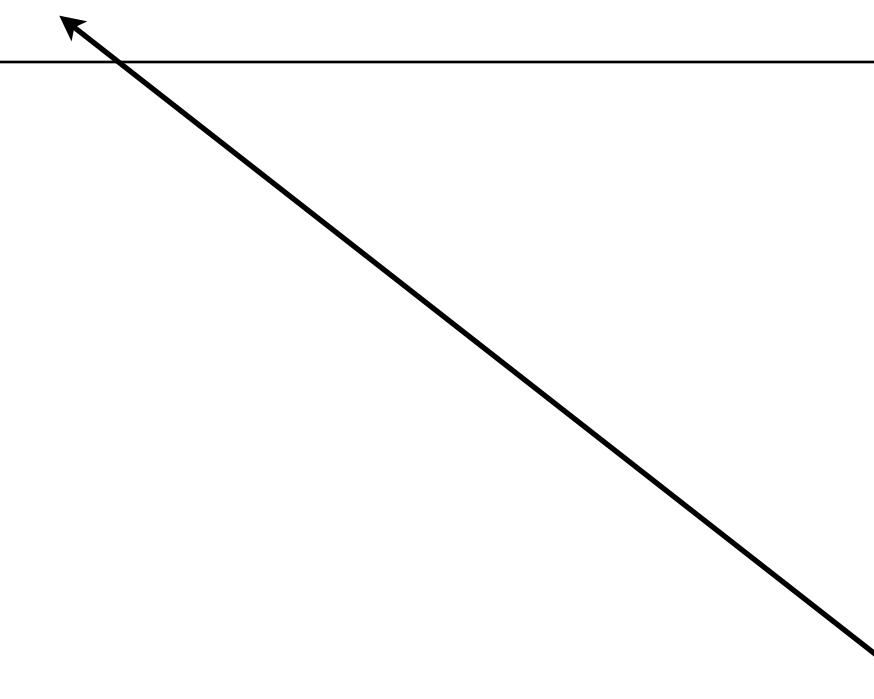
set all the instance variables with properties! **Not a good practice!!!!**

```
- (id) initWithFrame:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
    if(self = [super init]){
        self.name = name;
        self.latitude = latitude;
        self.longitude = longitude;
    }
    return self;
}
```

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
    if(self = [super init]){
        _name = [name copy];
        _latitude = latitude;
        _longitude = longitude;
    }
    return self;
}
```

return a reference to the
initialized object





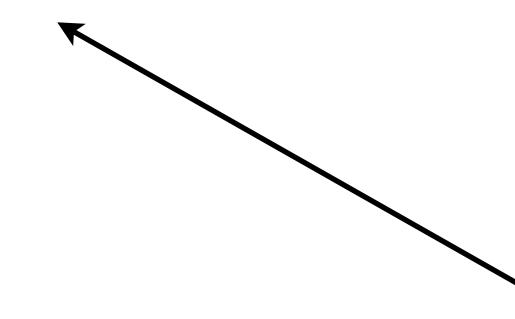
Creating an object

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
```



Creating an object

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
```



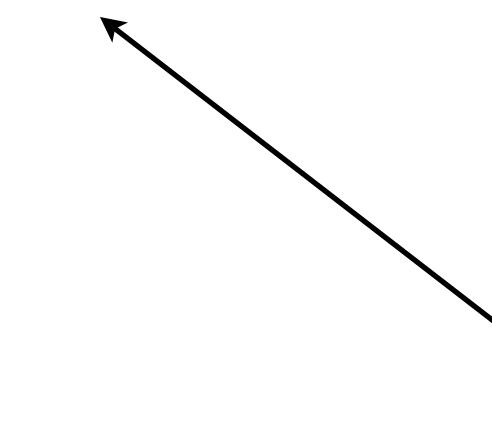
call **alloc**, returns a reference to an instance of **MDPoi**



Creating an object

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
```

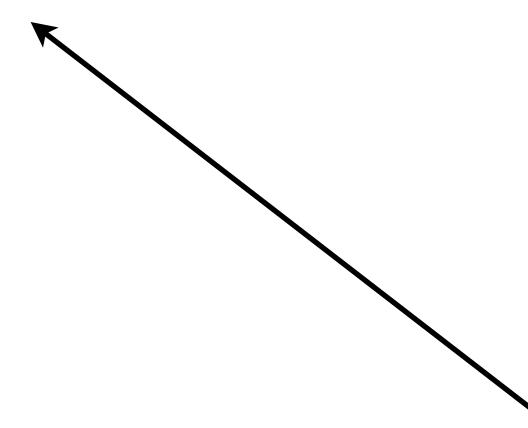
call **initializer** to set
instance variables





Creating an object

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
```



initializer returns a reference to the initialized instance which is assigned to a pointer for later use



Getting an object

- There are many other ways in which you can get an object, other than calling `alloc/init`

```
NSString *str = [words objectAtIndex:0];
```

- **How do we handle memory in this case?**



Memory management: stack and heap

- In Objective-C, all objects are allocated in the heap, so to access them we always use a pointer to the object
- There are two areas in which memory is managed: the stack and the heap
- Roughly speaking, the **stack** is a memory where data are stored with a last-in first-out policy
- The stack is where local (function) variables are stored
- The **heap** is where dynamic memory is allocated (e.g., when we use `new` in C++)
- Anytime an object is instantiated, we are asking to allocate and return a pointer to an area of the heap of the proper size
- But when we do not need the object anymore, we must free the memory so that it can be used again (`free()` in C or `delete` in C++)



The stack

- Suppose we have the code on the right side of this slide
- What does it happen in the stack?

```
- (void) countdown:(int)n{
    int j;
    if([self isEven:n] == YES) j = n/2;
    else j = n + 1;
    for(j; j > 0; j--){
        NSLog(@"%@", j);
    }
}

- (BOOL) isEven:(int)m{
    BOOL even = (m % 2 == 0);
    if(even == YES) return YES;
    else return NO;
}
```



The stack

- Each time a function gets called (or in OOP-terms, a method gets invoked), an activation record gets created and put in the stack, by copying the values and references in the stack as defined by the method's signature

```
- (void) countdown:(int)n{
    int j;
    if([self isEven:n] == YES) j = n/2;
    else j = n + 1;
    for(j; j > 0; j--){
        NSLog(@"%@", j);
    }
}

- (BOOL) isEven:(int)m{
    BOOL even = (m % 2 == 0);
    if(even == YES) return YES;
    else return NO;
}
```

1

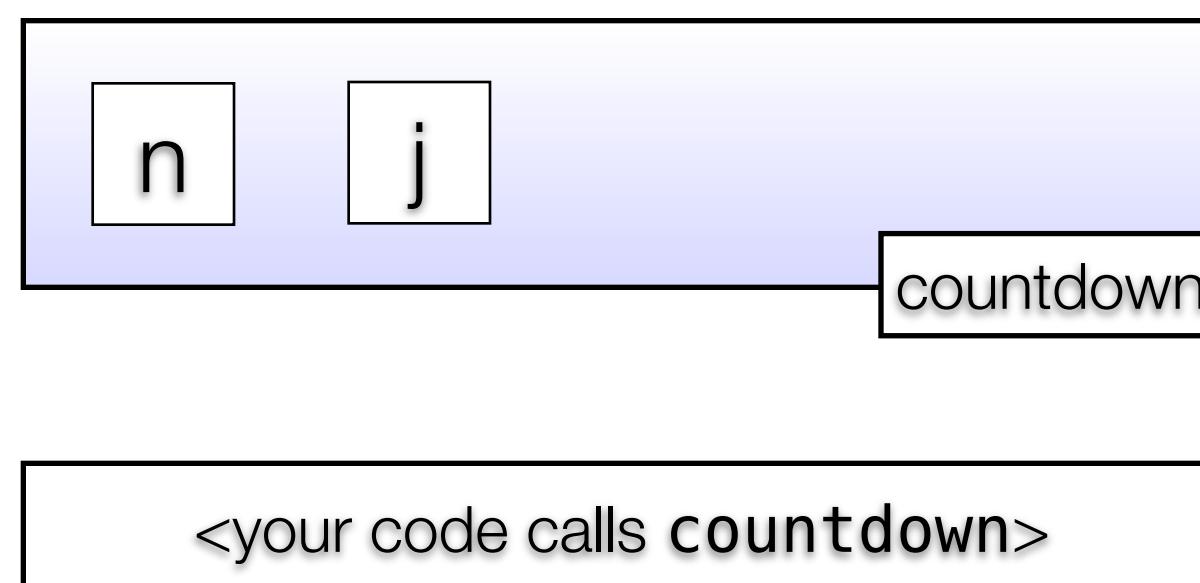
<your code calls **countdown**>

Stack

The stack

- Each time a function gets called (or in OOP-terms, a method gets invoked), an activation record gets created and put in the stack, by copying the values and references in the stack as defined by the method's signature

2



1

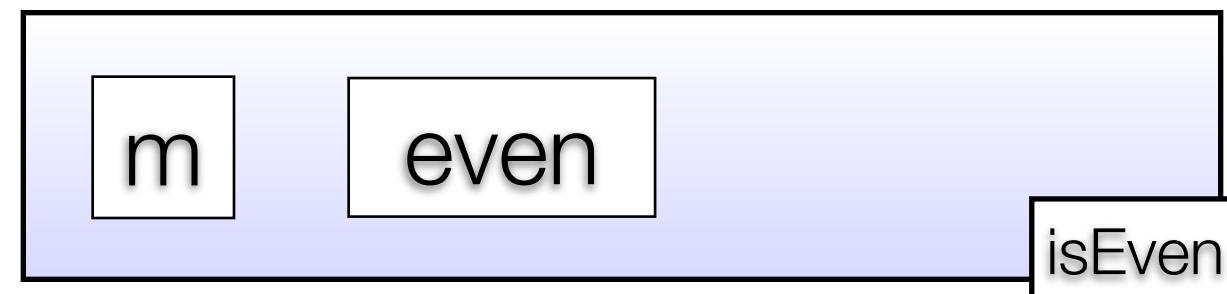
Stack

- ```
(void) countdown:(int)n{
 int j;
 if([self isEven:n] == YES) j = n/2;
 else j = n + 1;
 for(j; j > 0; j--){
 NSLog(@"%@", j);
 }
}
```
- ```
(BOOL) isEven:(int)m{
    BOOL even = (m % 2 == 0);
    if(even == YES) return YES;
    else return NO;
}
```

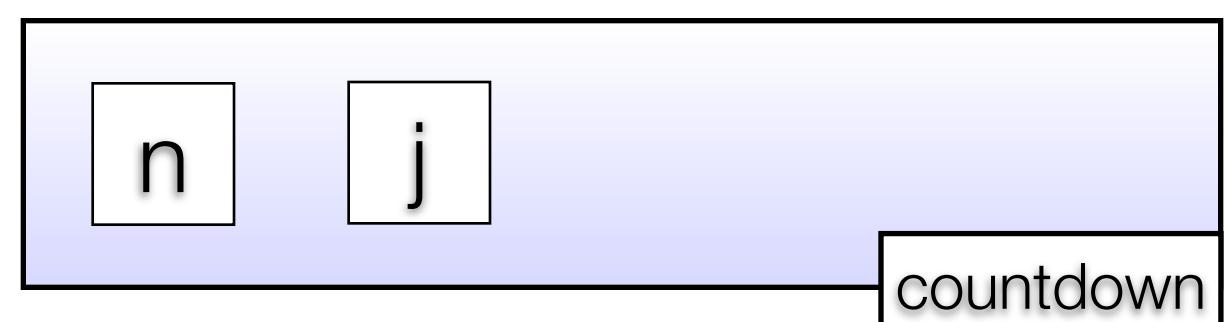
The stack

- Each time a function gets called (or in OOP-terms, a method gets invoked), an activation record gets created and put in the stack, by copying the values and references in the stack as defined by the method's signature

3



2



1



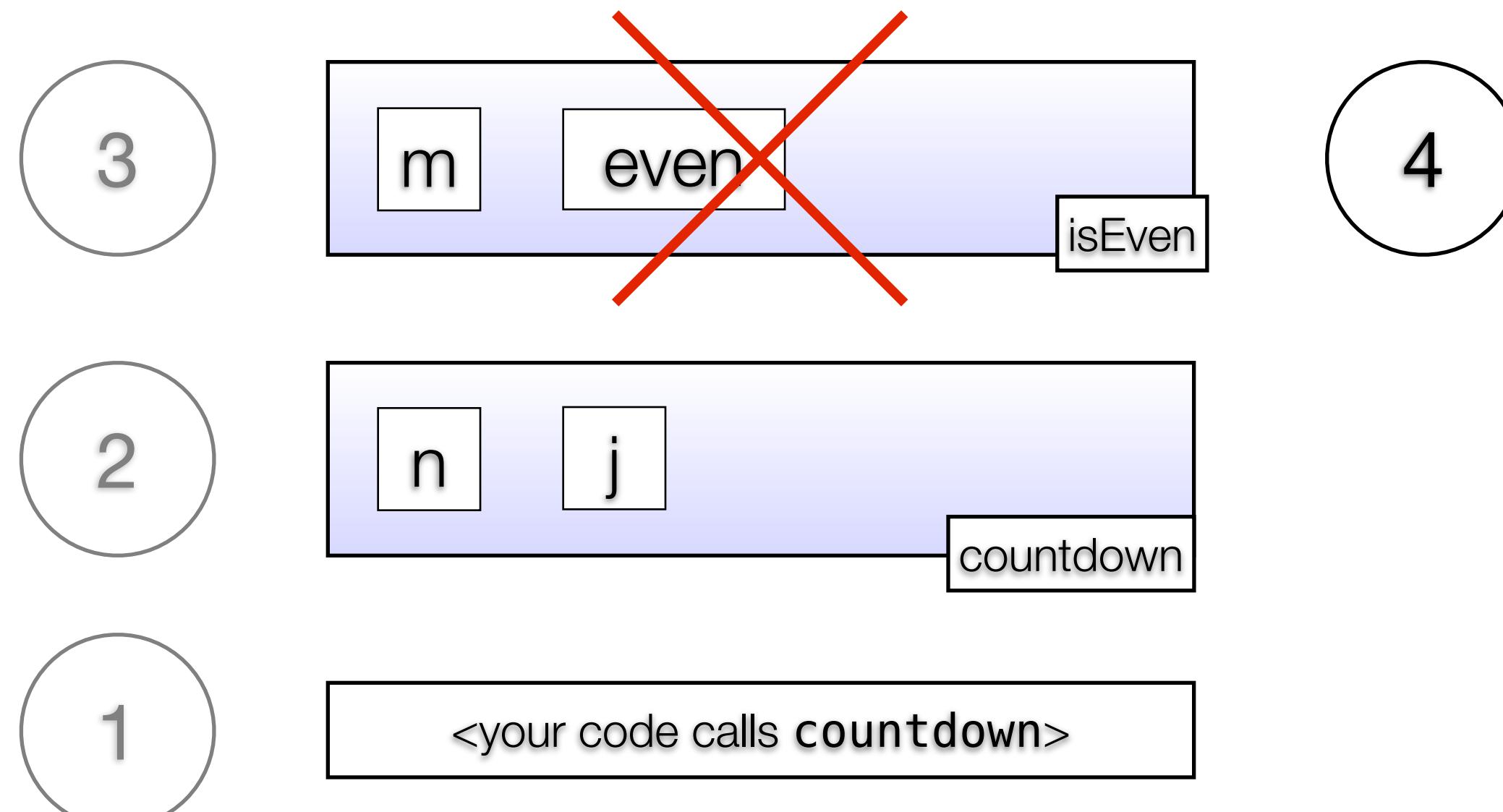
Stack

```
- (void) countdown:(int)n{
    int j;
    if([self isEven:n] == YES) j = n/2;
    else j = n + 1;
    for(j; j > 0; j--){
        NSLog(@"%@", j);
    }
}

- (BOOL) isEven:(int)m{
    BOOL even = (m % 2 == 0);
    if(even == YES) return YES;
    else return NO;
}
```

The stack

- When the method returns, its stack area gets destroyed

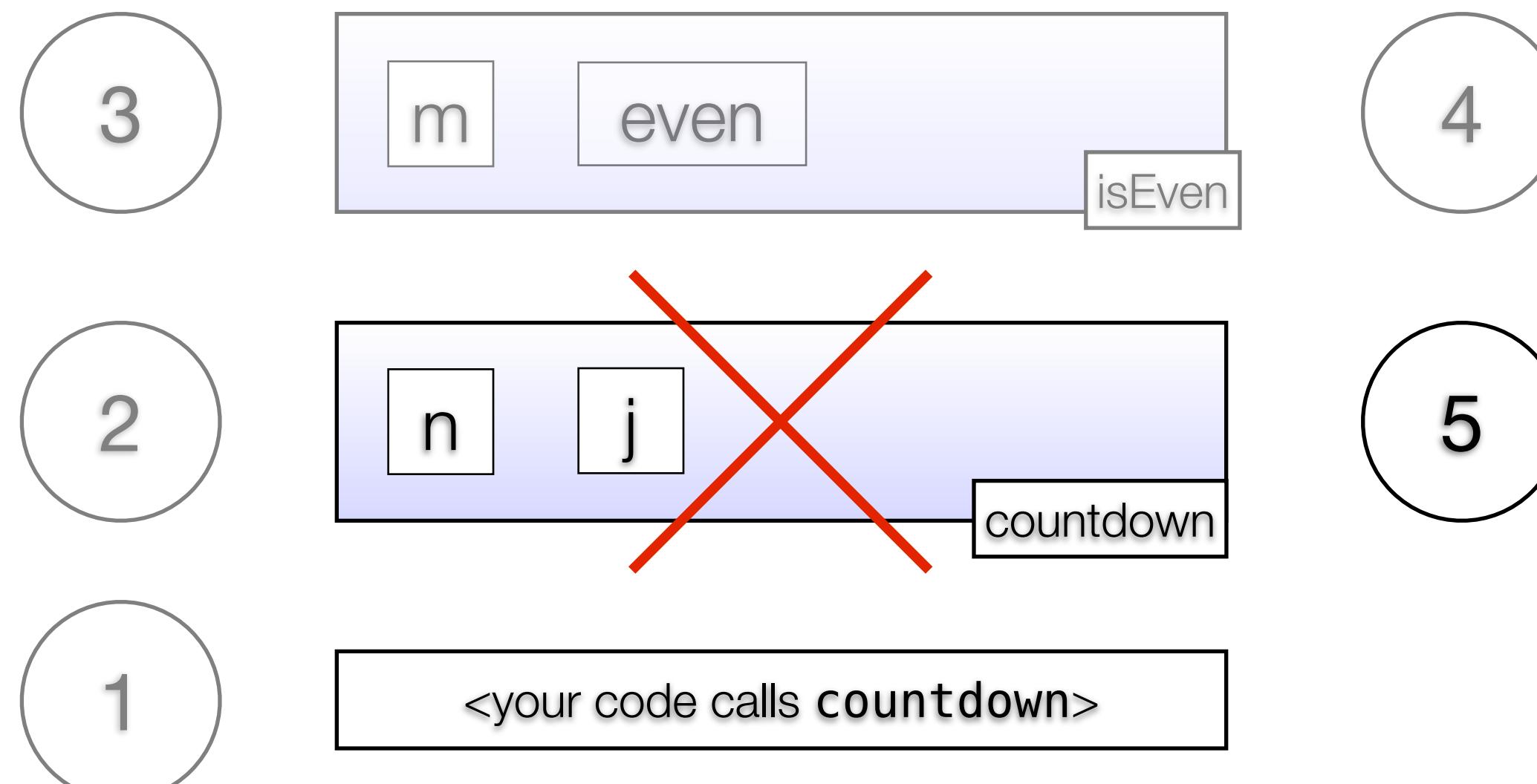


Stack

- `(void) countdown:(int)n{`
 `int j;`
 `if([self isEven:n] == YES) j = n/2;`
 `else j = n + 1;`
 `for(j; j > 0; j--){`
 `NSLog(@"%@", j);`
 `}`
}
- `(BOOL) isEven:(int)m{`
 `BOOL even = (m % 2 == 0);`
 `if(even == YES) return YES;`
 `else return NO;`
}

The stack

- When the method returns, its stack area gets destroyed



Stack

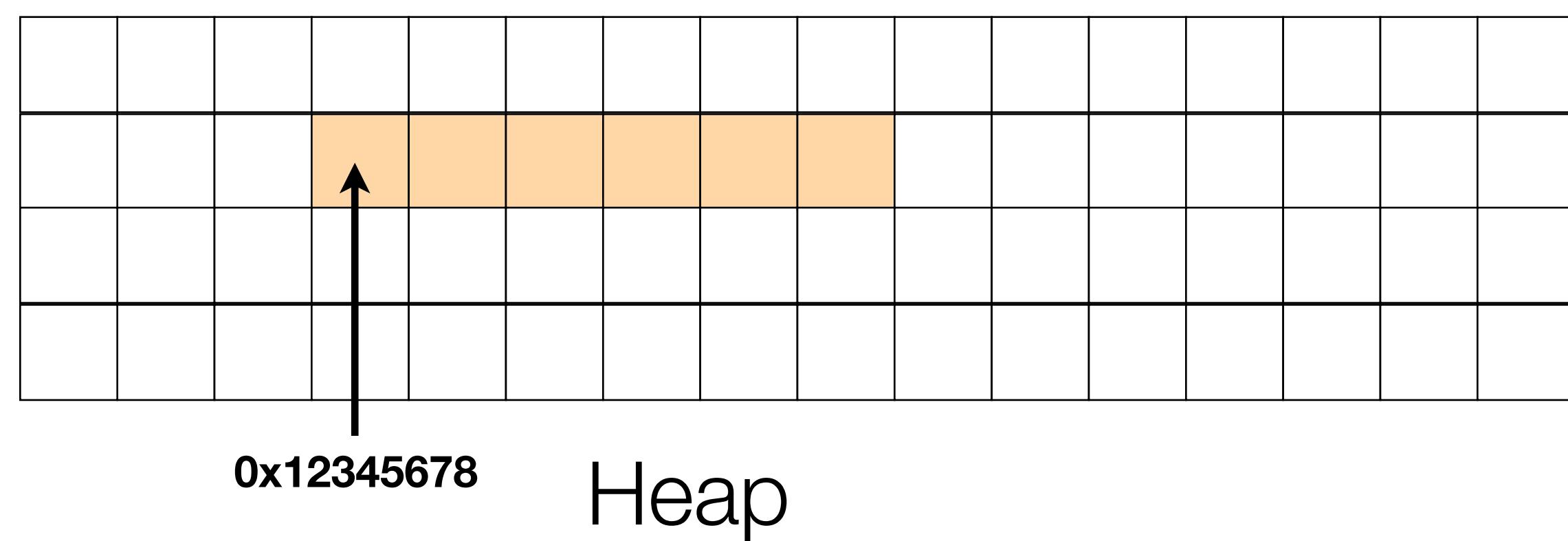
```
- (void) countdown:(int)n{
    int j;
    if([self isEven:n] == YES) j = n/2;
    else j = n + 1;
    for(j; j > 0; j--){
        NSLog(@"%@", j);
    }
}

- (BOOL) isEven:(int)m{
    BOOL even = (m % 2 == 0);
    if(even == YES) return YES;
    else return NO;
}
```

The heap

- Dynamic memory allocation allows to create and destroy objects on-the-fly as you need in your code
- As said before, dynamic memory allocation is performed in the heap, which is an area of memory managed by the operating system to be used as a common resource among processes
- When calling malloc() in C, or new in C++, the operating system will try to allocate the needed amount of space in the heap and return a pointer to the allocated area
- Once you get a pointer to the allocated area, it is your responsibility to use it and keep track of it, so that it can be freed once you are done using it: failing to free allocated memory in the heap results in a memory leak, which eventually might end up consuming all the memory resources of your OS

```
char* cstr = malloc(sizeof(char)*6);
for(int i = 0; i < 5; i++)
    cstr[i] = ('a' + i);
cstr[5] = '\0';
NSLog(@"%@", cstr);
free(cstr);
```



The heap

`malloc` returns the address of an area in the heap of the right size which is assigned to a pointer

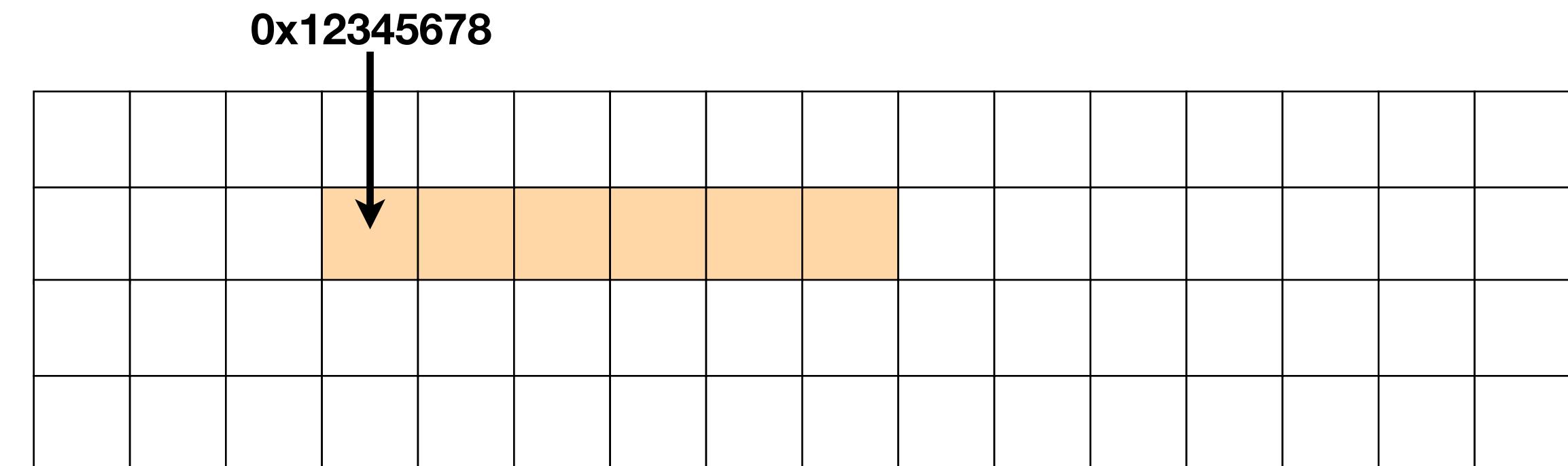
```
char* cstr = malloc(sizeof(char)*6);
```

```
for(int i = 0; i < 5; i++)  
    cstr[i] = ('a' + i);
```

```
cstr[5] = '\0';
```

```
NSLog(@"%@",cstr);
```

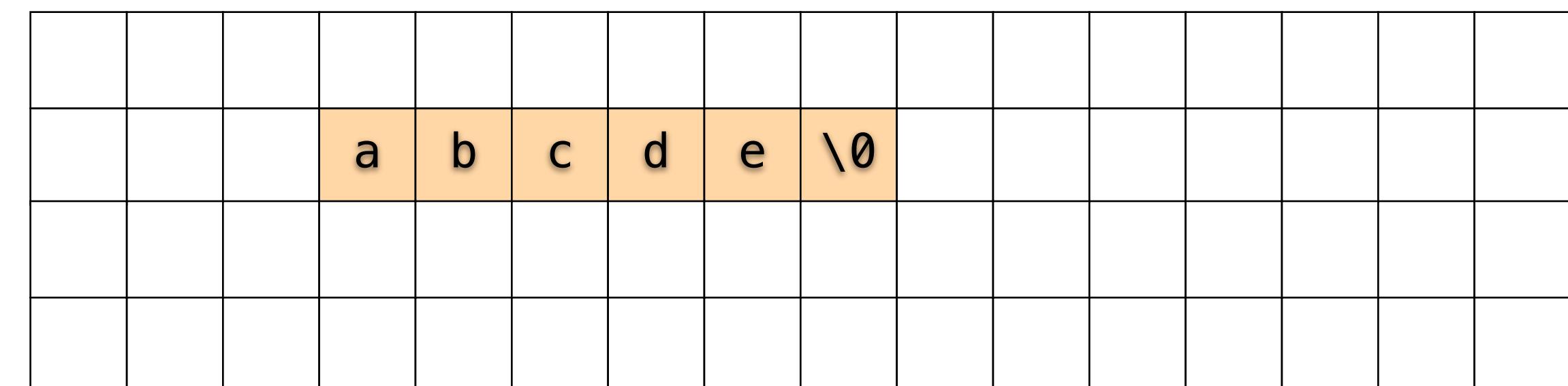
```
free(cstr);
```



Heap

The heap

```
char* cstr = malloc(sizeof(char)*6);  
  
for(int i = 0; i < 5; i++)  
    cstr[i] = ('a' + i);  
  
cstr[6] = '\0';  
  
NSLog(@"%@", cstr);  
  
free(cstr);
```



Heap

it is possible to access
the allocated memory
through the pointer

The heap

```
char* cstr = malloc(sizeof(char)*6);

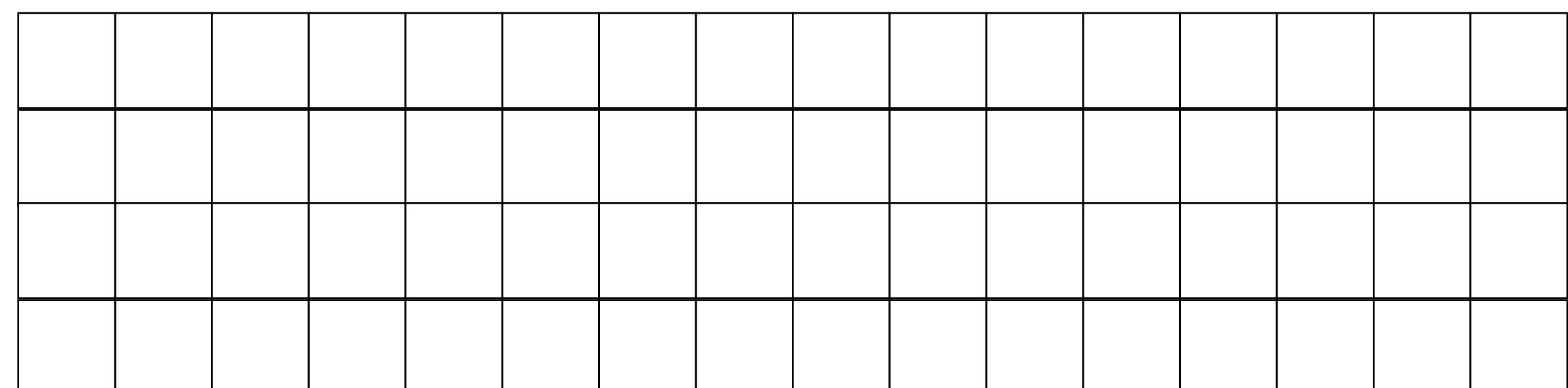
for(int i = 0; i < 5; i++)
    cstr[i] = ('a' + i);

cstr[6] = '\0';

NSLog(@"%@", cstr);

free(cstr);
```

memory is deallocated by calling `free()`



Heap



Memory management

- Originally, Objective-C did not have anything like Java's garbage collector
- The technique used in Objective-C to achieve this is called **reference counting**
- Reference counting means that:
 1. we keep a count of the number of times that we point to an object
 2. when we need to get a reference to the object, we increment the count
 3. when we are done with it, we decrease the count
 4. when the counter goes to 0, the memory is freed (accessing a freed object will crash application): the method **dealloc** (inherited by **NSObject**) gets invoked



Manual Reference Counting

- An object returned by `alloc/init` has a reference count of 1
- `NSObject` defines two methods used to increment and decrement the reference count:
 - `retain`: increase the reference count by 1
 - `release`: decrease the reference count by 1
- We retain an object when we need to use it; the object is retained as long as needed in order to avoid that it will be destroyed while we are using it
- We release the object when we are done using it, so that the reference count can decrease and eventually reach 0 to be freed
- The method `retainCount` (inherited by `NSObject`) can be used to get the current reference count of an object



Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12]; ← retain count = 1
NSLog(@"retain count = %d", [poi retainCount]);

[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);

[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);

[poi release];
NSLog(@"retain count = %d", [poi retainCount]);

[poi release];
NSLog(@"retain count = %d", [poi retainCount]);

[poi release];
```



Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain]; ←
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 2



Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain]; ←
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 3



Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release]; ←
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 2



Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release]; ←
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 1



Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release]; ←
```

retain count = 0



Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 0

object poi is marked for deallocation!



Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 0

at the end of the event loop, object **poi** gets deallocated!



Object ownership

- The memory management model is based on **object ownership**
- Any object may have one or more owners
- As long as an object has at least one owner, it continues to exist
- Rules for object ownership:
 1. You own any object you create (using `alloc/init`)
 2. You can take ownership of an object using `retain` (since the original owner is who created it)
 3. When you no longer need it, you must relinquish ownership of an object you own using `release`
 4. You must not relinquish ownership of an object you do not own
- The number of `alloc/init` and `retain` must always match that of `release`

Temporary object ownership

- There are times when sending a release message might create a premature deallocation of an object
 - ```
(NSArray *)getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 [array release];
 return array;
}
```

# Temporary object ownership

- There are times when sending a release message might create a premature deallocation of an object

```
- (NSArray *)getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 [array release];
 return array;
}
```

`release` brings reference count to 0 and gets deallocated, so the returned object does not exist anymore

# Temporary object ownership

- There are times when sending a release message might create a premature deallocation of an object
  - ```
(NSArray *)getAllPois{
    NSMutableArray *array = [[NSMutableArray alloc] init];
    ...
    [array release];
    return array;
}
```
- To avoid this situation, we must instruct the object to be released later on: use **autorelease**
 - ```
(NSArray *)getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 [array autorelease];
 return array;
}
```

# Temporary object ownership

- There are times when sending a release message might create a premature deallocation of an object

```
- (NSArray *)getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 [array release];
 return array;
}
```

- To avoid this situation, we must instruct the object to be released later on: use **autorelease**

```
- (NSArray *)getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 [array autorelease];
 return array;
}
```

**autorelease** sends a deferred **release** message, so the object gets deallocated after it is returned

# Temporary object ownership

- It is possible to avoid the use of `alloc/init` and `autorelease`

```
- (NSArray *)getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 [array autorelease];
 return array;
}
```

- We can get an `autoreleased` object so we don't need to worry about (auto)releasing it

```
- (NSArray *)getAllPois{
 NSMutableArray *array = [NSMutableArray array];
 ...
 return array;
}
```

the `array` method returns an autorelease object, which can be returned

# Temporary object ownership

- Many classes provide both
  - initializers to be used to get an object you own
  - methods that return an **autorelease**d object, which you do not own
- Common examples of methods returning **autorelease**d objects:

```
NSString *str = [NSString stringWithFormat:@"item %d", i];

NSArray *array = [NSArray arrayWithObjects:@"one", @"two", @"three", nil];

NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
 @{@"key1": @"value1",
 @"key2": @"value2"}, nil];
```



# Deallocating memory

- When the reference count of an object reaches 0, its `dealloc` method is invoked
- `dealloc` is inherited from `NSObject`, so if you do not implement it the default method is used
- `dealloc` must never be called directly, as it is automatically called when the reference count reaches 0
- `dealloc` must be overridden if your class has instance variables of a non-primitive type
- The last instruction of `dealloc` must be a call to the superclass implementation of `dealloc` (only time you are authorized to call `dealloc` directly)

```
- (void) dealloc{
 [_name release];
 [super dealloc];
}
```

# Memory management with properties

- What about ownership of objects returned by getters or passed in to setters?
- Typically, getters return an object that is going to live long enough to be retained (similarly to an autoreleased object)
- Setters should retain the object being set (or we would have an object that might be deallocated)
- The problem is: what happens when properties are **@synthesized**?
- There are three options to tell how the setters should be implemented: **retain**, **copy**, and **assign**

```
@property (retain) NSString *name;
@property (copy) NSString *name;
@property (assign) NSString *name;
```



# Properties: retain

```
@property (retain) NSString *name;
```

**retain** means that the instance variable backing the property is retained

```
@synthesize name;
```



```
- (void) setName:(NSString *)name{
 [_name release];
 _name = [name retain];
}
```

# Properties: retain

```
@property (retain) NSString *name;
```

```
@synthesize name;
```



```
- (void) setName:(NSString *)name{
 [_name release];
 _name = [name retain];
}
```



first, the current object is sent a **release** message, since it is no longer used (ok if it is **nil!**)

# Properties: retain

```
@property (retain) NSString *name;
```

```
@synthesize name;
```



```
- (void) setName:(NSString *)name{
 [_name release];
 _name = [name retain];
}
```

2

next, the new object is sent a **retain** message, since we are going to use it



# Properties: copy

```
@property (copy) NSString *name;
```

copy means that the instance variable backing the property is copied

```
@synthesize name;
```



```
- (void) setName:(NSString *)name{
 [_name release];
 _name = [name copy];
}
```

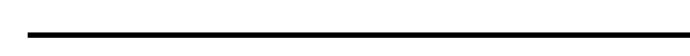


# Properties: assign

```
@property (assign) NSString *name;
```

assign means that the instance variable backing the property is not retained

```
@synthesize name;
```



```
- (void) setName:(NSString *)name{
 _name = name;
}
```



# Good practice

## Do not use accessor methods in initializers!

- Using accessor methods (both by sending a message or by using the property, which are equivalent) is bad because accessor methods might rely on some state of the object, while in the initializer we have no guarantee about the state since it is being initialized

```
- (id) initWithFrame:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
 if(self = [super init]){
 self.name = name;
 self.latitude = latitude;
 self.longitude = longitude;
 }
 return self;
}
```



# Good practice

## Do not use accessor methods in initializers!

- Using accessor methods (both by sending a message or by using the property, which are equivalent) is bad because accessor methods might rely on some state of the object, while in the initializer we have no guarantee about the state since it is being initialized

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
if(self = [super init]){
 self.name = name;
 self.latitude = latitude;
 self.longitude = longitude;
}
return self;
}
```

# Good practice

**Do not use accessor methods in initializers!**



- Using accessor methods (both by sending a message or by using the property, which are equivalent) is bad because accessor methods might rely on some state of the object, while in the initializer we have no guarantee about the state since it is being initialized

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
 if(self = [super init]){
 _name = [name copy];
 _latitude = latitude;
 _longitude = longitude;
 }
 return self;
}
```

# Good practice

**Do not use accessor methods in dealloc!**



- If we are in `dealloc`, it means that the object has a reference count of 0 and that it is being deallocated, so it is neither safe nor appropriate to send messages

```
- (void) dealloc{
 [_name release];
 [super dealloc];
}
```

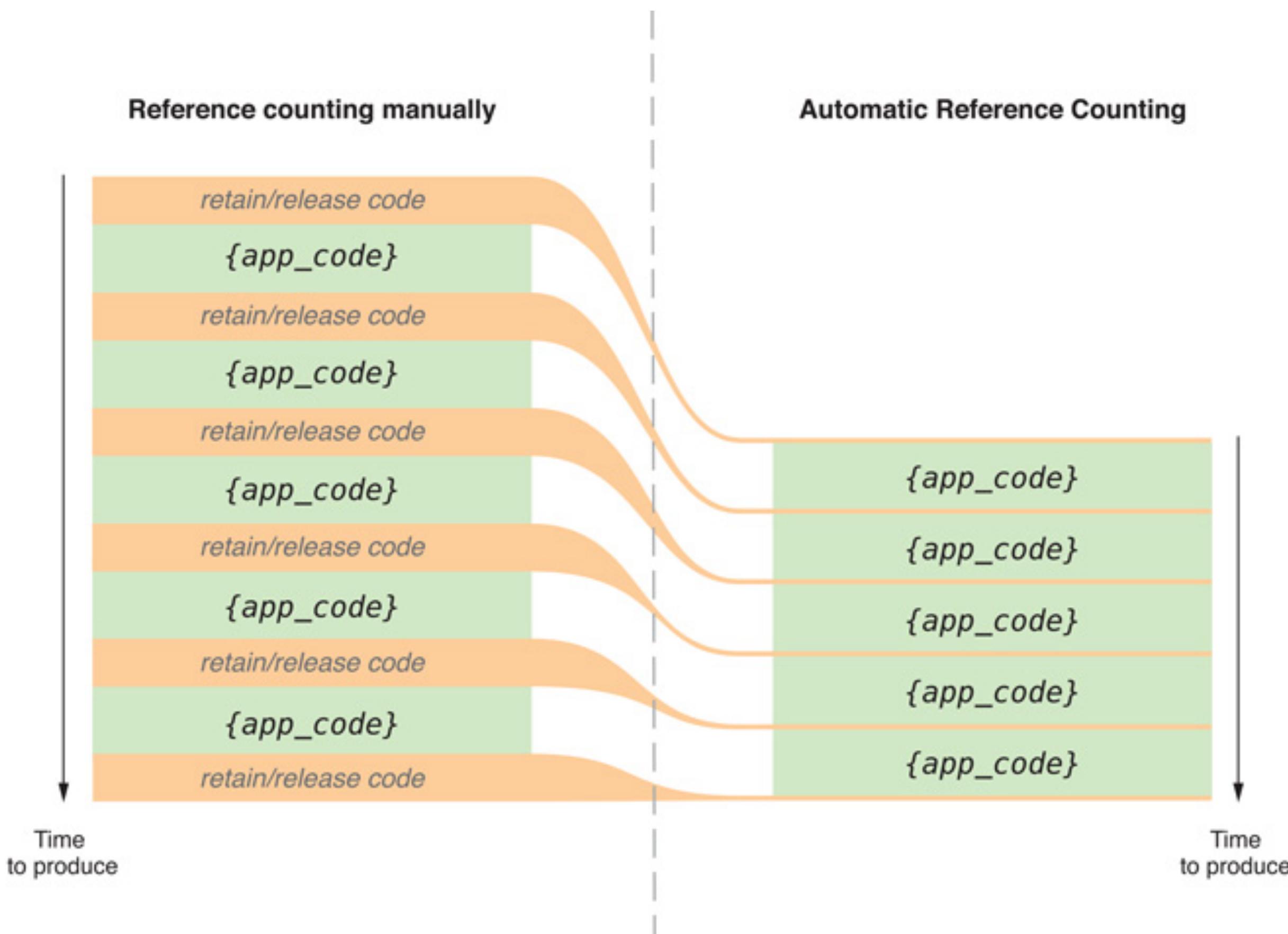


# Automatic Reference Counting

- Since iOS 4, a new feature has been introduced to simplify life of Objective-C programmers for managing memory allocation: **Automatic Reference Counting (ARC)**
- ARC delegates the responsibility to perform reference counting from the programmer to the compiler
- `retain`, `release`, and `autorelease` must no longer be explicitly invoked
- ARC evaluates the lifetime requirements of objects and automatically inserts appropriate memory management calls for you at compile time
- The compiler also generates appropriate `dealloc` methods; implement a custom `dealloc` if you need to manage resources other than releasing instance variables, but in any case you must not call `[super dealloc`], since ARC does it automatically
- Corollary: with ARC, the use of `retain`, `release`, `autorelease`, `retainCount`, and `dealloc` is forbidden
- ARC is NOT Garbage Collection, since no process is being executed to cleanup memory

# Automatic Reference Counting

- How does code get updated with ARC?





# Automatic Reference Counting

- How does code get updated with ARC?

## MRC

```
- (NSArray *) getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 [array autorelease];
 return array;
}
```

# Automatic Reference Counting

- How does code get updated with ARC?

## MRC

```
- (NSArray *) getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 [array autorelease];
 return array;
}
```

**autorelease** is used to defer the release  
of the object

# Automatic Reference Counting

- How does code get updated with ARC?

## MRC

```
- (NSArray *) getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 [array autorelease];
 return array;
}
```

## ARC

```
- (NSArray *) getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 ...
 return array;
}
```

**autorelease** is used to defer the release  
of the object

# Automatic Reference Counting

- How does code get updated with ARC?

## MRC

```
- (NSArray *) getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 [array autorelease];
 return array;
}
```

## ARC

```
- (NSArray *) getAllPois{
 NSMutableArray *array = [[NSMutableArray alloc] init];
 ...
 ...
 return array;
}
```

**autorelease** is used to defer the release of the object

the compiler generates the appropriate call for releasing the object



# Properties with ARC

- What about the ownership of objects returned by accessor methods?
- The implementation of the accessor methods depends on the declaration of the properties
- ARC introduces the concept of **weak references**
- A weak reference does not extend the lifetime of the object it points to, and automatically becomes **nil** when there are no strong references to the object
- The keywords **weak** and **strong** are introduced to manage weak and strong references for objects

# Properties with ARC: strong

- **strong** properties are equivalent to **retain** properties

```
@property (strong) MyClass *myObject;
```

**ARC**



```
@property (retain) MyClass *myObject;
```

**MRC**

# Properties with ARC: **weak**

- **weak** properties are similar to **assign** properties
- If the **MyClass** instance is deallocated, the property value is set to nil instead of remaining as a dangling pointer

```
@property (weak) MyClass *myObject;
```

**ARC**



```
@property (assign) MyClass *myObject;
```

**MRC**



# Protocols

- Objective-C provides a way to define a set of methods similarly to Java interfaces: **protocols**
- Interfaces (.h files) are used to declare the methods and properties associated with a class
- Protocols declare the methods expected to be used for a particular situation
- Sometimes it is not important who is going to perform an action, we just need to know that someone is going to do it (e.g., table view data source)
- **Protocols define messaging contracts**
- A protocol is used to declare methods and properties that are independent of any specific class
- The usage of protocols allows to maximize the reusability of code by minimizing the interdependence among parts of your code



# Defining a protocol

- Protocols are defined in the interface file, between the `@protocol` directive and the corresponding `@end`

```
@protocol MyProtocol
```

```
// definition of methods and properties
- (void)method1;
- (NSString *)method2;
- (NSArray *)method3:(NSString *)str;
```

```
@end
```



# Defining a protocol

- `@protocol` methods can be declared as `@required` (default) or `@optional`

```
@protocol MyProtocol
```

- `(void)method1;`
- `(NSString *)method2;`

```
@optional
```

- `(void)optionalMethod;`

```
@required
```

- `(NSArray *)method3:(NSString *)str;`

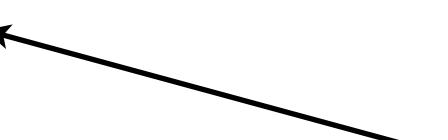
```
@end
```

# Defining a protocol

- `@protocol` methods can be declared as `@required` (default) or `@optional`

```
@protocol MyProtocol
```

```
- (void)method1;
- (NSString *)method2;
```



Required methods

```
@optional
- (void)optionalMethod;
```

```
@required
- (NSArray *)method3:(NSString *)str;
```

```
@end
```

# Defining a protocol

- `@protocol` methods can be declared as `@required` (default) or `@optional`

```
@protocol MyProtocol
```

```
- (void)method1;
- (NSString *)method2;
```

```
@optional
- (void)optionalMethod;
```

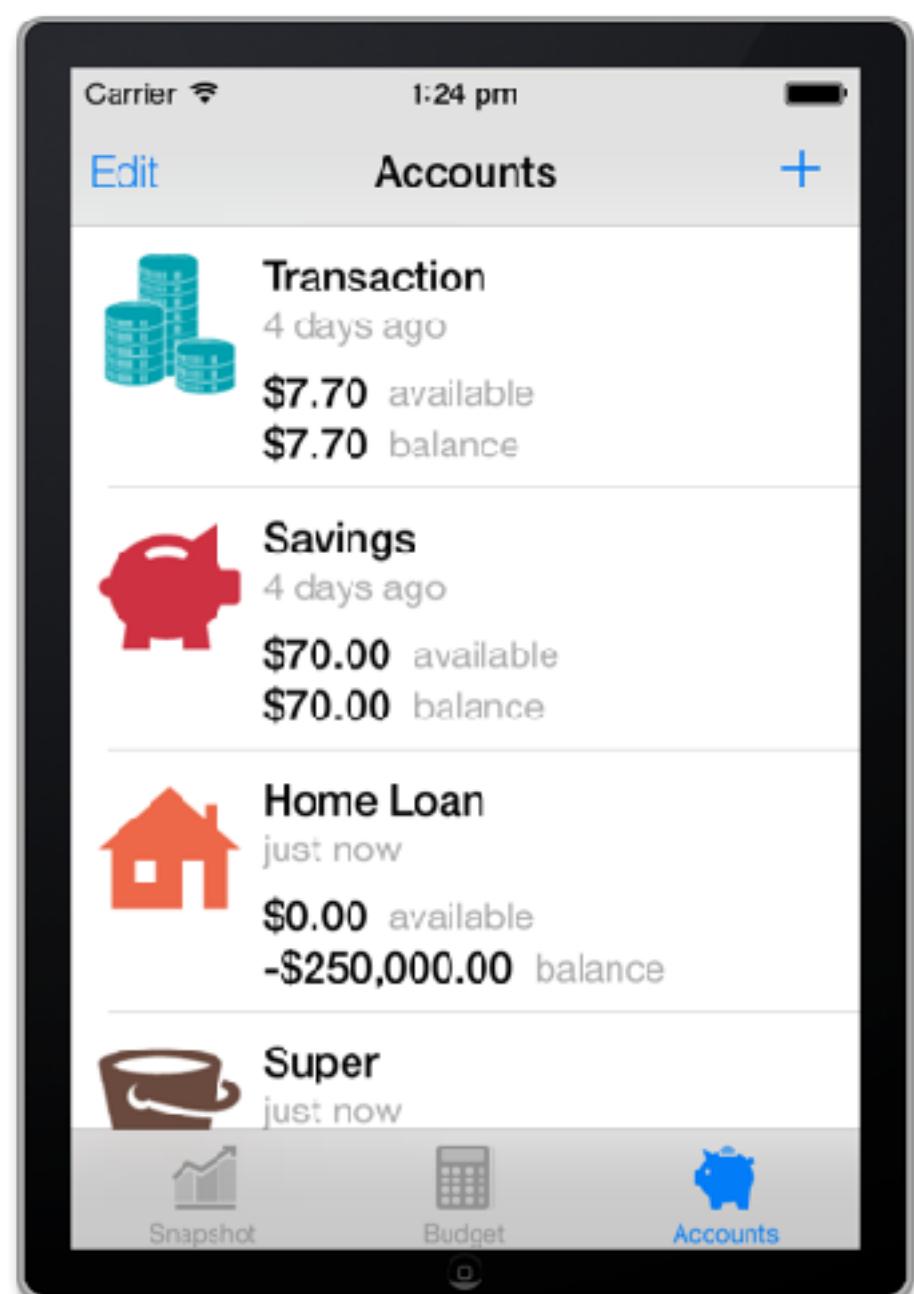
**Optional methods**

```
@required
- (NSArray *)method3:(NSString *)str;
```

```
@end
```

# Conforming to protocols

- Classes that implement the required methods defined in a protocol are said to **conform to the protocol**
- Or the other way: if a class is willing to conform to a protocol, it must implement all the required methods defined in the protocol definition
- By conforming to a protocol, a class is allowed to be used to provide concrete behavior to parts of the program that depend on such behavior but do not depend on a specific implementation of that behavior
- For example:
  - **UITableViewDataSource** is a protocol that defines methods that a **UITableView** depends on
  - **UITableView** implementation (how it is displayed, ...) is independent however from the data it has to display
  - **UITableView** uses a **UITableViewDataSource** to fetch the data it will display
  - a class that conforms to **UITableViewDataSource** will then be passed to the **UITableView** at runtime





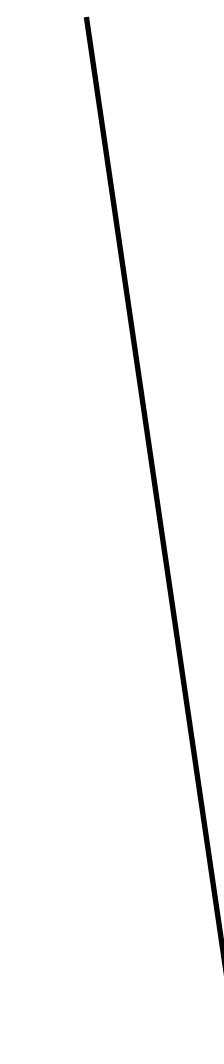
# Defining a class that conforms to a protocol

- In order for a class to conform to a protocol, it must:
  1. declare it between angle brackets

```
@interface MyClass : NSObject<MyProtocol>{
 ...
}
@end
```

# Defining a class that conforms to a protocol

- In order for a class to conform to a protocol, it must:
  1. declare it between angle brackets



```
@interface MyClass : NSObject<MyProtocol>{
 ...
}
@end
```



# Defining a class that conforms to a protocol

- In order for a class to conform to a protocol, it must:
  1. declare it between angle brackets
  2. implement the required protocol methods

```
@protocol MyProtocol
- (void)method1;
- (NSString *)method2;

@optional
- (void)optionalMethod;

@required
- (NSArray *)method3:(NSString *)str;

@end
```

```
@implementation MyClass
- (void)method1{
 ...
}

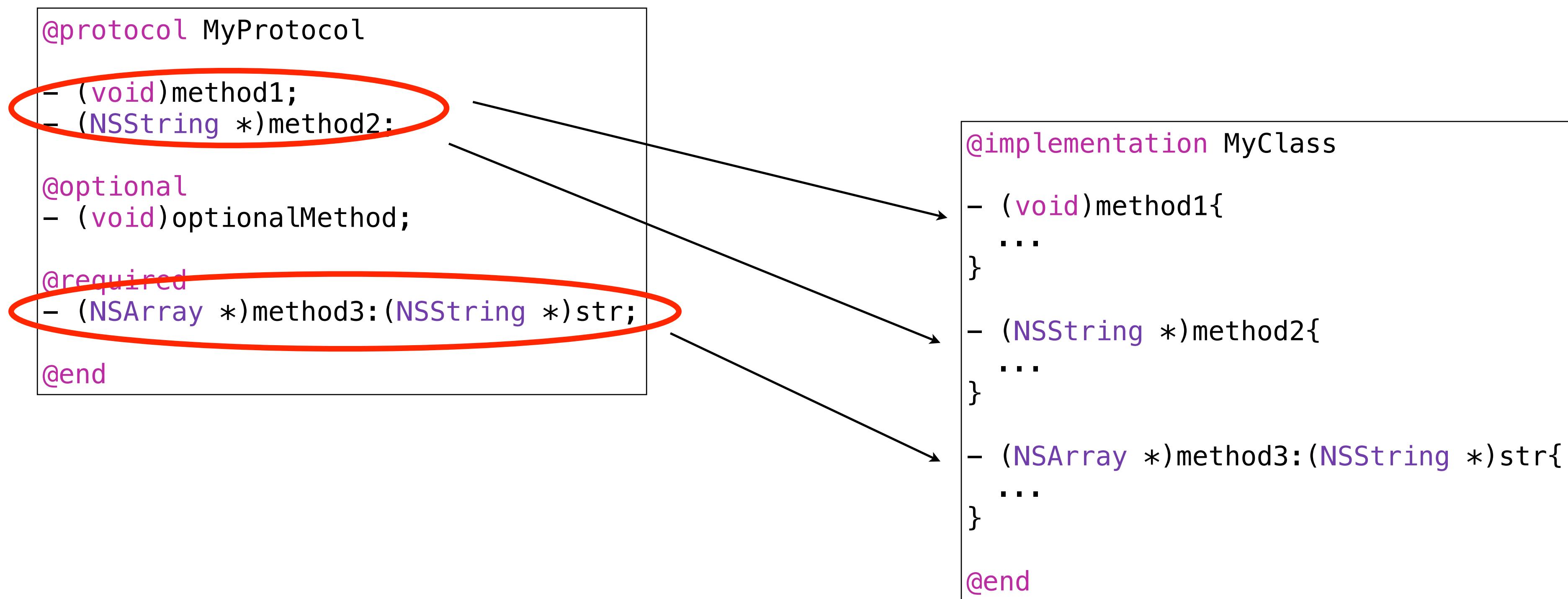
- (NSString *)method2{
 ...
}

- (NSArray *)method3:(NSString *)str{
 ...
}

@end
```

# Defining a class that conforms to a protocol

- In order for a class to conform to a protocol, it must:
  1. declare it between angle brackets
  2. implement the required protocol methods





# Protocols

- Since some methods may have been declared as optional, it is important to check at runtime whether the target object has implemented a method before sending a message to it

```
if ([target respondsToSelector:@selector(optionalMethod)]) {
 ...
 [target optionalMethod];
 ...
}
```



# Categories

- Sometimes, you may wish to extend an existing class by adding behavior that is useful only in certain situations
- Subclassing in those situations might be a “hard” approach
- Objective-C provides a solution to manage these situations in a clean and lightweight way:  
**categories**
- Categories add methods to existing classes, without subclassing
- If you need to add a method to an existing class, perhaps to add functionality to make it easier to do something in your own application, the easiest way is to use a category



# Defining a category that enhances a class

- In order to define a category for a class, you must:
  1. “re-define” the class with the **@interface** directive
  2. specify the name of the category between parentheses
  3. declare the new methods
  4. provide an implementation for the category in the **.m** file using the **@implementation** directive
- There is a naming convention for the name of the files: **<BaseClass>+<Category>.h** and **<BaseClass>+<Category>.m**



# Example: a category that enhances NSString

```
@interface NSString (MyCategory)
- (int) countOccurrences:(char)c;
@end
```

NSString+MyCategory.h

```
@implementation NSString (MyCategory)
- (int) countOccurrences:(char)c{
 ...
}
@end
```

NSString+MyCategory.m

```
#import "NSString+MyCategory.h"

...
int occurA = [self.name countOccurrences:'a'];
...
```

usage



# A trick with categories: private methods and properties

- It is nice to have utility methods that you use in your class “privately” and that you do not want to expose
- We have seen that the @private modifier can be used in conjunction with attributes in order to restrict the visibility of the variables
- Unfortunately, this does not apply with methods
- A little workaround to get private methods in your class is to define an anonymous category in your implementation file and declare your private methods there
- Since the declaration is inside the implementation file, it will be hidden to other classes



# A trick with categories: private methods and properties

```
@interface MDPoi ()

@property (strong) NSString *name;
- (void) privateMethod;

@end

@implementation MDPoi

@synthesize name = _name;

- (void) privateMethod{
 ...
}

@end
```

MDPoi.m



# Mobile Application Development



Lecture 2  
Introduction to Objective-C (Part II)



This work is licensed under a [Creative Commons Attribution-  
NonCommercial-ShareAlike 4.0 International License](#).



# Mobile Application Development



Lecture 3  
iOS SDK



This work is licensed under a [Creative Commons Attribution-  
NonCommercial-ShareAlike 4.0 International License](#).

# Lecture Summary

- iOS operating system
- iOS SDK
- Tools of the trade
- Model-View-Controller
- MVC interaction patterns
- View Controllers





# iOS

- iOS is Apple's mobile operating system, shipping with iPhone, iPod Touch, and (old) iPad devices
- First released in 2007
- Current version is iOS 14
  - released on September 2020
- This class covers iOS7
  - released on September 2014
  - runs on iPhone 4s and later, iPad 2 and later, iPod Touch 5th gen and later, iPad Mini and later
- iOS apps run in a UNIX-based system and have full support for threads, sockets, etc...

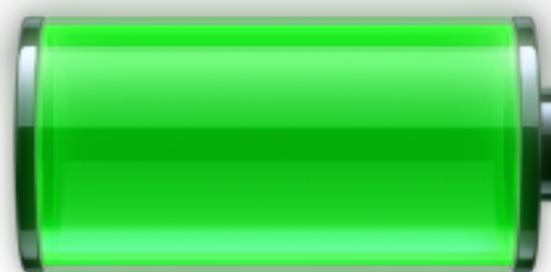


# Memory management in iOS

- iOS uses a virtual memory system: each program has its own virtual address space
- iOS runs on constrained devices, in terms of available memory: memory is limited to the amount physical memory available
- iOS does not support paging to disk when memory gets full, so the virtual memory system releases memory if it needs more space
- Notifications of low-memory are sent to apps, so they can free memory

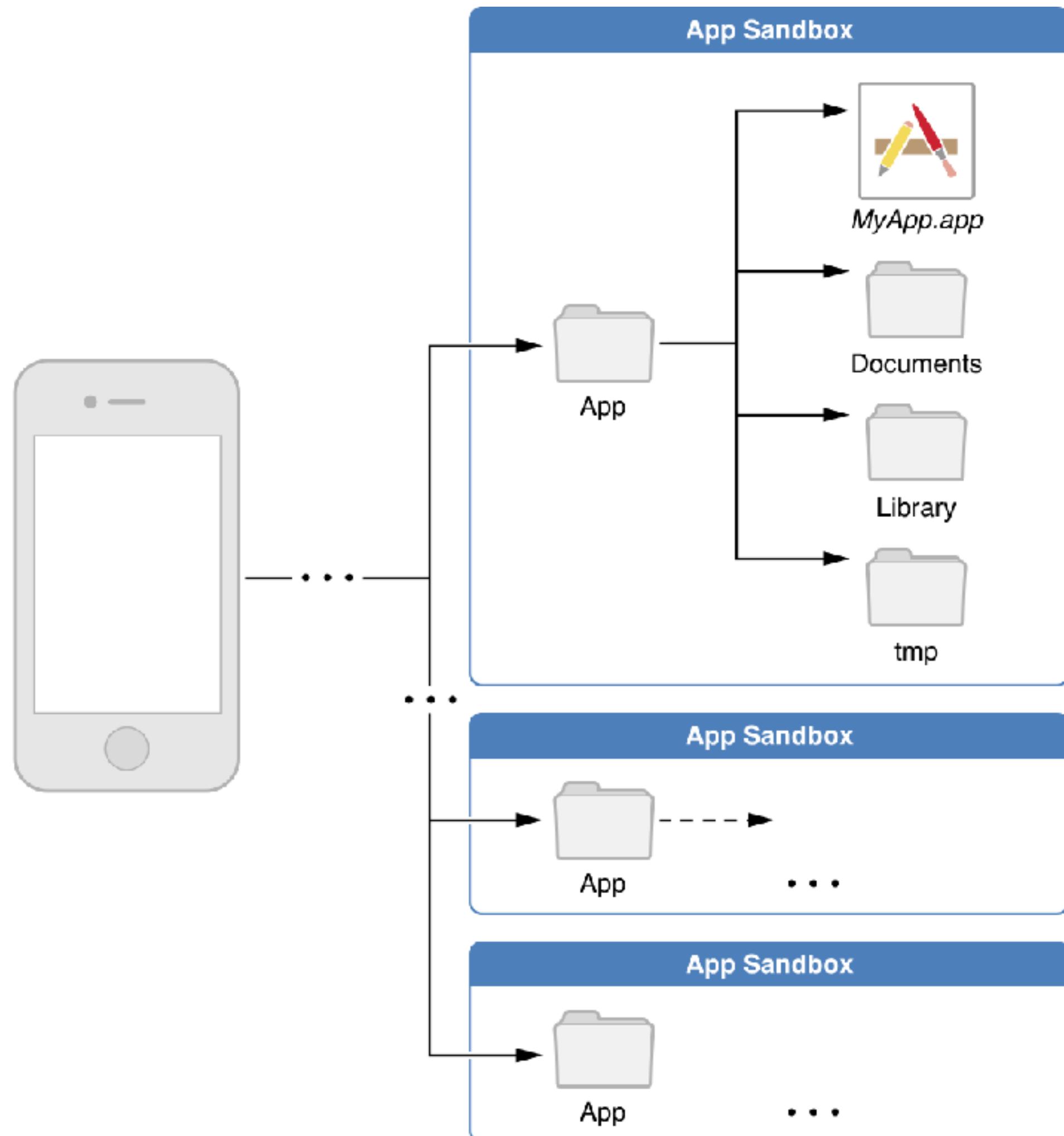
# Multi-threading in iOS

- Since version 4, iOS allows applications to be run in the background even when they are not visible on the screen
- Most background apps reside in memory but do not actually execute any code
- Background apps are suspended by the system shortly after entering the background to preserve battery life
- In some cases, apps may ask the OS for background execution, but this requires a proper management of the app states

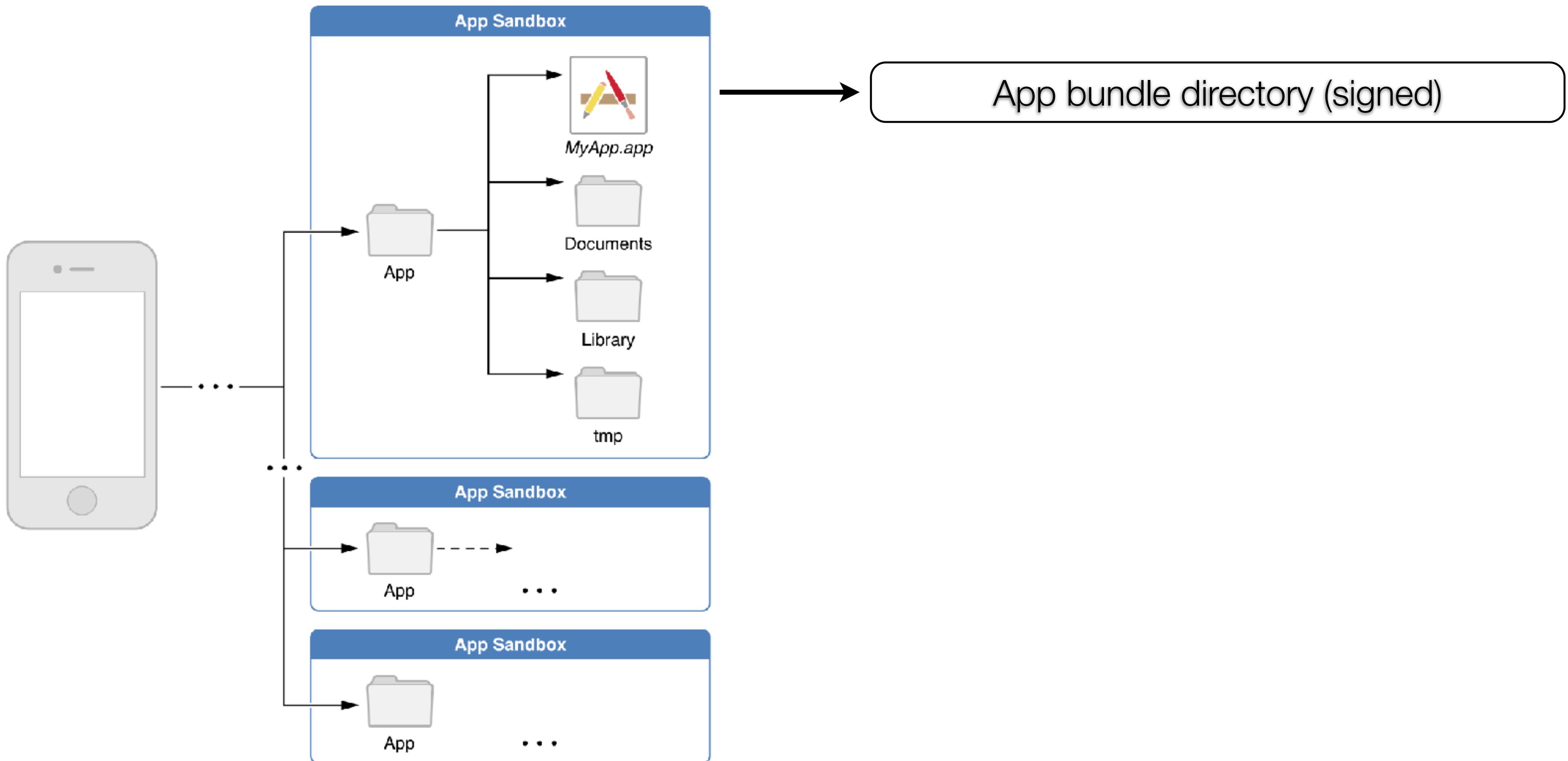


# App sandbox

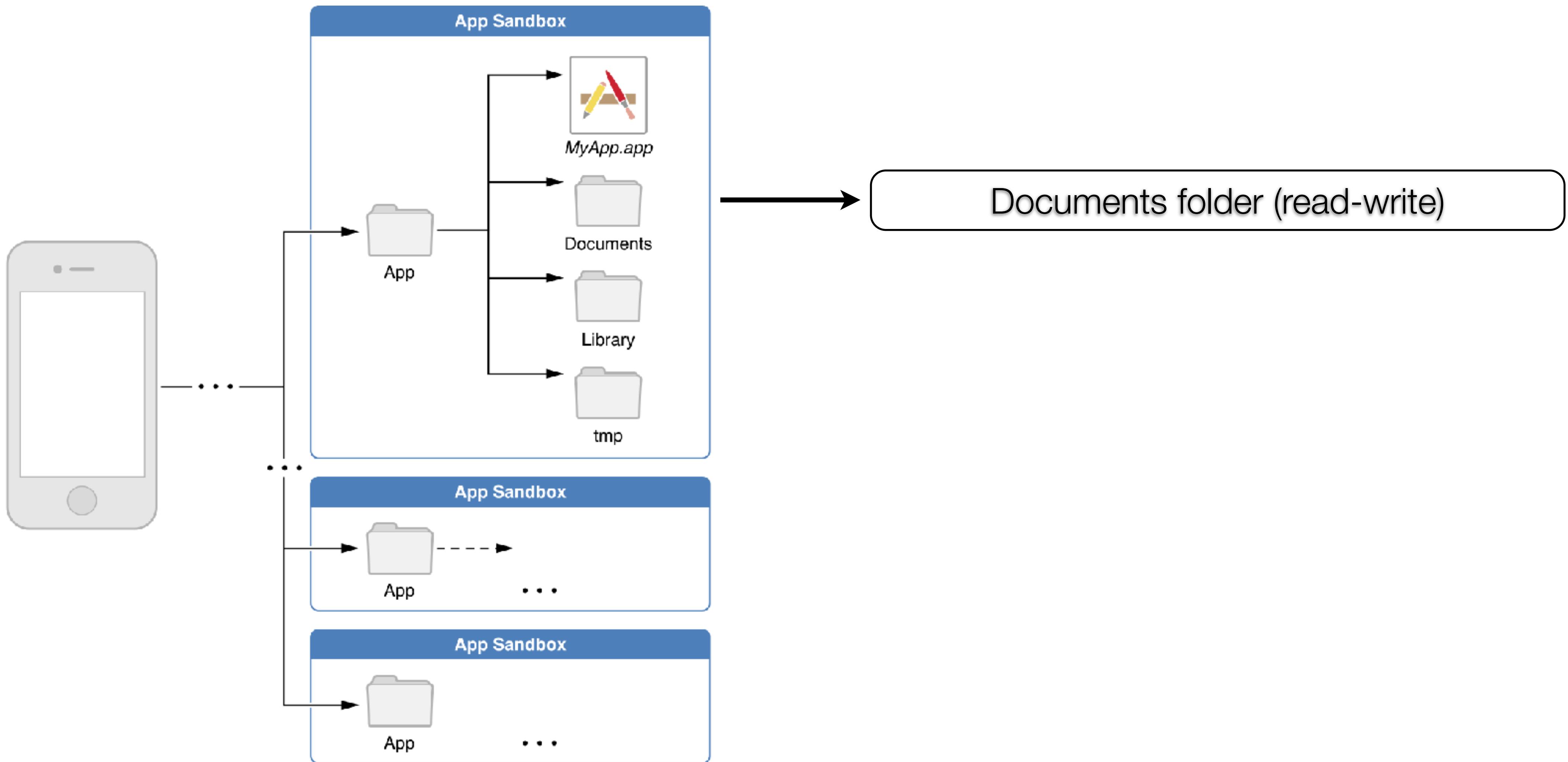
- For security reasons, iOS places each app (including its preferences and data) in a sandbox at install time
- A sandbox provides controls that limit the app's access to files, preferences, network resources, hardware, ...
- The system installs each app in its own sandbox directory, which can be seen as the home for the app and its data
- Each sandbox directory contains several well-known subdirectories for placing files
- **The sandbox only prevents the hijacked app from affecting other apps and other parts of the system, not the app itself**



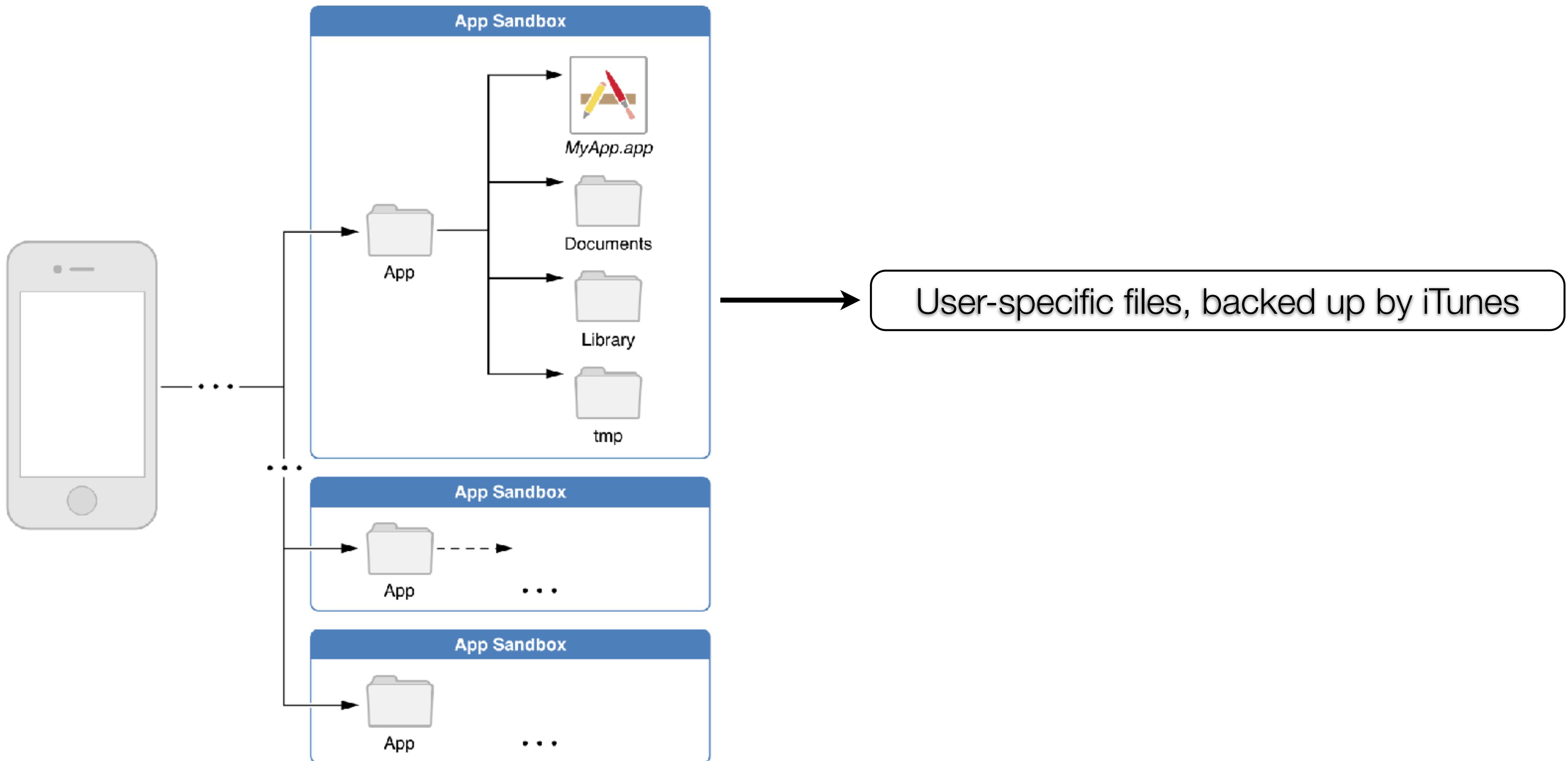
# App sandbox



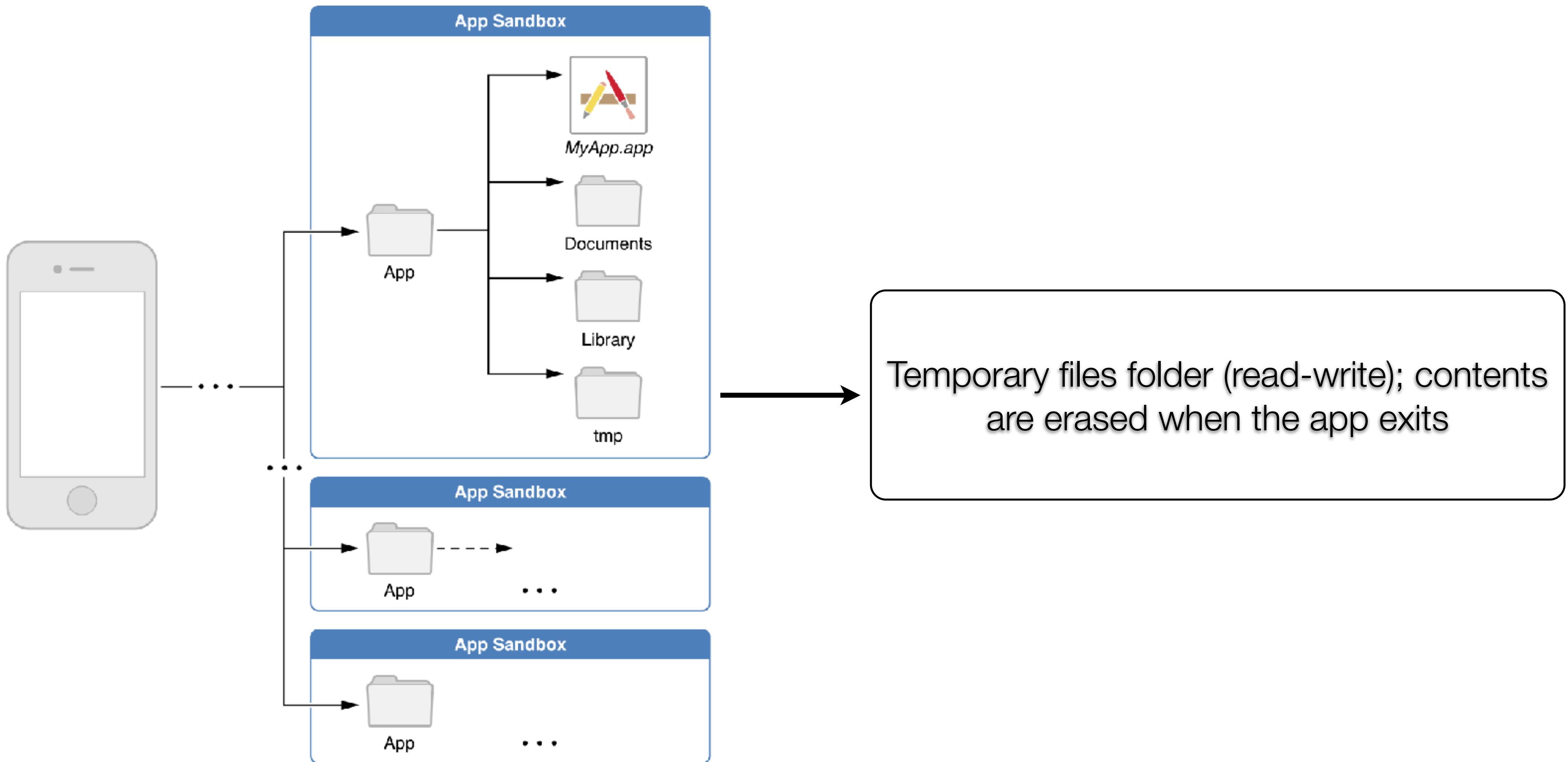
# App sandbox



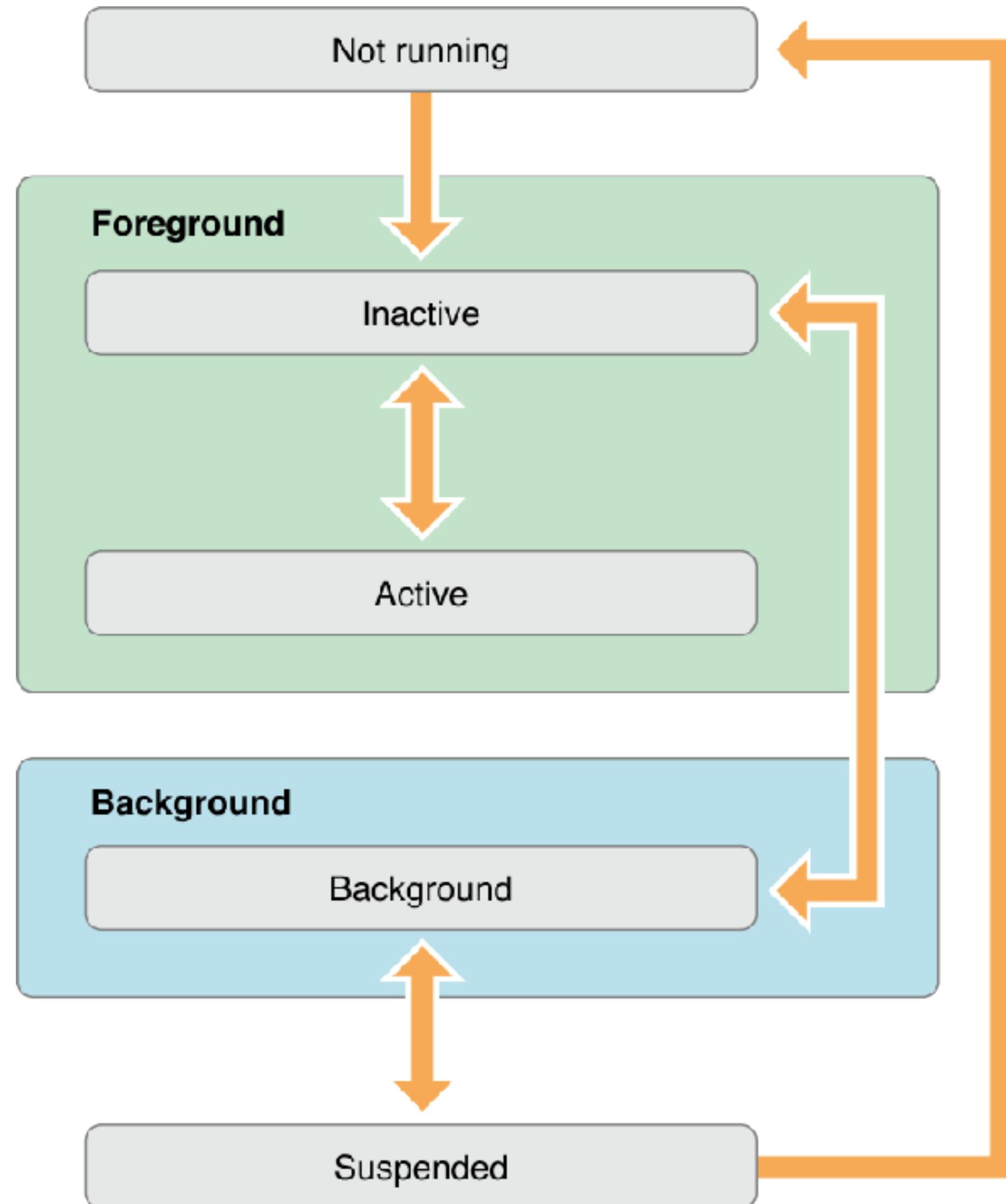
# App sandbox



# App sandbox

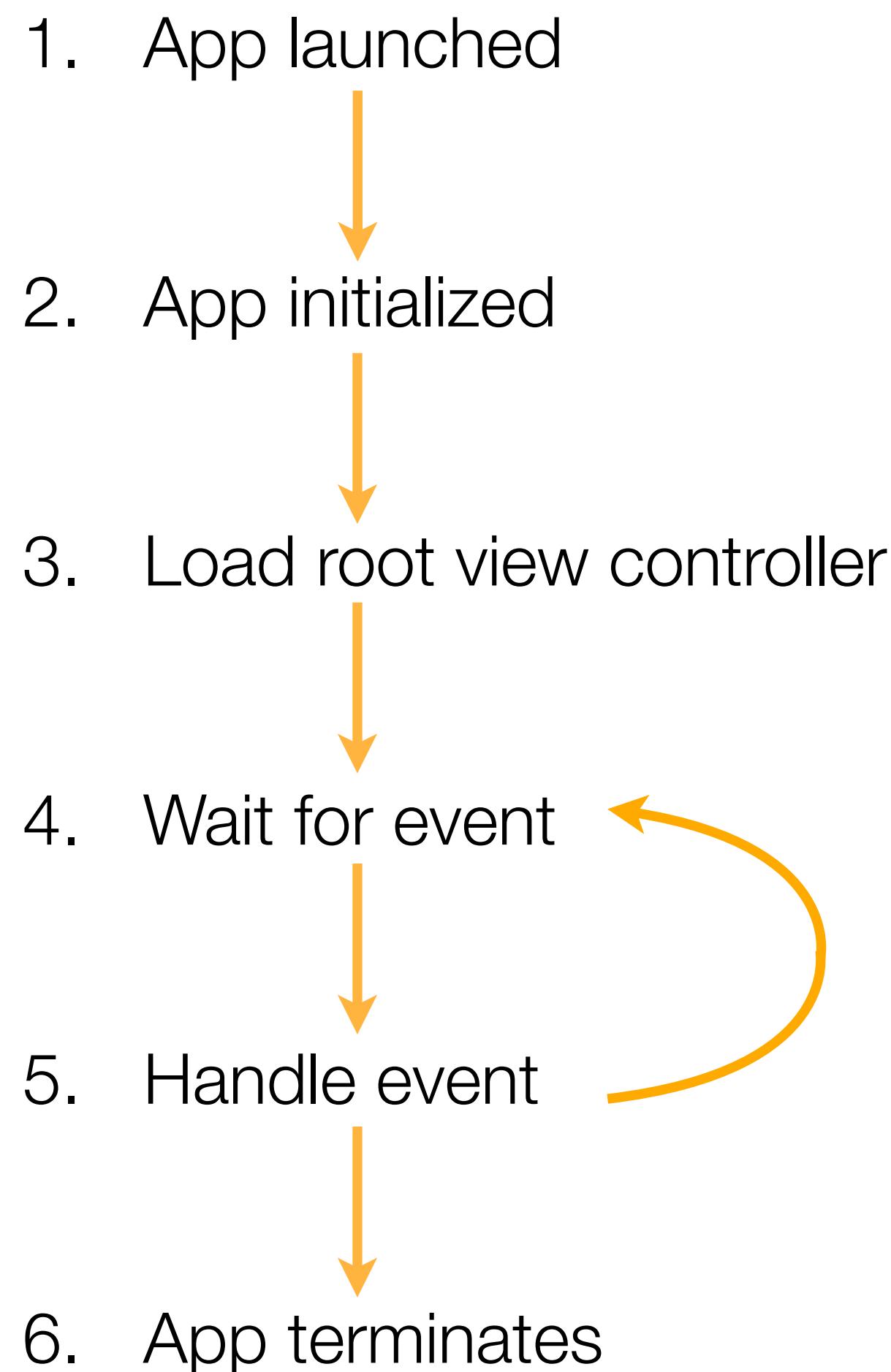


# App Launch Cycle

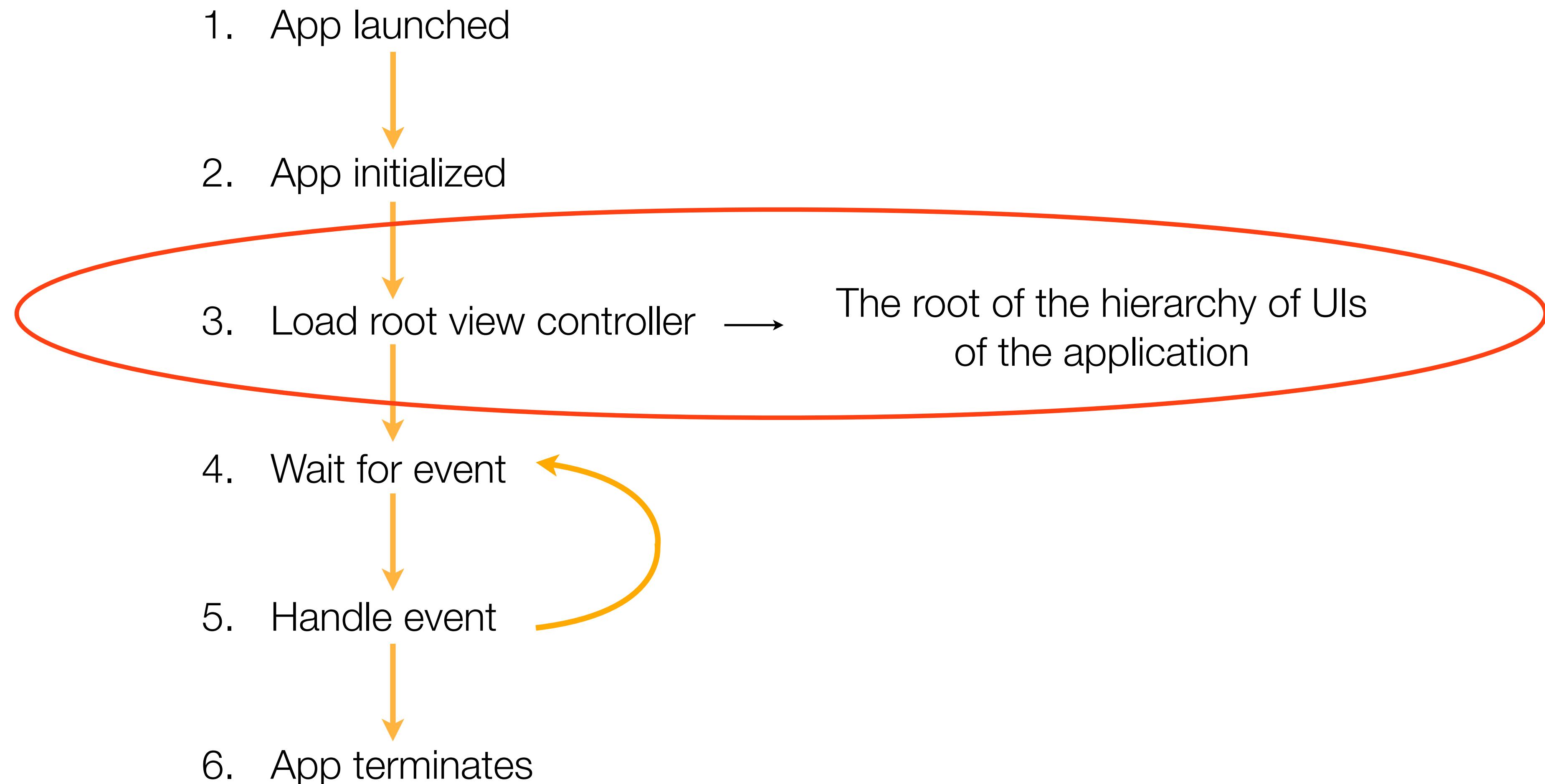


- When the app is launched it moves from the *not-running* state to the *active* or *background* state
- iOS creates a process and main thread for the app and calls the app's main function on that main thread (**main event loop**)
- The main event loop receives events from the operating system that are generated by user actions (e.g. UI-related events)

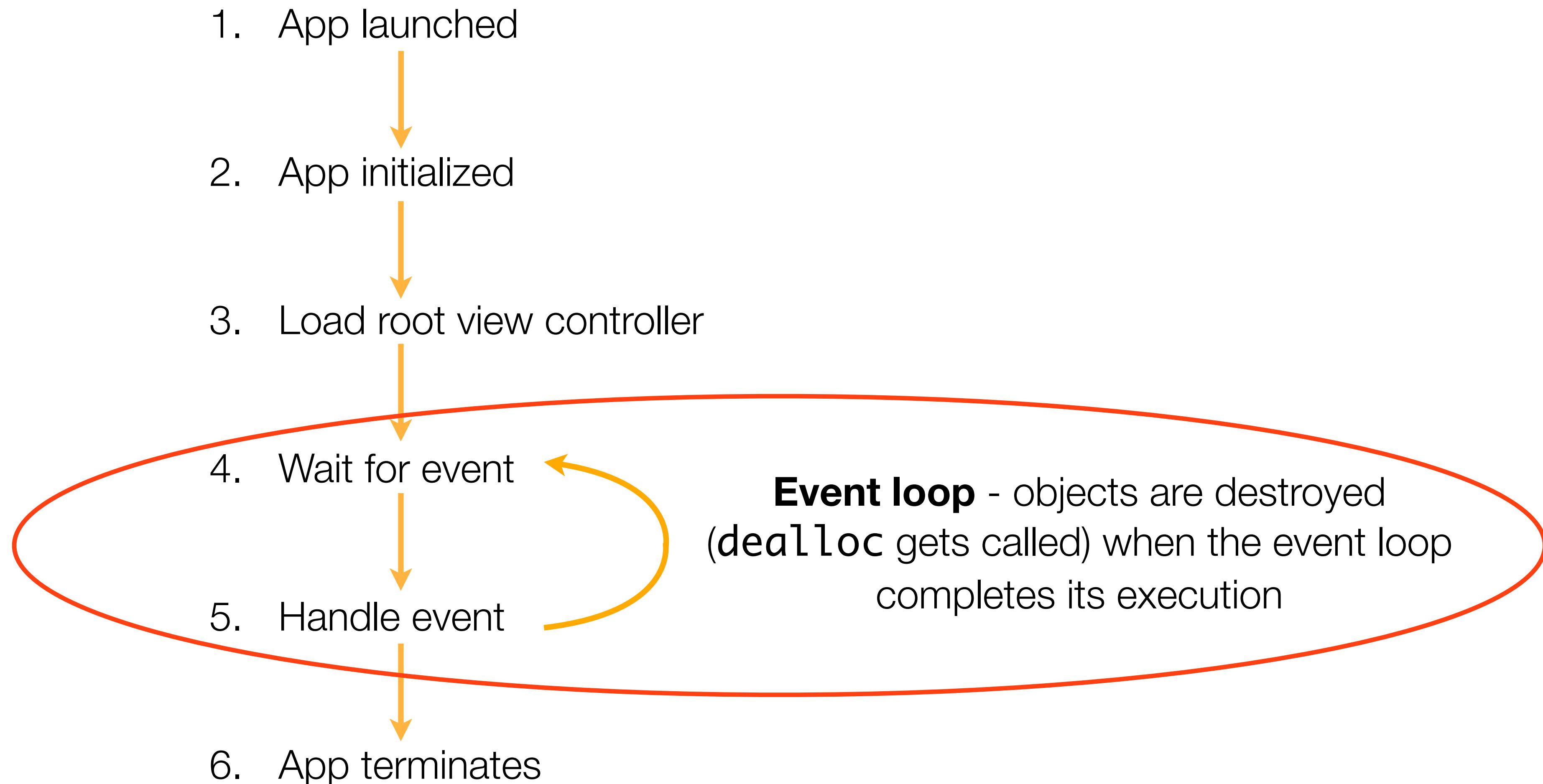
# App Life Cycle



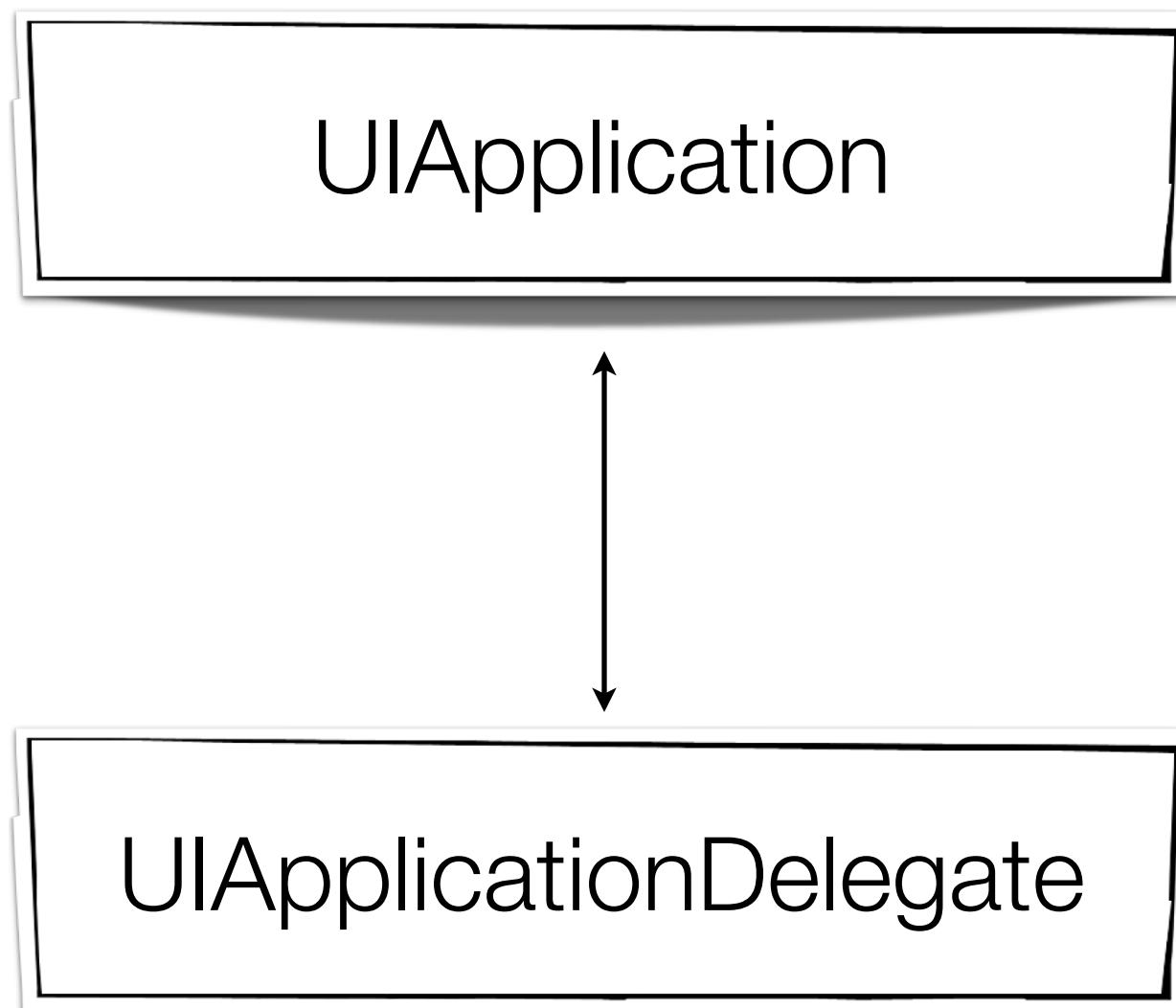
# App Life Cycle



# App Life Cycle



# UIApplication and UIApplicationDelegate



- In iOS, the application is an instance of a class, called **UIApplication**
- The **UIApplication** is never touched directly, it is managed by the operating system
- Each application has a *delegate* object (conforming to the **UIApplicationDelegate** protocol) which receives messages when the app changes its state
- State transitions are accompanied by a corresponding call to the methods of the app delegate object
- These methods are a chance to respond to state changes in an appropriate way

# UIApplicationDelegate protocol methods

**application:willFinishLaunchingWithOptions:** This method is the app's first chance to execute code at launch time

**application:didFinishLaunchingWithOptions:** This method allows you to perform any final initialization before your app is displayed to the user

**applicationDidBecomeActive:** Lets your app know that it is about to become the foreground app; use this method for any last minute preparation

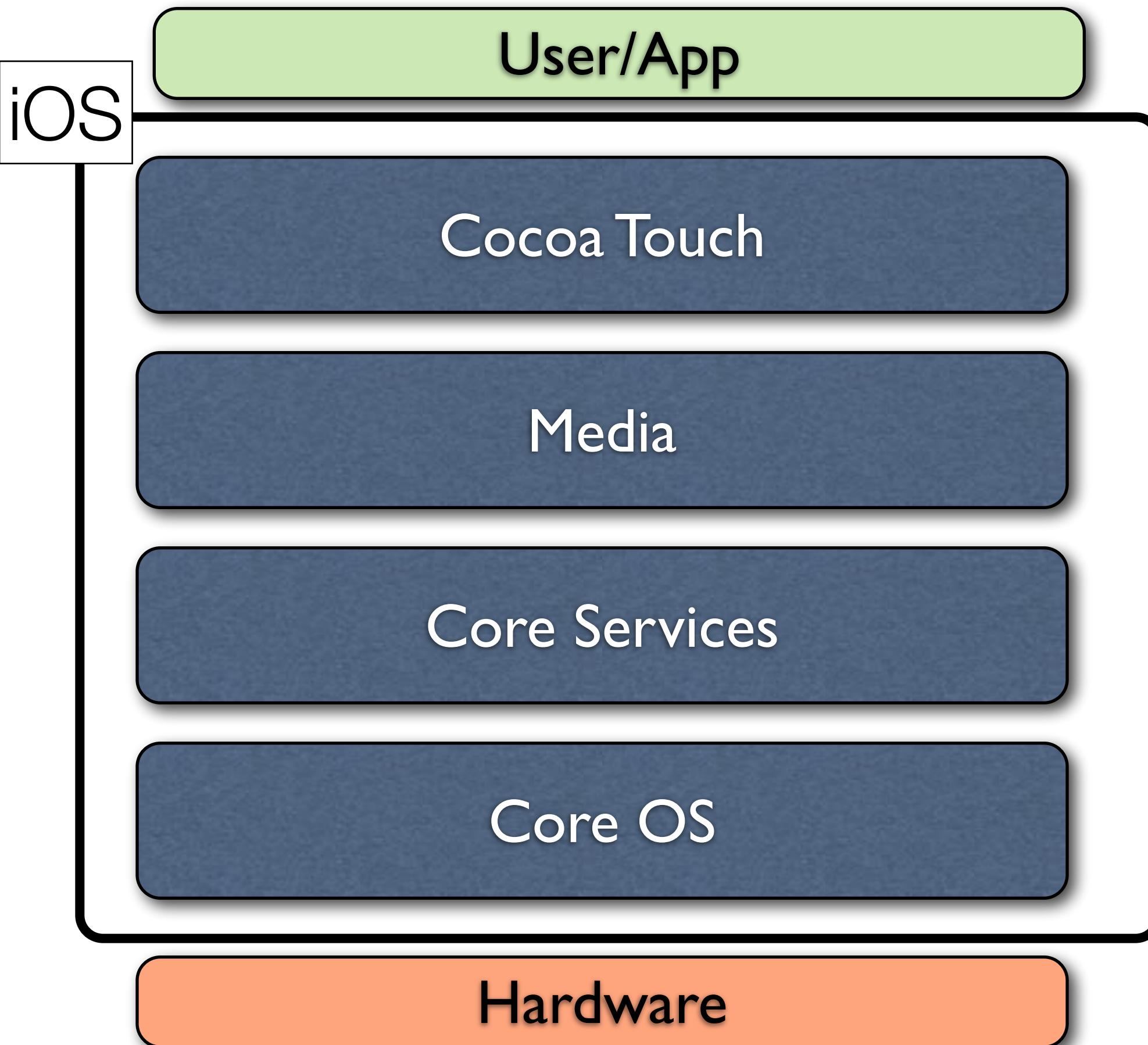
**applicationWillResignActive:** Lets you know that your app is transitioning away from being the foreground app; use this method to put your app into a dormant state

**applicationDidEnterBackground:** Lets you know that your app is now running in the background and may be suspended at any time

**applicationWillEnterForeground:** Lets you know that your app is moving out of the background and back into the foreground, but that it is not yet active

**applicationWillTerminate:** Lets you know that your app is being terminated; this method is not called if app is suspended

# iOS Layers



- The iOS architecture is layered
- iOS acts as an intermediary between the underlying hardware and the apps
- Apps communicate with the hardware through a set of well-defined system interfaces
- Lower layers contain fundamental services and technologies
- Higher-level layers build upon the lower layers and provide more sophisticated services and technologies
- iOS technologies are packaged as frameworks (especially **Foundation and UIKit frameworks**)
- A framework is a directory that contains a dynamic shared library and the resources (such as header files, images, and helper apps) needed to support that library



# iOS Layers

Cocoa Touch

Media

Core Services

Core OS

- Low-level features
- Main frameworks:
  - **Accelerate Framework:** vector and matrix math, digital signal processing, large number handling, and image processing
  - **Core Bluetooth Framework**
  - **Security Framework:** support for symmetric encryption, hash-based message authentication codes (HMACs), and digests
  - **System:** kernel environment, drivers, and low-level UNIX interfaces of the OS; Support for Concurrency (POSIX threads and Grand Central Dispatch), Networking (BSD sockets), File-system access, Standard I/O, Bonjour and DNS services, Locale information, Memory allocation, Math computations



# iOS Layers

Cocoa Touch

Media

Core Services

Core OS

- Fundamental system services for apps
- Main frameworks:
  - **CFNetwork Framework:** BSD sockets, TLS/SSL connections, DNS resolution, HTTP/HTTPS connections
  - **Core Data Framework**
  - **Core Foundation Framework:** (C library) collections, strings, date and time, threads
  - **Core Location Framework:** provides location and heading information to apps
  - **Foundation Framework:** wraps Core Foundation in Objective-C types
  - **System Configuration Framework:** connectivity and reachability



# iOS Layers

Cocoa Touch

Media

Core Services

Core OS

- Graphics, audio, and video technologies
- Main frameworks:
  - **AV Foundation Framework:** playing, recording, and managing audio and video content
  - **Media Player Framework:** high-level support for playing audio and video content
  - **Core Audio Frameworks:** native (low-level) support for handling audio
  - **Core Graphics Framework:** support for path-based drawing, antialiased rendering, gradients, images, colors
  - **Quartz Core Framework:** efficient view animations through Core Animation interfaces
  - **OpenGL ES Framework:** tools for drawing 2D and 3D content



# iOS Layers

Cocoa Touch

Media

Core Services

Core OS

- Frameworks for building iOS apps
- Multitasking, touch-based input, push notifications, and many high-level system services
- Main frameworks:
  - **UIKit Framework:** construction and management of an application's user interface for iOS
  - **Map Kit Framework:** scrollable map to be incorporated into application user interfaces
  - **Game Kit Framework:** support for Game Center
  - **Address Book Framework:** standard system interfaces for managing contacts
  - **MessageUI Framework:** interfaces for composing email or SMS messages
  - **Event Kit Framework:** standard system interfaces for managing calendar events



# UIKit Framework

- The UIKit framework provides the classes needed to construct and manage an application's user interface for iOS
- It provides an application object, event handling, drawing model, windows, views, and controls specifically designed for a touch screen interface
- UIKit provides:
  - Basic app management and infrastructure, including the app's main run loop
  - User interface management, including support for storyboards and nib files
  - A view controller model to encapsulate the contents of your user interface
  - Objects representing the standard system views and controls
  - Support for handling touch- and motion-based events



# iOS SDK

- The iOS Software Development Kit (SDK) contains the tools and interfaces needed to develop, install, run, and test native apps
- Tools: **Xcode**
- Language: **Objective-C** (plus some C/C++)
- Libraries: **iOS frameworks**
- Documentations: **iOS Developer Library** (API reference, programming guides, release notes, tech notes, sample code...)

# Xcode

- Xcode is the development environment used to create, test, debug, and tune apps
  - The Xcode app contains all the other tools needed to build apps:
    - **Interface Builder**
    - **Debugger**
    - **Instruments**
    - **iOS Simulator**
  - Xcode is used to write code which can be run on the simulator or a connected iDevice
  - Instruments is used to analyze the app's behavior, such as monitoring memory allocation





# Model-View-Controller

- All iOS applications are built using the **MVC pattern**
- MVC is used to organize parts of code into clean and separate fields, according to the responsibilities and features of each of them
- This organization is extremely important because it provides a way to create applications that are easy to write, maintain, and debug
- Basically, the way that the iOS SDK is built, drives developers to build applications using MVC
- Understanding and enforcing MVC is 90% of the job when developing in iOS



# Model-View-Controller

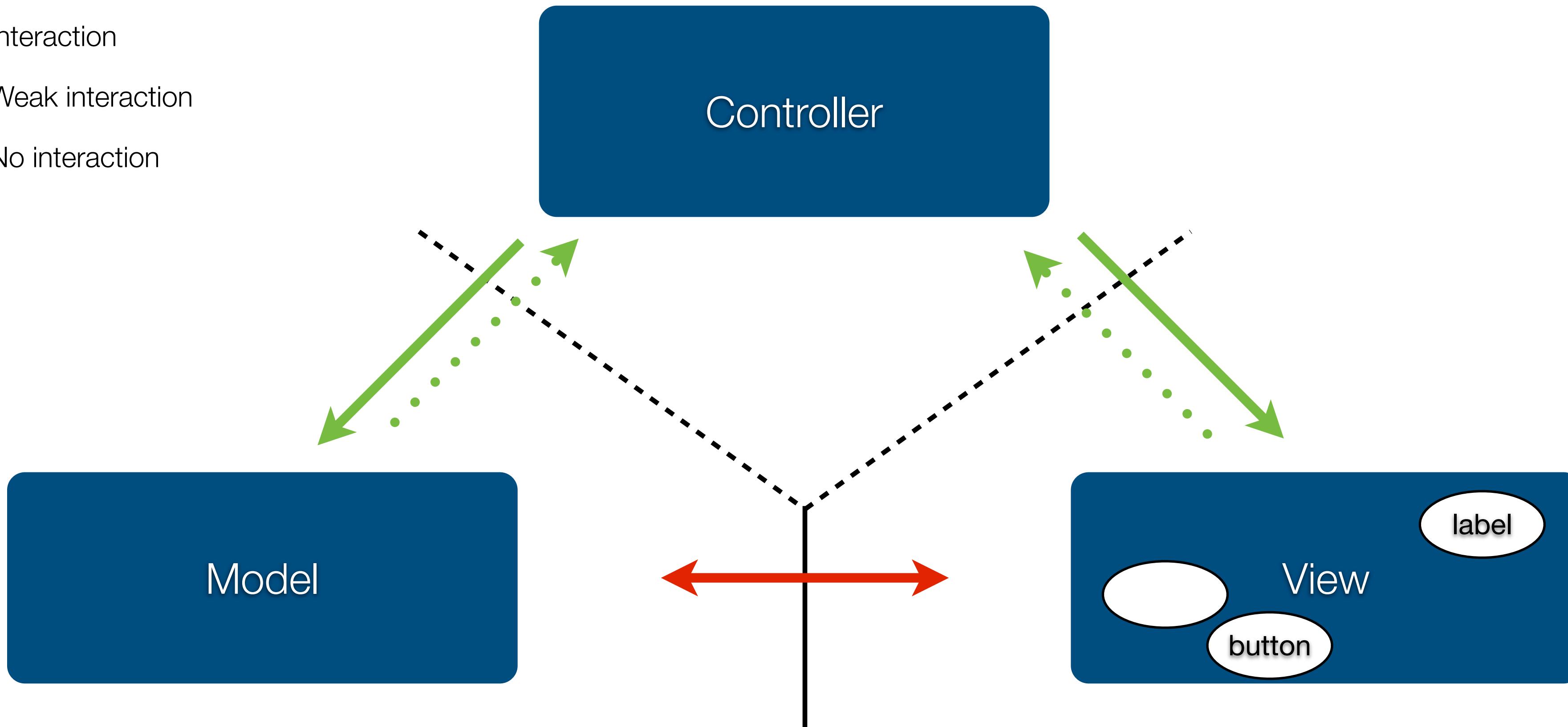
- MVC stands for Model-View-Controller
- An application's code can belong either to the model, to the view, or to the controller
- The **Model** is the representation of the data that will be used in the application; the model is independent from the View, since it does not know how data will be displayed (e.g. iPod library)
- The **View** is the user interface that will display the application's contents; the view is independent from the Model since it contains a bunch of graphical elements that can be used in any application (e.g. buttons, labels, sliders, ...)
- The **Controller** is the brain of the application: it manages how the data in the Model should be displayed in the View; it is highly dependent from the Model and the View since it needs to know which data it will handle and which graphical elements it will need to interact with
- The Controller coordinates and manages the application's UI logic

# MVC interactions

— Interaction

• • • • Weak interaction

— No interaction



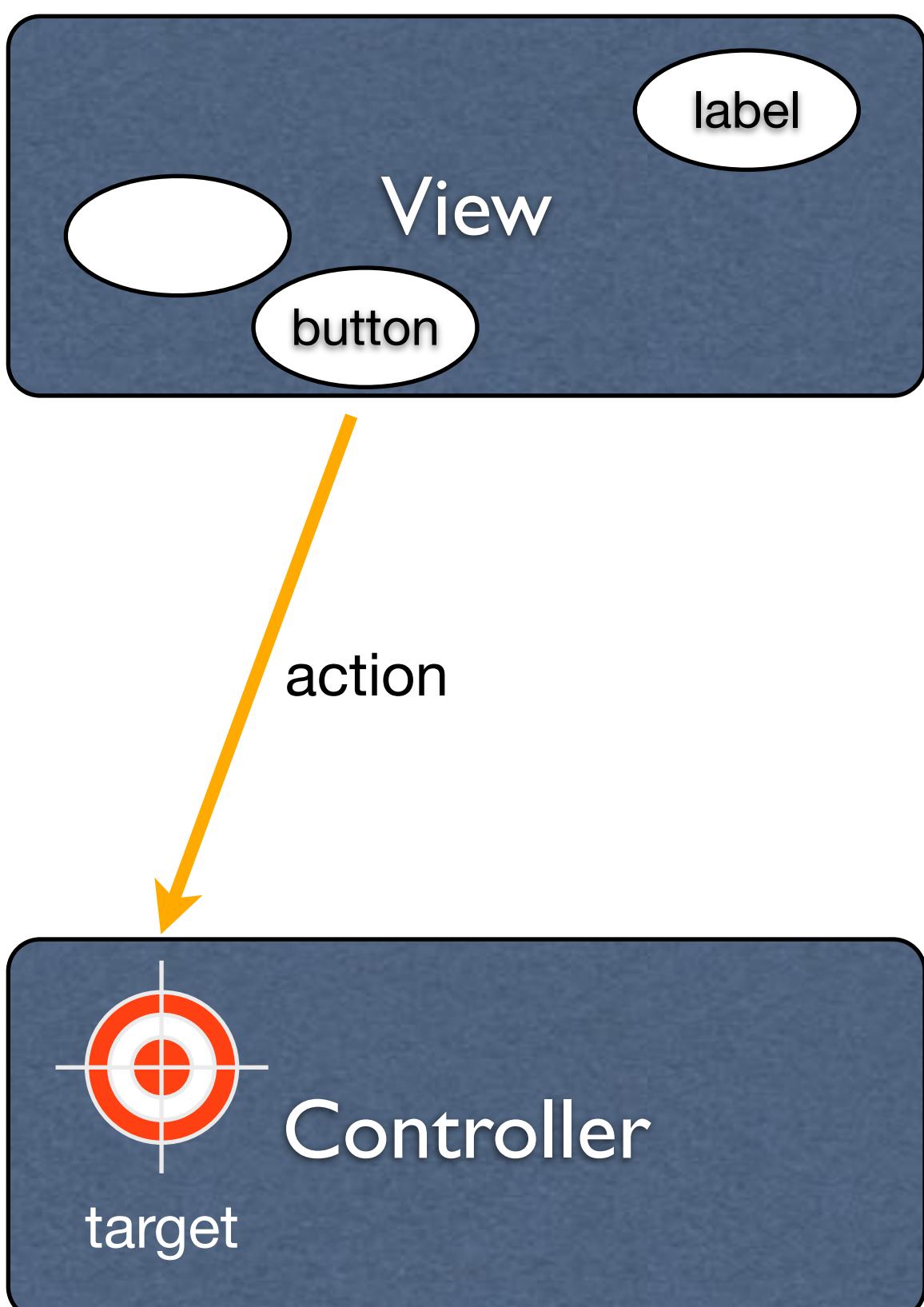


# Model-View-Controller interactions

- The Model has direct interaction with neither the View (obviously) nor the Controller, since its responsibility is just to keep the data (e.g. a database)
- The Controller has direct interaction with the Model since it needs to retrieve and store the data
- The Controller has direct interaction with the View since it needs to update UI elements
- The Controller keeps a reference to UI elements it will use, called **outlets (IBOutlet)**
- The View has no direct interaction with neither the Model (obviously) nor the Controller, since its job is just to display UI elements on the screen

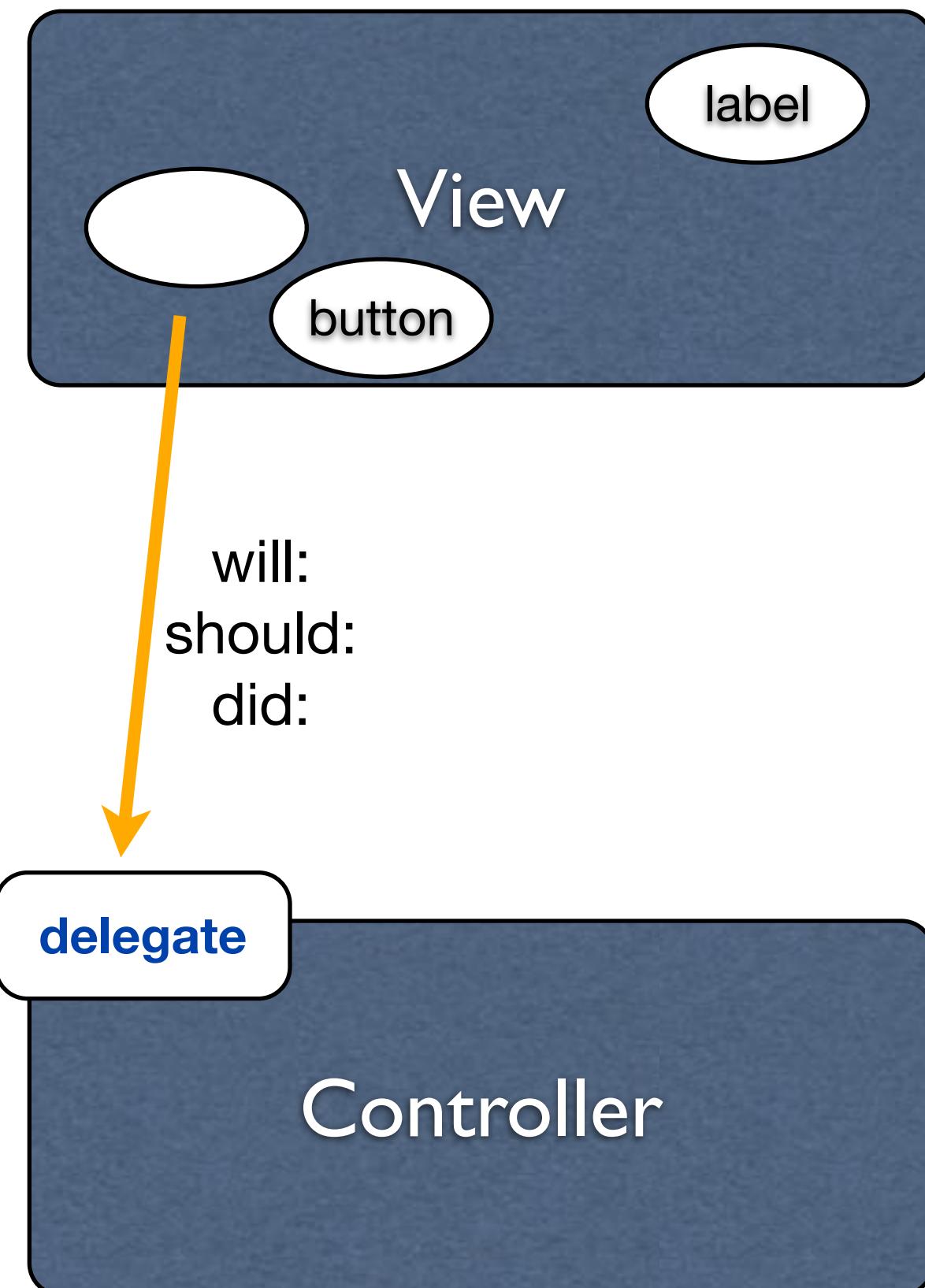
# View-to-Controller interactions

- The View does not interact directly with the Controller, since View classes do not even know about the existence of the Controller
- However, the View should inform the Controller that certain events have occurred (e.g. a button has been clicked)
- The interaction between a View and its Controller occurs in a blind way, through **actions (IBAction)**
- The Controller can register to the View to be the **target** for an action; when the action is performed, the View will send it to the target
- This interaction model lets the View be totally independent from the Controller, yet allows the View to interact with the Controller



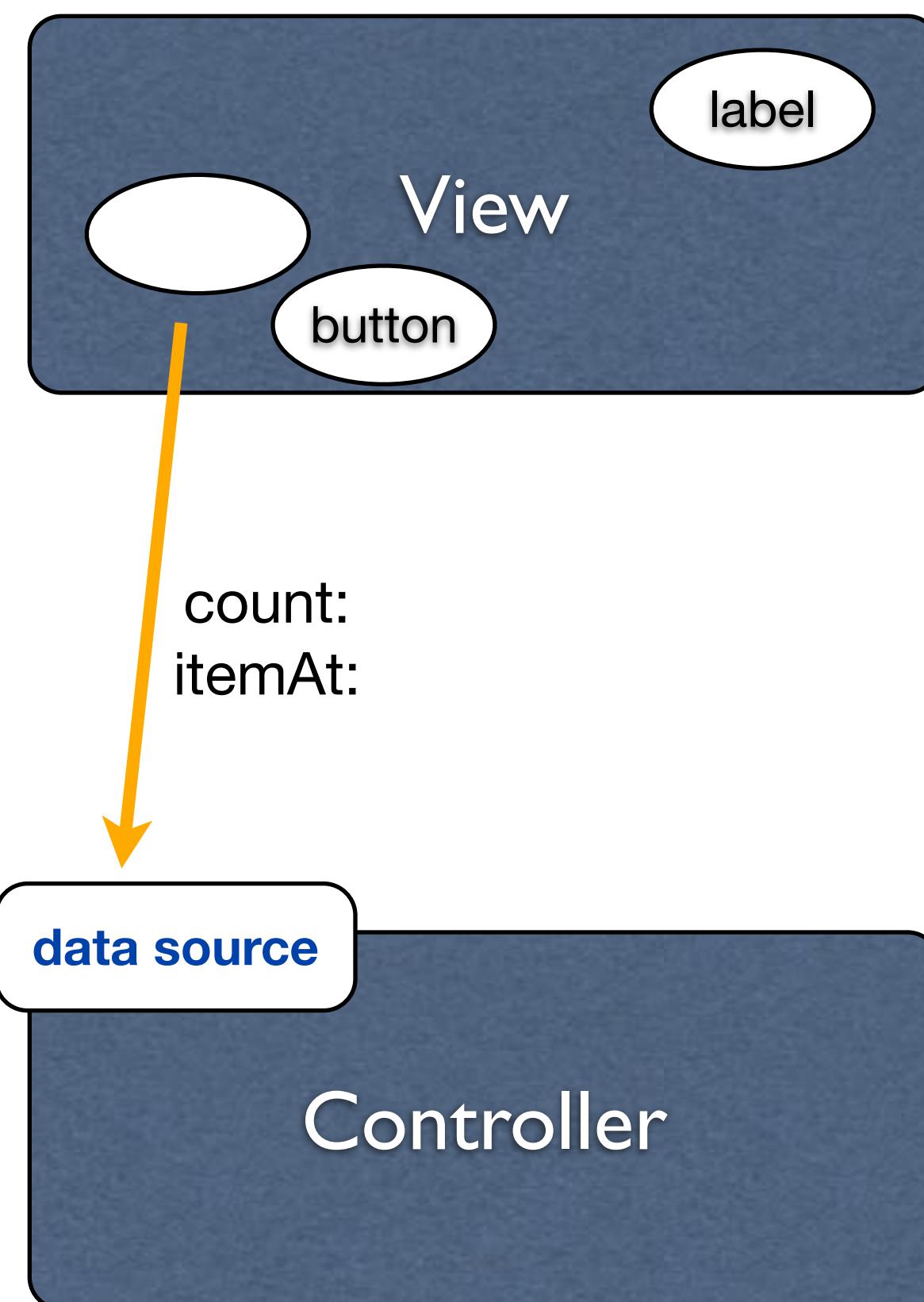
# View-to-Controller interactions

- Some Views interact with the Controller, in order to coordinate (synchronize)
- When certain events **should**, **will**, or **did** occur (e.g. a list item was selected), the View must inform the Controller so that it can perform some operations
- This is called **delegation**: the Controller is the delegate, which means that the view passes the responsibility to the Controller to accomplish certain tasks
- There is a loose coupling between the View and the Controller
- Delegation is accomplished by using protocols



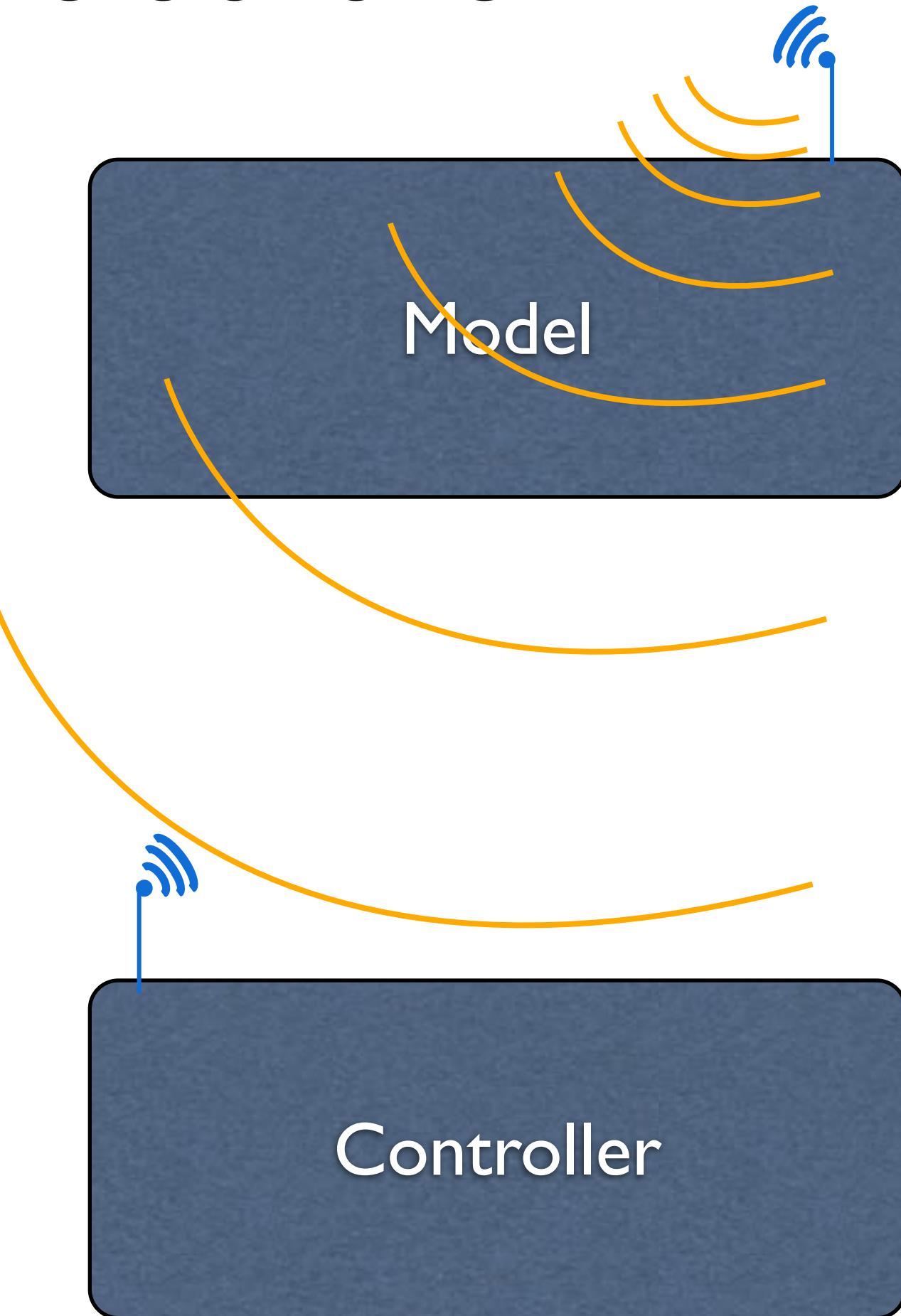
# View-to-Controller interactions

- Some Views do not have enough information to be displayed directly (e.g. a list of elements)
- In general, Views do not own the data that they display, data belong to the Model
- The View needs the Controller to provide those data so that it can display them
- The Controller is a **data source** for the View
- Again, this is accomplished by using protocols
- Data source is indeed delegation, since the View is delegating the Controller to be the provider for the data to be displayed
- Data source is a protocol for providing data, delegate is a protocol for handling view-related events



# Model-to-Controller interactions

- The Model cannot interact with the View, because it is UI-independent
- When data change, the Model should inform the Controller so that it can instruct the View to change what is being displayed
- The Model “broadcasts” the change event
- If the Controller is interested in the event, it will be notified
- This interaction occurs through **notifications** or **KVO (key-value observing)**





# Model-View-Controller

- The Controller's job is to retrieve and format data from the Model so that it can be displayed in the View
- Most of the work when developing apps is done within the Controller(s)
- Complex applications require several MVC to come into play, for instance when an event on a view causes another view to be displayed (typically, this is done by a Controller interacting with other Controllers)
- Some parts of a MVC are other MVCs (e.g. the tabs of a tabbed view are separate MVCs)

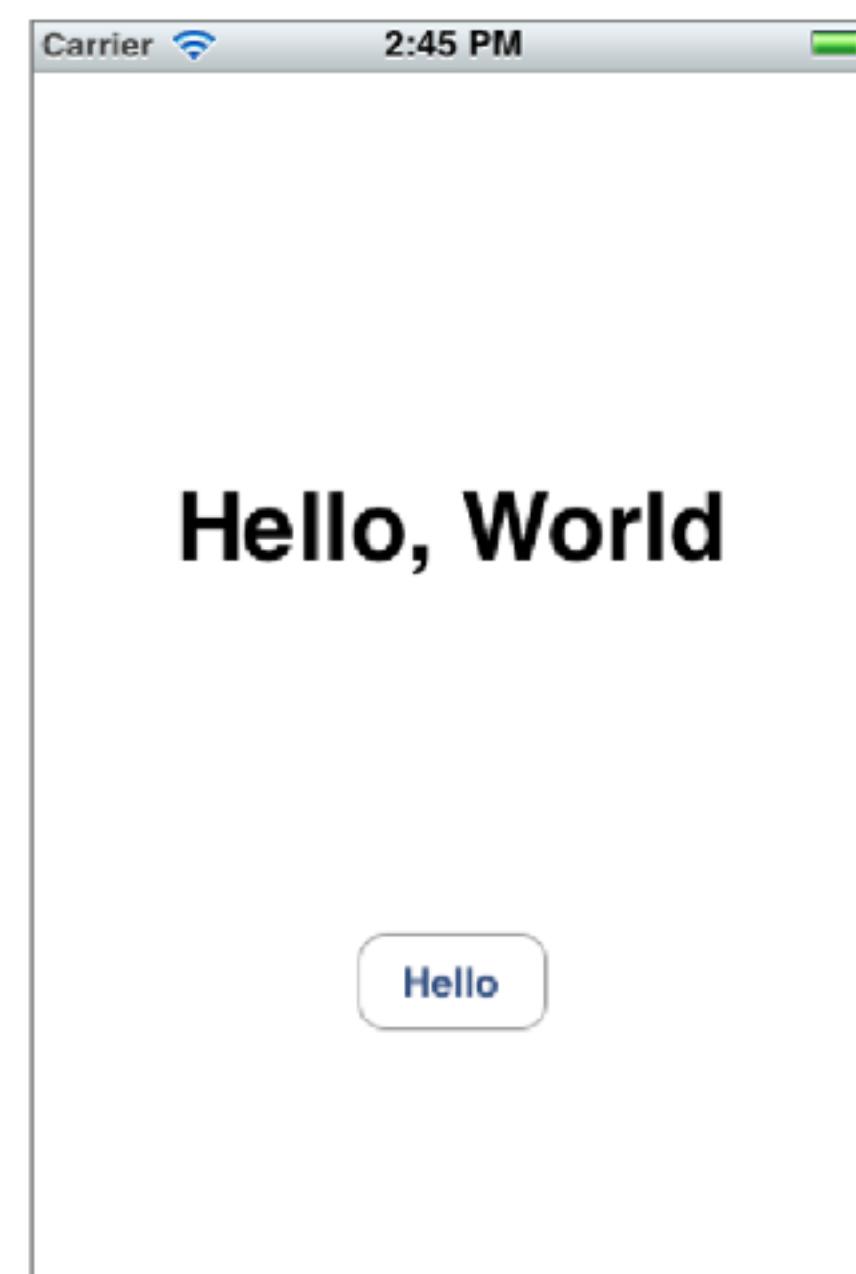
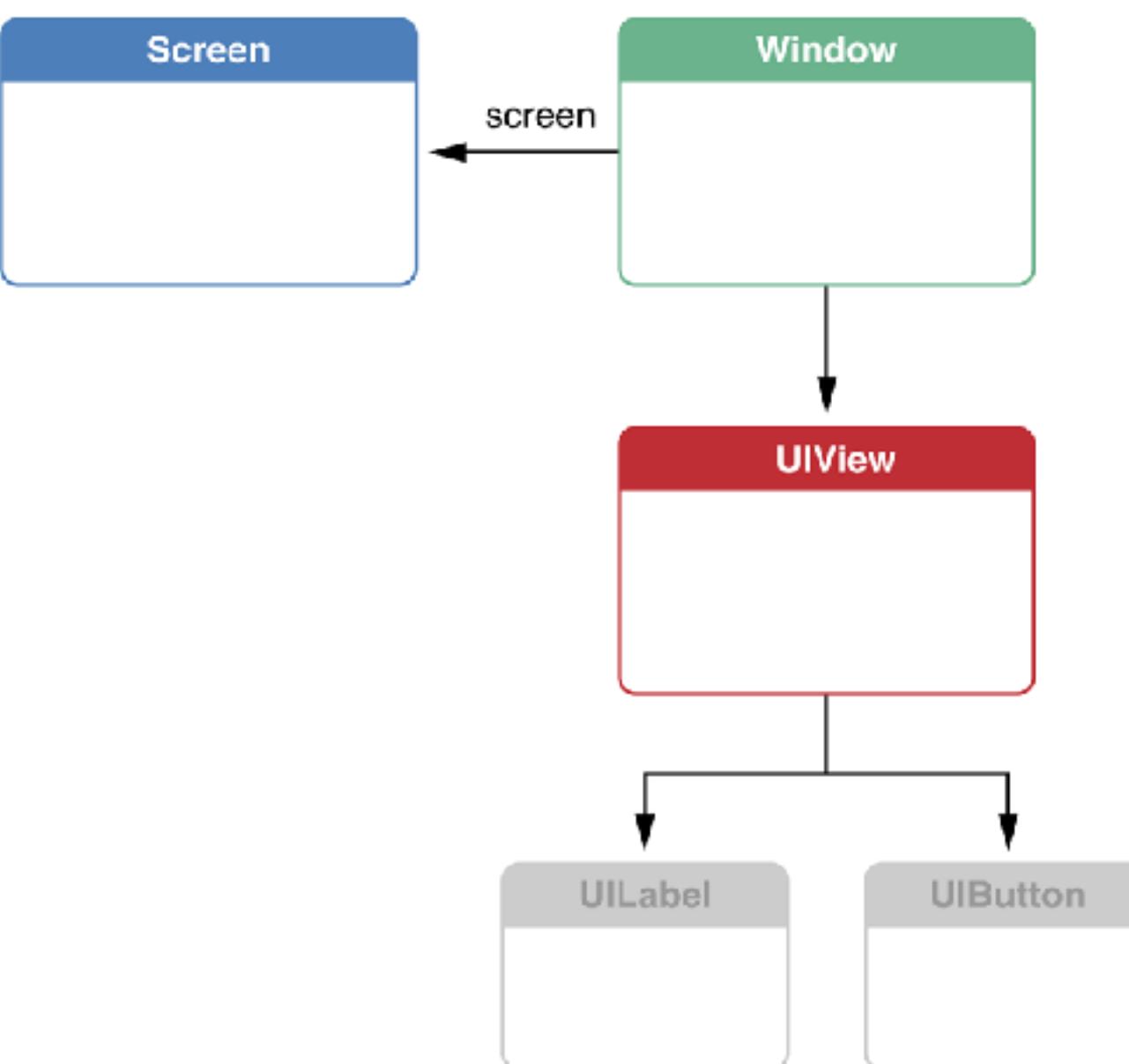


# View Controllers

- View controller objects provide the infrastructure for managing content and for coordinating the showing and hiding of it
- By having different view controller classes control separate portions of user interface, the implementation of the user interface is broken up into smaller and more manageable units
- View controller objects represent the Controller part of the application's MVC

# User interface: Screen, Window, and View

- **UIScreen** identifies a physical screen connected to the device
- **UIWindow** provides drawing support for the screen
- **UIView** objects perform the drawing; these objects are attached to the window and draw their contents when the window asks them to





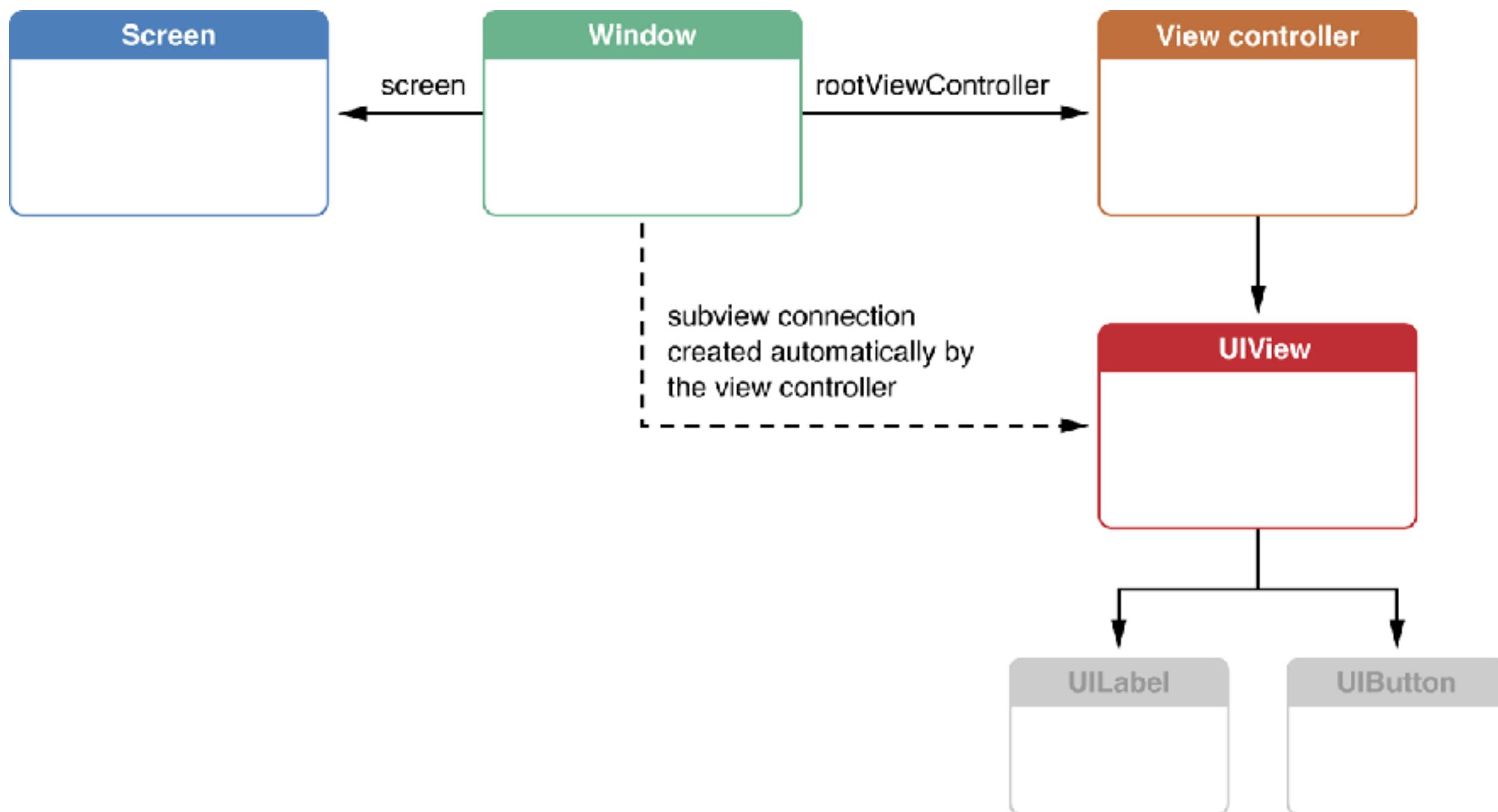
# Views

- A view represents a user interface element; each view covers a specific area; within that area, it displays contents or responds to user events
- Views can be nested in a view hierarchy; subviews are positioned and drawn relative to their superview
- Views can animate their property values; animation are crucial to allow users understand changes in the user interface
- Views typically communicate with the controller through target/action, delegation, and data source patterns
- Complex apps are composed of many views, which can be grouped in hierarchies and animated
- Views that respond to user interaction are called **controls (UIControl)**: **UIButton**s and **UISlider**s are controls

[https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/UIKitUICatalog/index.html#/apple\\_ref/doc/uid/TP40012857](https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/UIKitUICatalog/index.html#/apple_ref/doc/uid/TP40012857)

# View Controllers

- A view controller organizes and controls a view
- A view controller is a controller in the application's MVC
- View controllers are subclasses of the **UIViewController** class
- View controllers also have specific tasks iOS expects them to perform, which are defined in the **UIViewController** class
- Normally, a view controller is attached to a window and automatically adds its view as a subview of the window





# View Controllers

- View controller must carefully load views in order to optimize resource usage
- A view controller should only load a view when the view is needed and it can also release the view under certain conditions (low memory)
- View controller coordinate actions occurring in its connected views
- Because of their generality (which is required for reusability), view objects are agnostic on their meaning in the application and typically send messages to their controller; view controllers, instead, are required to understand and react to certain events that occur in the views they manage



# Views and View Controllers

- Every view is controlled by only one view controller
- A view controller has a `view` property; when a view is assigned to the `view` property, the view controller owns the view
- Subviews might be controlled by different view controllers: several view controllers might be involved in managing portions of a complex view
- Each view controller interacts with a subset of the app's data: they are responsible for displaying specific content and should know nothing about data other than what they show (e.g. Mail app)



# Mobile Application Development



Lecture 3  
iOS SDK



This work is licensed under a [Creative Commons Attribution-  
NonCommercial-ShareAlike 4.0 International License](#).



# Mobile Application Development



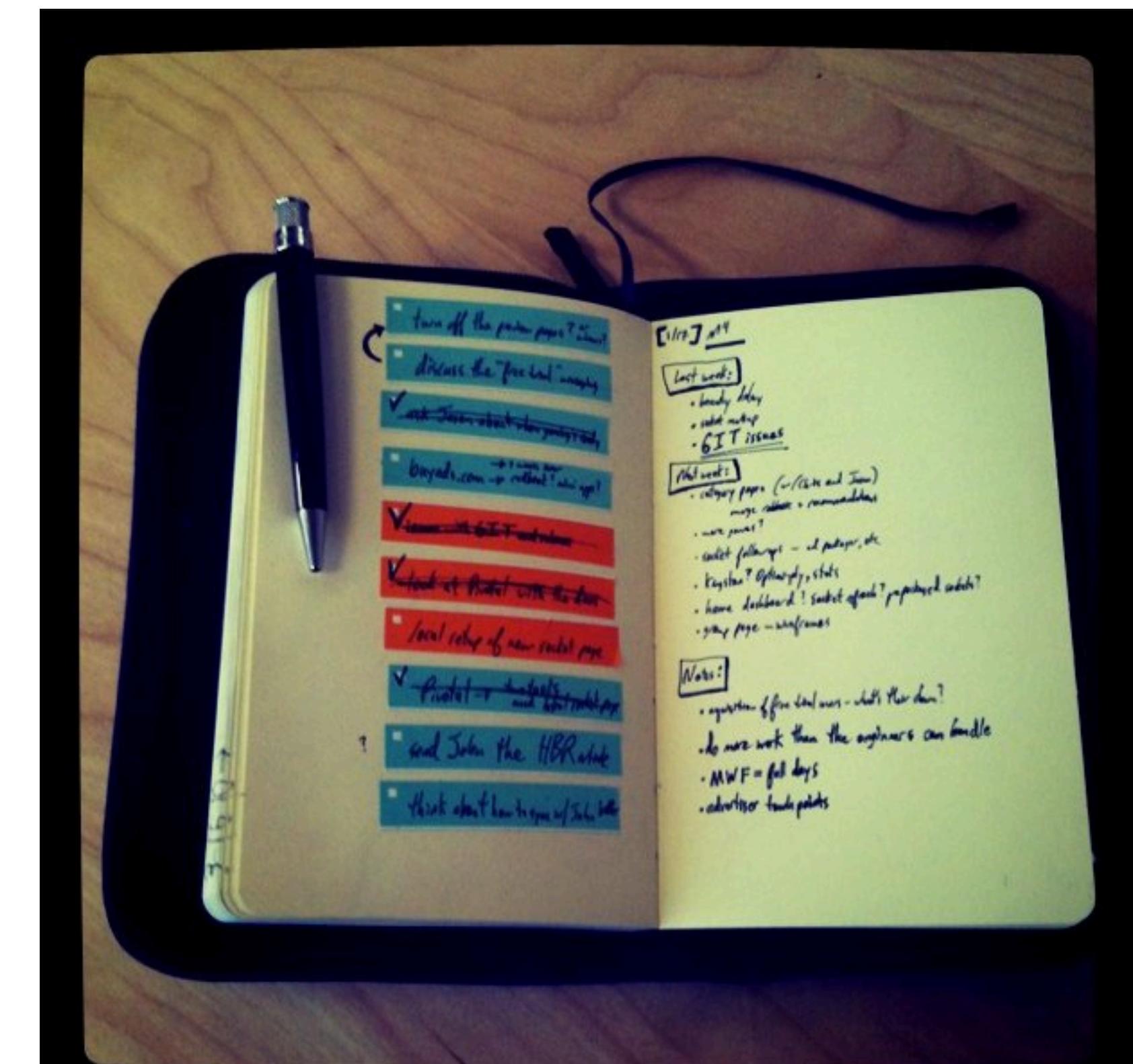
Lecture 4  
UIKit views and controls



This work is licensed under a [Creative Commons Attribution-  
NonCommercial-ShareAlike 4.0 International License](#).

# Lecture Summary

- View Controller lifecycle
- UIColor, UIFont, NSAttributedString
- UIKit views and controls: UILabel, UIButton, UISlider, UISwitch, UITextField, UITextView
- NSNotificationCenter, keyboard notifications





# View Controller Lifecycle

- View controllers receive messages whenever certain events related to the view controller's lifecycle occur
- Every view controller is a subclass of the **UIViewController** class which defines a set of methods that will be executed as the view controller receives lifecycle messages
- Subclasses of **UIViewController** might override such methods to provide specific behavior (the implementation of the superclass must also be called through **super**)



# View Controller Lifecycle

1. The first step in the lifecycle is the creation; view controllers can be created
  - through *storyboards*
  - programmatically
2. Next, the outlets of the view controller are set
3. View controllers (their managed views) may appear and disappear from the screen
4. Low-memory notification

**Anytime one of these events occurs, the system sends a message to the view controller**



# viewDidLoad

- After the view controller has been created and the outlets set, `viewDidLoad` gets invoked
- `viewDidLoad` is where most of the initialization of the view controller can be performed, after the outlets have been set (safe)
- `viewDidLoad` is called only once in the life of a view controller, after it has been created
- At this point all outlets have been set, but bounds property of the view is not set, so it is not safe to perform geometry-based settings in `viewDidLoad`

```
- (void)viewDidLoad{
 [super viewDidLoad];
 // Do any additional setup after loading the view, typically from a nib.
 // ...
}
```



# viewWillAppear

- When the view is about to appear on the screen, `viewWillAppear` gets invoked
- The argument tells whether the view is appearing through an animation or instantly
- This method gets invoked as many times as the view appears on the screen, so it could be invoked multiple times: DO NOT PUT CODE THAT SHOULD BE EXECUTED JUST ONCE, USE `viewDidLoad` TO DO THAT!
- Typically, `viewWillAppear` is used to perform:
  - operations that are related to changes that occurred while the view was not on screen
  - long-running operations that could be unnecessary if the view is never displayed or could block the rendering of the view if executed in `viewDidLoad` (possibly in a separate thread)

```
- (void)viewWillAppear:(BOOL)animated{
 [super viewWillAppear:animated];
 // ...
}
```



# viewWillDisappear

- When the view is about to go off the screen, `viewWillDisappear` gets invoked
- The argument tells whether the view is disappearing through an animation or instantly
- This method gets invoked as many times as the view disappears from the screen, so it could be invoked multiple times
- Typically, `viewWillDisappear` is used to:
  - save view state for later retrieval (e.g. scroll position in a scrollable view)
  - cleanup resources (memory and processing) that are not necessary and could be brought back up the next time the view appears

```
- (void)viewWillDisappear:(BOOL)animated{
 [super viewWillDisappear:animated];
 // ...
}
```



# viewDidAppear and viewDidDisappear

- When the view has appeared on the screen, `viewDidAppear` gets invoked
- The argument tells whether the view has appeared through an animation or instantly
- `viewDidAppear` invoked as many times as the view has come up on the screen, so it could be invoked multiple times
- Conversely, `viewDidDisappear` gets invoked when the view has disappeared from the screen

```
- (void)viewDidAppear:(BOOL)animated{
 [super viewDidAppear:animated];
 // ...
}
```

```
- (void)viewDidDisappear:(BOOL)animated{
 [super viewDidDisappear:animated];
 // ...
}
```



# viewWillLayoutSubviews and viewDidLayoutSubviews

- These methods are invoked when the subviews of the view are about to be or have just been laid out
- Between the execution of `viewWillLayoutSubviews` and `viewDidLayoutSubviews`, **autolayout** is performed
- Geometry-related code can be performed here

```
- (void)viewWillLayoutSubviews{
 [super viewWillLayoutSubviews];
 // ...
}
```

```
- (void)viewDidLayoutSubviews{
 [super viewDidLayoutSubviews];
 // ...
}
```

# Autorotation

- The view controller is responsible to handle device rotation (must be set in the project's settings file **Info.plist**)
- If the view controller's **shouldAutorotate** method returns YES, then the view contents should rotate
  - `(BOOL)shouldAutorotate{  
 return YES;  
}`
- If so, the supported orientations are defined as return value of the **supportedInterfaceOrientations** method
- **supportedInterfaceOrientations** returns a **UIInterfaceOrientationMask**

```
- (NSUInteger)supportedInterfaceOrientations{
 return UIInterfaceOrientationMaskPortrait;
}
```



# Autorotation

- Some methods are invoked to notify the view controller about rotation events
  
- `(void)willRotateToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation  
duration:(NSTimeInterval)duration;`
  
- `(void)willAnimateRotationToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation  
duration:(NSTimeInterval)duration;`
  
- `(void)didRotateFromInterfaceOrientation:(UIInterfaceOrientation)fromInterfaceOrientation;`



# didReceiveMemoryWarning

- When memory is low, the view controller is notified and the `didReceiveMemoryWarning` method gets invoked
- Should this happen, all unnecessary (and big) resources (in the heap) should be released; to do this, strong pointers should be set to nil
- Resources allocated in the memory released at this point should be re-creatable
- Good code should avoid releasing big resources at this point, but should do this well in advance

```
- (void) didReceiveMemoryWarning{
 [super didReceiveMemoryWarning];
 // ...
}
```



# awakeFromNib and initialization of view controllers

- `init` is not invoked on objects instantiated by the storyboard
- `awakeFromNib` is invoked by any object instantiated by the storyboard (views, subviews, view controllers, ...) before outlets are set (before `viewDidLoad`)
- `awakeFromNib` should have initialization code that could not be executed anywhere else
- UIViewController's designated initializer and `awakeFromNib` should have the same initialization code:

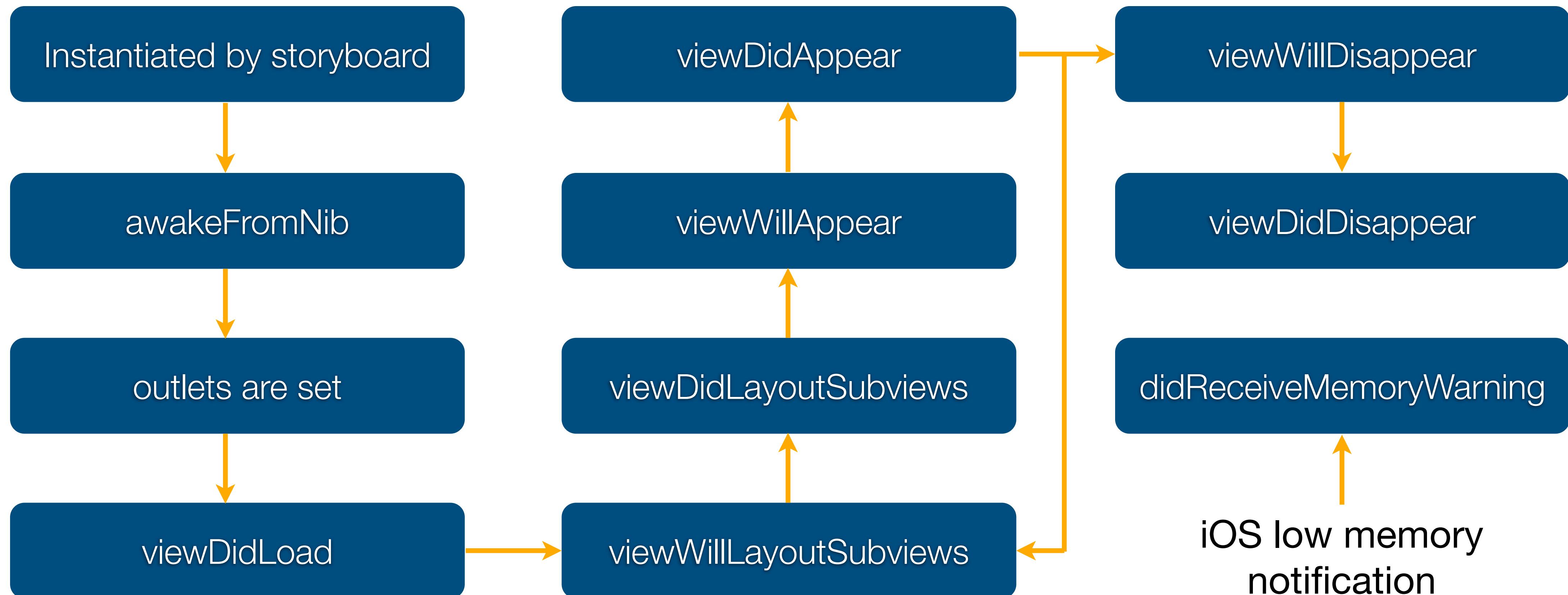
```
- (void)setup{...}

- (void)awakeFromNib{
 [self setup];
}

- (instancetype)initWithNibName:(NSString *)NibNameOrNil bundle:(NSBundle *)nibBundleOrNil{
 self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
 [self setup];
 return self;
}
```

- `viewDidLoad` is preferable

# View Controller Lifecycle





# UIColor

- **UIColor** class represents a color which can be initialized from:
  - RGB (red/green/blue)
  - HSB (hue/saturation/brightness)
  - from images (patterns)
- Colors handle transparency through an **alpha** property, ranging from 0 to 1
- System colors are provided (**blackColor**, **redColor**, **greenColor**, ...)



# Working with fonts

- Fonts can make applications great to see and might hugely improve user experience
- Choosing the right fonts and font sizes is extremely important to make applications good-looking and enjoyable for users
- UIFont class represents fonts; the best way to get a font is to ask the OS for the preferred font for a certain style of text (e.g. UIFontTextStyleHeadline, UIFontTextStyleBody, ...)

```
UIFont *font = [UIFont preferredFontForTextStyle:UIFontTextStyleHeadline];
```

- System fonts are used for buttons, not for content
- Creating fonts by specifying their name:
  - + (UIFont \*)fontWithName:(NSString \*)fontName size:(CGFloat)fontSize
- Example:

```
UIFont *font = [UIFont fontWithName:@"HelveticaNeue-Bold" size:15.0];
```
- To get the list of available font name:
  - + (NSArray \*)fontNamesForFamilyName:(NSString \*)familyName



# NSAttributedString

- **NSAttributedString** object manages character strings and associated sets of attributes (for example, font and kerning) that apply to individual characters or ranges of characters in the string
- Attributes (such as color, stroke width, ...) apply to portions (ranges) of the string
- Key/value pairs (dictionaries of attributes) are assigned to a certain range of a string
- Attributed strings allow developers to render styled text
- The UIKit framework adds methods to **NSAttributedString** to support the drawing of styled strings and to compute the size and metrics of a string prior to drawing
- **NSAttributedString** is used to render fonts on the screen
- **NSAttributedString** is not a subclass of **NSString**; it is possible however to get a **NSString** from a **NSAttributedString** with the **string** method



# NSMutableAttributedString

- **NSMutableAttributedString** are typically created from
  1. a simple (unattributed) **NSString**
  2. an existing **NSMutableAttributedString**
  3. a **NSString** and a **NSDictionary** of attributes
  
- 1. - (id)initWithString:(NSString \*)aString
- 2. - (id)initWithAttributedString:(NSMutableAttributedString \*)attributedString
- 3. - (id)initWithString:(NSString \*)aString attributes:(NSDictionary \*)attributes



# NSAttributedString

- NSAttributedString are typically created from a simple (unattributed) **NSString**, an existing NSAttributedString, or a NSString and a NSDictionary of attributes
  - `(id)initWithString:(NSString *)aString`
  - `(id)initWithAttributedString:(NSAttributedString *)attributedString`
  - `(id)initWithString:(NSString *)aString attributes:(NSDictionary *)attributes`
- A mutable version of NSAttributedString, named **NSMutableAttributedString**, allows you to change dynamically the characters and attributes of the string at runtime through the methods:
  - `(void)addAttribute:(NSString *)name value:(id)value range:(NSRange)aRange`
  - `(void)removeAttribute:(NSString *)name range:(NSRange)aRange`
  - `(void)setAttributes:(NSDictionary *)attributes range:(NSRange)aRange`



# NSAttributedString

- Attributes that can be set are:

| Attribute                      | Value type |
|--------------------------------|------------|
| NSFontAttributeName            | UIFont*    |
| NSForegroundColorAttributeName | UIColor*   |
| NSBackgroundColorAttributeName | UIColor*   |
| NSStrokeWidthAttributeName     | NSNumber*  |
| NSStrokeColorAttributeName     | UIColor*   |



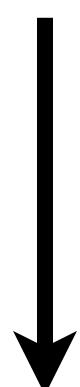
# NSAttributedString example

```
NSString *string = @"testo piccolo, testo grande!";

NSMutableAttributedString *attrString = [[NSMutableAttributedString alloc]
 initWithString:string];

[attrString addAttribute:NSFontAttributeName
 value:[UIFont systemFontOfSize:24.0]
 range:NSMakeRange(15,13)];

[attrString addAttribute:NSForegroundColorAttributeName
 value:[UIColor redColor]
 range:NSMakeRange(6,7)];
```



testo **piccolo**, testo grande!



# UIKit Views

- The **UIView** class defines a rectangular area on the screen and the interfaces for managing the content in that area
- Views can embed other views and create sophisticated visual hierarchies, creating a parent-child relationship between the view and its subviews
- The geometry of a view is defined by some properties:
  - **frame**: origin and dimensions of the view in the coordinate system of its superview
  - **bounds**: the internal dimensions of the view as it sees them
  - **center**: the coordinates of the center point of the rectangular area covered by the view
- All UI elements inherit from **UIView**

[https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/UIKitUICatalog/index.html#/apple\\_ref/doc/uid/TP40012857](https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/UIKitUICatalog/index.html#/apple_ref/doc/uid/TP40012857)

# UIKit Views

- The following are the most commonly used `UIView` properties:

| Property                            | Value type            | Description                                                                                  |
|-------------------------------------|-----------------------|----------------------------------------------------------------------------------------------|
| <code>frame</code>                  | <code>CGRect</code>   | Frame rectangle describing the view's location and size in the superview's coordinate system |
| <code>bounds</code>                 | <code>CGRect</code>   | Bounds rectangle describing the view's location and size in its own coordinate system        |
| <code>center</code>                 | <code>CGPoint</code>  | Center of the frame                                                                          |
| <code>backgroundColor</code>        | <code>UIColor*</code> | Background color; defaults to <code>nil</code> (transparent)                                 |
| <code>alpha</code>                  | <code>CGFloat</code>  | 0.0 means transparent and 1.0 means opaque                                                   |
| <code>hidden</code>                 | <code>BOOL</code>     | YES means the view is invisible, NO means visible                                            |
| <code>userInteractionEnabled</code> | <code>BOOL</code>     | NO means user events are ignored                                                             |

# UILabel

- `UILabel` objects are used to display static text on a fixed (by you) number of lines
- `UILabel` has the following properties to work with the text to display:

| Property       | Value type                       | Description                                                                                                                            |
|----------------|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| text           | <code>NSString*</code>           | Text being displayed                                                                                                                   |
| font           | <code>UIFont*</code>             | Font of the text                                                                                                                       |
| textColor      | <code>UIColor*</code>            | Color of the text                                                                                                                      |
| textAlignment  | <code>NSTextAlignment</code>     | Alignment of the text ( <code>NSTextAlignmentLeft</code> , <code>NSTextAlignmentRight</code> , <code>NSTextAlignmentCenter</code> ...) |
| attributedText | <code>NSAttributedString*</code> | Styled text being displayed                                                                                                            |
| numberOfLines  | <code>NSInteger</code>           | Maximum number of lines to use                                                                                                         |



# UIKit Controls

- A control is a communication tool between a user and an app
- Controls convey a particular action or intention to the app through user interaction
- Controls can be used to manipulate content, provide user input, navigate within an app, or execute other pre-defined actions
- The **UIControl** class (subclass of **UIView**) is the base class for all controls
- **UIControl** is never used, but its subclasses, such as **UIButton** and **UISlider**, are used instead
- Typical controls that can be used in iOS:
  - **Buttons**
  - **Date Pickers**
  - **Page Controls**
  - **Segmented Controls**
  - **Text Fields**
  - **Sliders**
  - **Steppers**
  - **Switches**

# Control states

- A **control state** describes the current interactive state of a control:
  - normal (enabled but not selected or highlighted)
  - selected (e.g. in UISegmentedControl) → 
  - disabled
  - highlighted (when a touch enters and exits during tracking and when there is a touch up event)
- The control state changes as the user interacts with the control
- Specific behavior and appearance can be specified for each control state



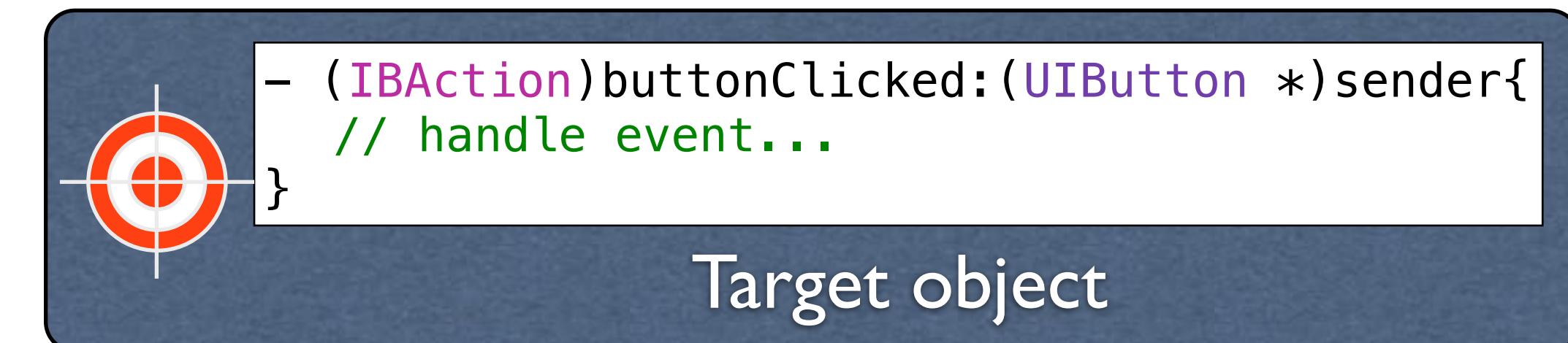
# Control events

- **Control events** represent the ways (physical gestures) that users can make on controls
- Typical control events:
  - `UIControlEventTouchDown`: touch down inside a control
  - `UIControlEventTouchDownRepeat`: repeated touch down
  - `UIControlEventTouchDragInside`: a finger is dragged inside the bounds of the control
  - `UIControlEventTouchDragOutside`: a finger is dragged outside the bounds of the control
  - `UIControlEventTouchDragEnter`: a finger is dragged into the bounds of the control
  - `UIControlEventTouchDragExit`: a finger is dragged from within a control to outside its bounds
  - `UIControlEventTouchUpInside`: a finger is lifted when inside the bounds of the control (typical for `UIButton`)
  - `UIControlEventTouchUpInside`: a finger is lifted when outside the bounds of the control
  - `UIControlEventTouchCancel`: system event canceling the current touches for the control
  - `UIControlEventValueChanged`: touch dragging or otherwise manipulating a control, causing a series of different values

# Target-action

- The **target-action** mechanism is a model for configuring a control to send an action message to a target object after a specific control event

Button

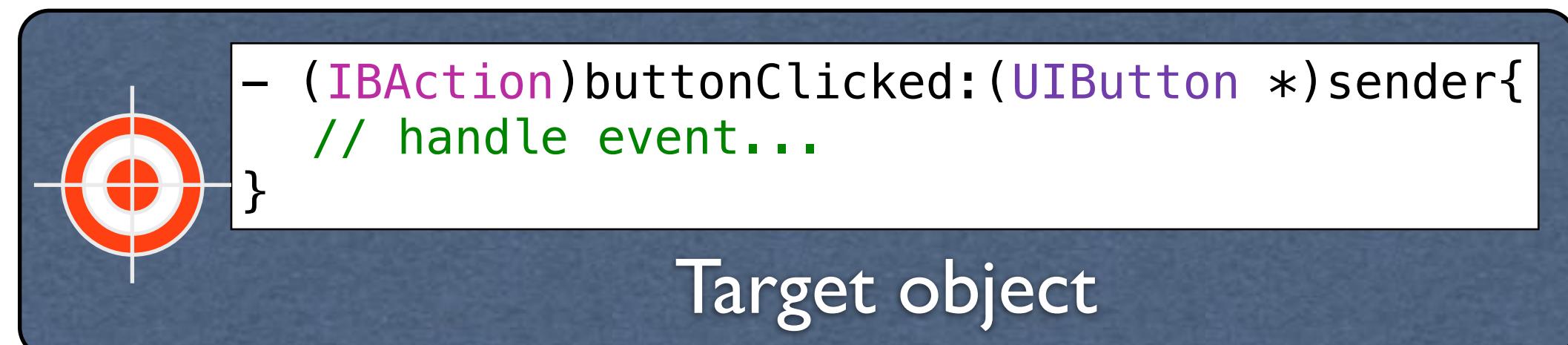


# Target-action

- The **target-action** mechanism is a model for configuring a control to send an action message to a target object after a specific control event

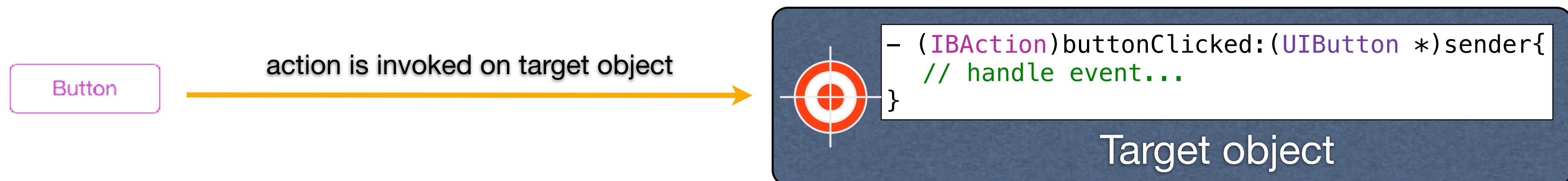


touch event



# Target-action

- The **target-action** mechanism is a model for configuring a control to send an action message to a target object after a specific control event

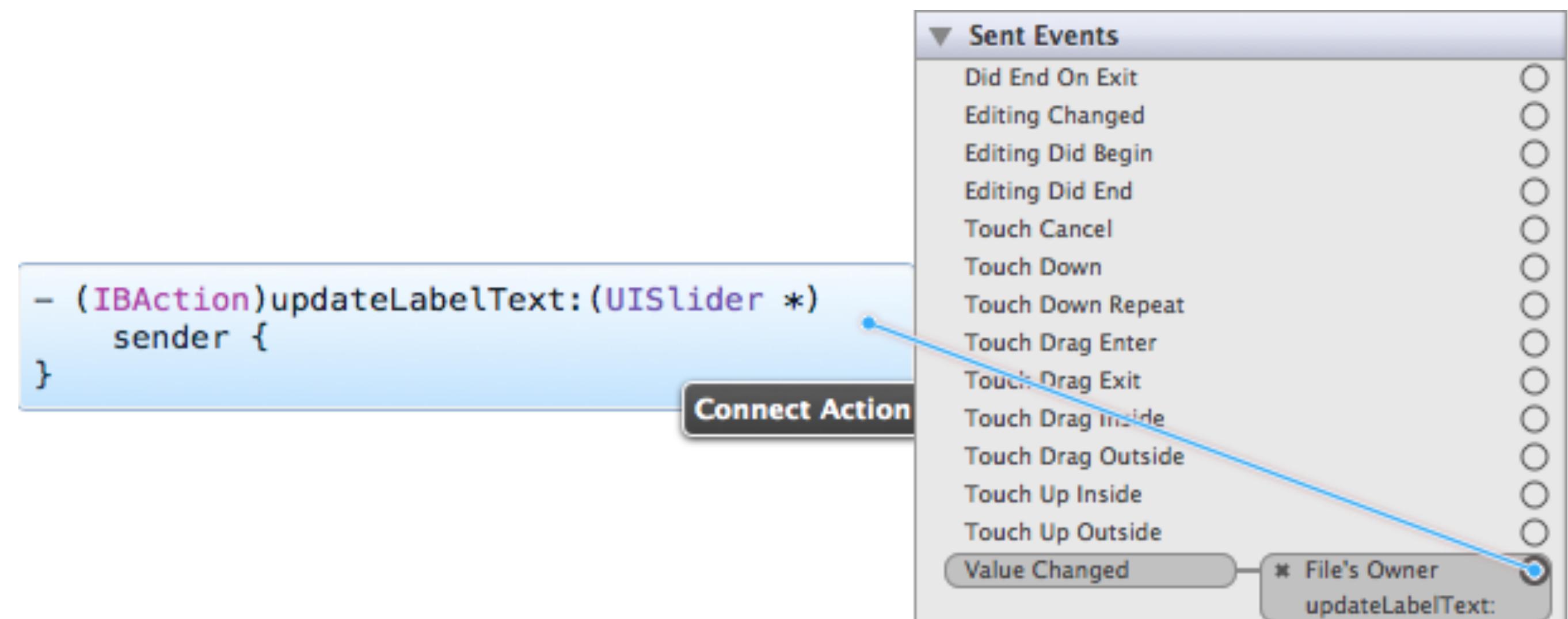


# Target-action binding

- Binding a target-action to a control event:
  1. programmatically

```
[self.mySlider addTarget:self action:@selector(myAction:) forControlEvents:UIControlEventValueChanged];
```

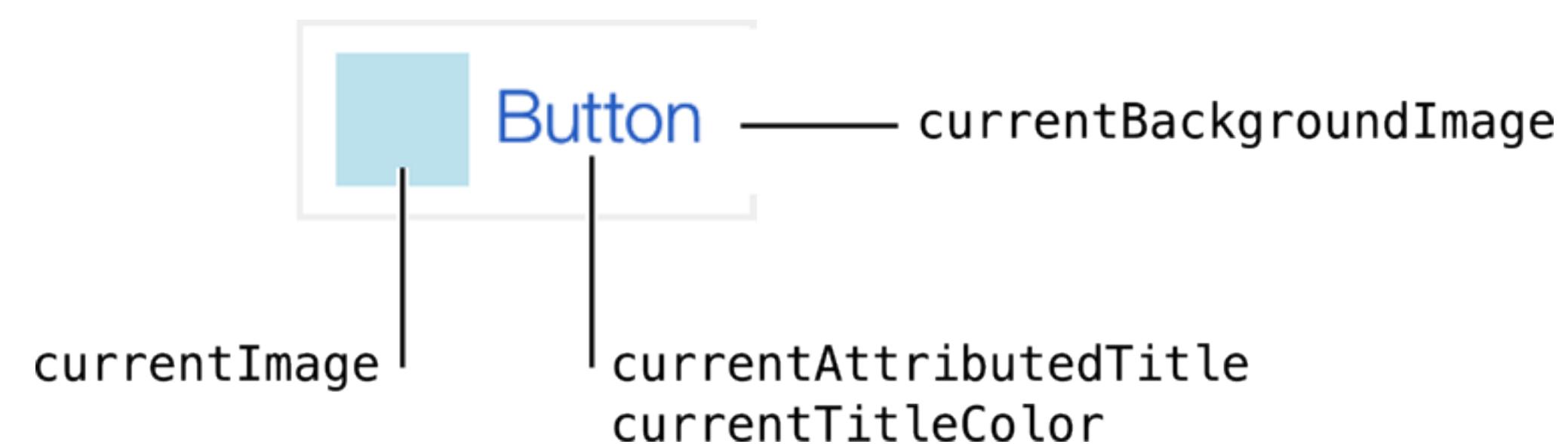
2. with the Connection Inspector in Interface Builder to Control-drag the slider's Value Changed event to the action method in the target file



3. Control-click the slider in Interface Builder, and drag its Value Changed event to the target object in your Storyboard and select the appropriate action from the list of actions available for the target

# UIButton

- Buttons let a user initiate behavior with a tap
- Buttons display textual or image content
- When the users taps on a button, it changes its state to highlighted and its appearance accordingly
- If an action has been bound for a certain event, a button will trigger the execution of that action each time the event occurs
- The current appearance of button can be retrieved with some read-only properties





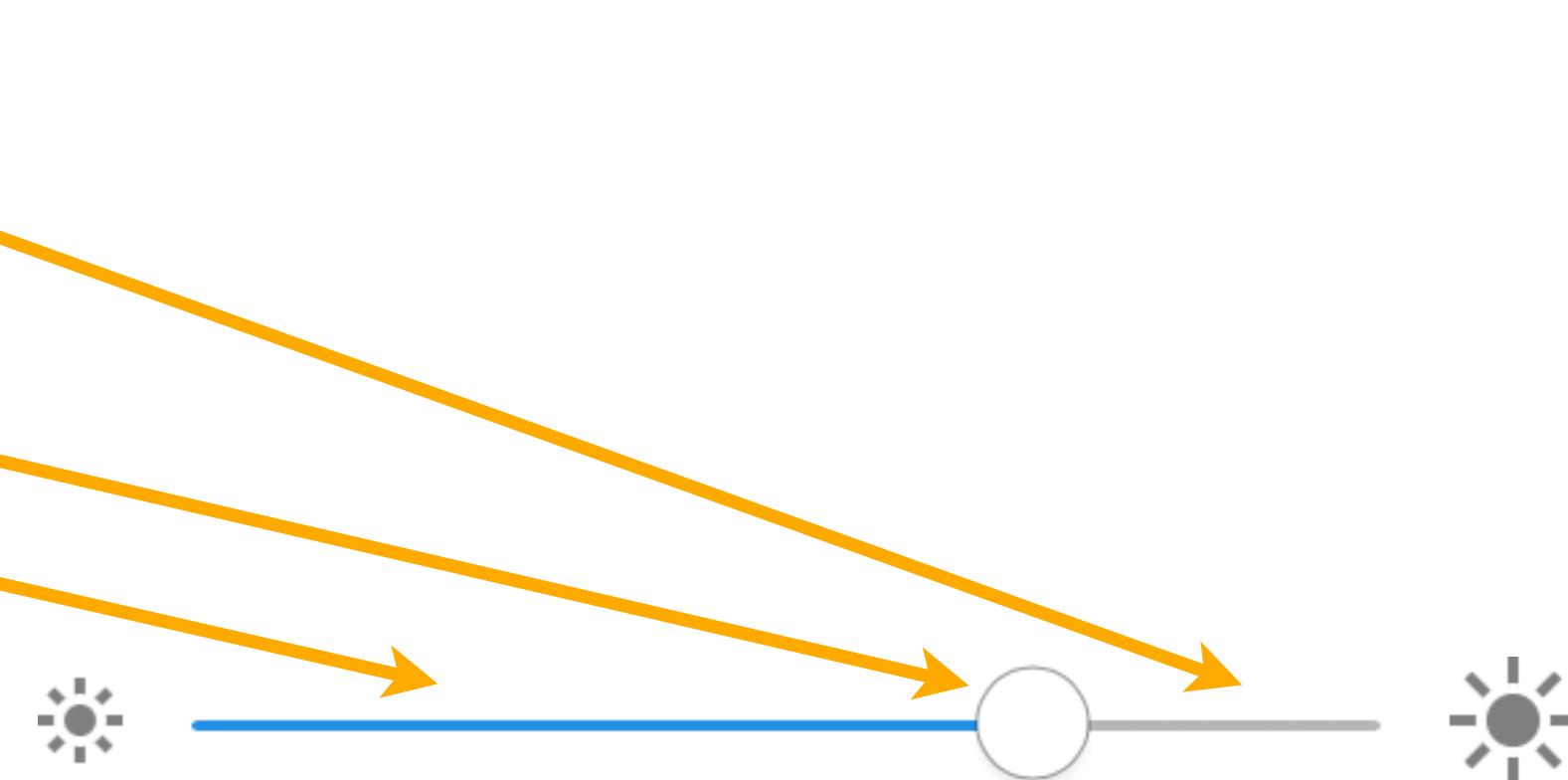
# UIButton

- It is possible to set the title, image, and background image of a button for each state
  1. in the attributes inspector
  2. programmatically
    - `(void)setTitle:(NSString *)title forState:(UIControlState)state`
    - `(void)setAttributedTitle:(NSAttributedString *)title forState:(UIControlState)state`
    - `(void)setImage:(UIImage *)image forState:(UIControlState)state`
    - `(void)setBackgroundImage:(UIImage *)image forState:(UIControlState)state`

# UISlider



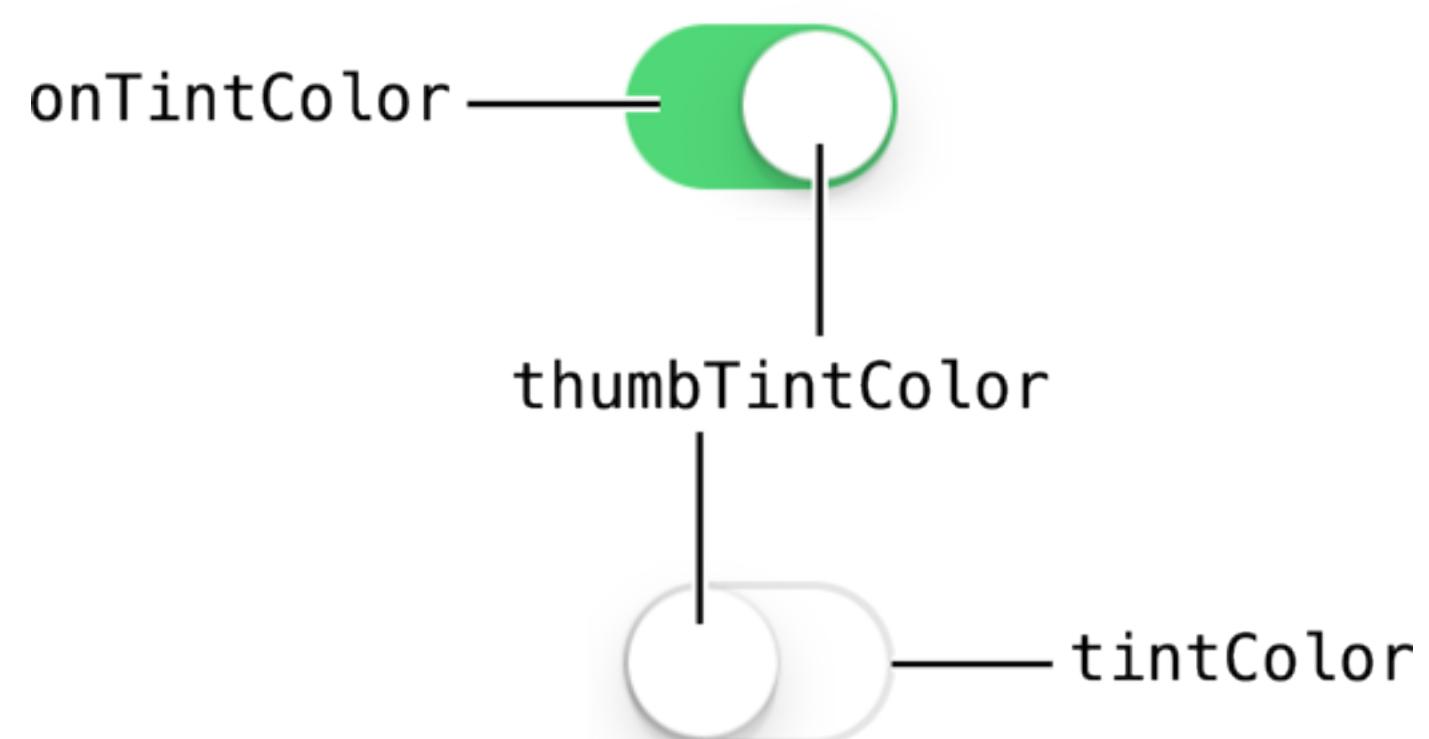
- Sliders enable users to interactively modify some adjustable value in an app
- It is possible to configure a minimum, maximum, and current value for your slider with the properties **minimumValue**, **maximumValue**, and **value**, respectively
- Default values are minimum = 0, maximum = 1, and current value = 0.5
- It is possible to change the tint of the slider with the properties
  - **maximumTrackTintColor**
  - **thumbTintColor**
  - **minimumTrackTintColor**



# UISwitch



- A switch lets the user turn an option on and off
- The appearance of a switch can be customized by setting the appropriate properties



- The boolean property **on** is used to set the off/on state of the switch

# UITextField



- Text fields allows the user to input a single line of text into an app
- You can set and retrieve the value of the input text with the **text** and **attributedText** properties
- The **placeholder** and **attributedPlaceholder** properties are used to set a placeholder text in the field
- Text can also be styled using **font**, **textAlignment**, **textColor** properties
- The clear button on the right is displayed by default; it is possible to configure whether it should be present or not by using the property **clearButtonMode** with a **UITextFieldViewMode**
- The keyboard style and layout can also be set (**UITextInputTraits**)





# Keyboard management

- Tapping a **UITextField** causes the keyboard to appear
- The keyboard slides in from the bottom of the screen and covers a certain area of the view
- The keyboard becomes the **first responder**
- It is necessary to handle the appearance of the keyboard properly:
  - it might be necessary to slide the view content up so that it does not get covered by the keyboard (more on this later)
  - it is necessary to make the keyboard disappear when done using it (tapping “done” won’t make it go away)



# Keyboard management

- The trick is to set the background view's class to be **UIControl** instead of **UIView** so that it can receive tap events
- Define a **IBAction** in the view controller to be invoked when the user “touches up inside” the control
- When the background view is tapped, a target-action method is invoked and that's where the keyboard can be dismissed by using:

```
[textField resignFirstResponder];
```



# UITextView

- More powerful way to display text:
  - multiple lines, editable and selectable, scrollable
  - Set the text through two properties:
    - **text: NSString** for normal text; properties that can affect the text style are **font (UIFont)**, **textColor (UIColor)**, and **textAlignment (NSTextAlignment)**
    - **attributedText: NSAttributedString** for styled text
  - Efficient way to manipulate text:
    - **textStorage: NSTextStorage** allows you to change the styled text and it will automatically update the view (since iOS7)

# UITextView

- UITextView has many following notable properties:

| <b>Property</b> | <b>Value type</b>   | <b>Description</b>                                                                          |
|-----------------|---------------------|---------------------------------------------------------------------------------------------|
| text            | NSString*           | Text being displayed                                                                        |
| font            | UIFont*             | Font of the text                                                                            |
| textColor       | UIColor*            | Color of the text                                                                           |
| textAlignment   | NSTextAlignment     | Alignment of the text (NSTextAlignmentLeft, NSTextAlignmentRight, NSTextAlignmentCenter...) |
| attributedText  | NSAttributedString* | Styled text being displayed                                                                 |
| textStorage     | NSTextStorage       | Efficient text manipulation                                                                 |
| editable        | BOOL                | Whether the receiver is editable                                                            |
| selectable      | BOOL                | Whether the receiver is selectable                                                          |
| selectedRange   | NSRange             | Current selection range in the text view                                                    |



# UITextViewDelegate

- **UITextViewDelegate** is a protocol that defines a set of optional methods can be used to receive editing-related messages for a certain **UITextView**
- It is possible to set a delegate for a **UITextView** object with the **delegate** property
- Protocol methods:
  - **(BOOL)textViewShouldBeginEditing:(UITextView \*)textView**
  - **(BOOL)textViewShouldEndEditing:(UITextView \*)textView**
  - **(void)textViewDidBeginEditing:(UITextView \*)textView**
  - **(void)textViewDidChange:(UITextView \*)textView**
  - **(void)textViewDidChangeSelection:(UITextView \*)textView**
  - **(void)textViewDidEndEditing:(UITextView \*)textView**



# Notifications

- iOS provides another way for object interaction, other than message passing
- Notifications are the standard way by which the model can notify the controller of certain event
- **NSNotificationCenter** is a class that provides a mechanism for broadcasting information within a program
- A reference to a **NSNotificationCenter** instance is retrieved in the following way:

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
```

- Each program has its own **NSNotificationCenter**, so it does not need to be created
- Objects can register with a notification center to receive notifications and upon receiving such notification can execute a particular method
- It is important to unregister for notifications when no longer needed to avoid possible crashes
- Typically, register when view appears and unregister when view disappears; if notifications should be received when the view is off screen, unregister in **dealloc** (rare)

# Registering for Notifications

- Objects can register with the **NSNotificationCenter** for certain notifications with the method:

```
- (void)addObserver:(id)notificationObserver
 selector:(SEL)notificationSelector
 name:(NSString *)notificationName
 object:(id)notificationSender
```

object that is listening for  
notifications





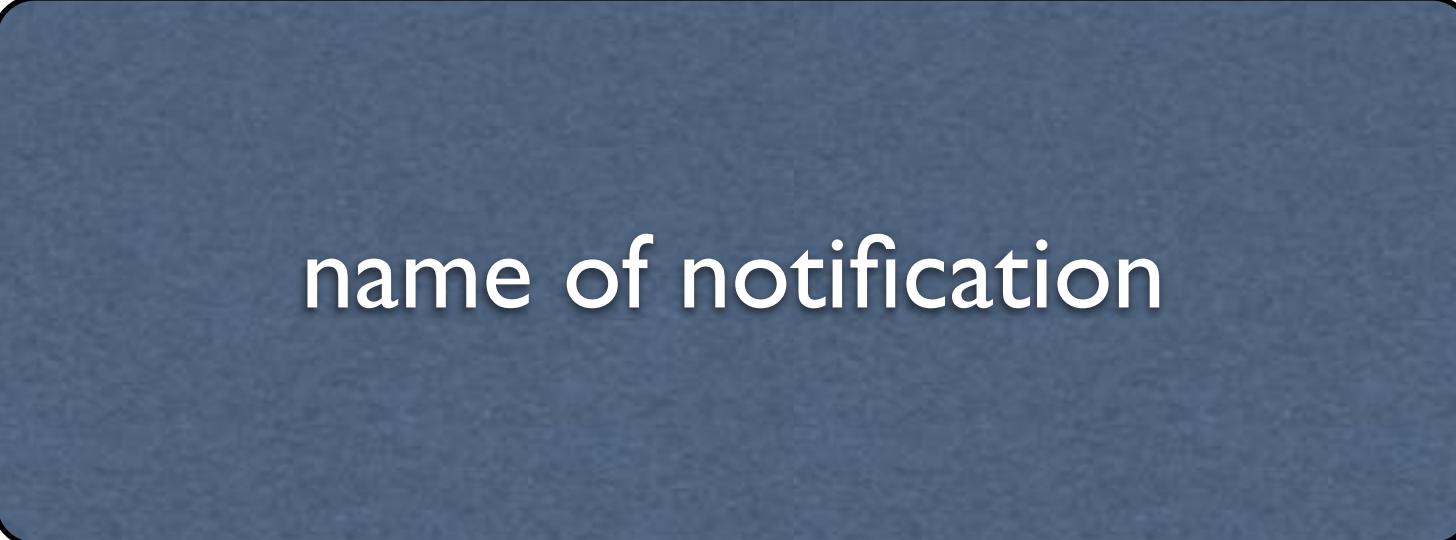
# Registering for Notifications

- Objects can register with the **NSNotificationCenter** for certain notifications with the method:
- `(void)addObserver:(id)notificationObserver  
selector:(SEL)notificationSelector ←  
name:(NSString *)notificationName  
object:(id)notificationSender`

selector to execute when  
receiving the notification

# Registering for Notifications

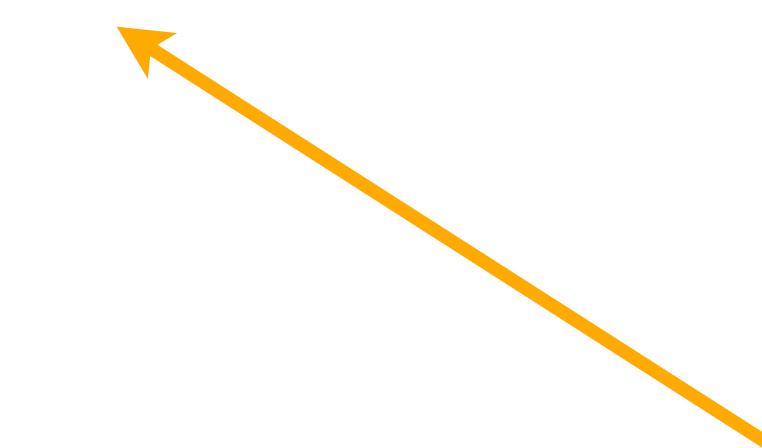
- Objects can register with the **NSNotificationCenter** for certain notifications with the method:
- **(void)addObserver:(id)notificationObserver  
selector:(SEL)notificationSelector  
name:(NSString \*)notificationName  
object:(id)notificationSender**



name of notification

# Registering for Notifications

- Objects can register with the **NSNotificationCenter** for certain notifications with the method:
- **(void)addObserver:(id)notificationObserver  
selector:(SEL)notificationSelector  
name:(NSString \*)notificationName  
object:(id)notificationSender**



source of notifications (nil if any object)



# Registering for Notifications

- For instance:

```
[[NSNotificationCenter defaultCenter] addObserver:self
 selector:@selector(aMethod)
 name:UIKeyboardWillShowNotification
 object:nil];
```



# Unregistering for Notifications

- When an objects is no longer interested in notifications, it can unregister with the **NSNotificationCenter** with the method:
  - **(void)removeObserver:(id)notificationObserver  
name:(NSString \*)notificationName  
object:(id)notificationSender**
- For instance:

```
[[NSNotificationCenter defaultCenter] removeObserver:self
 name:UIKeyboardWillShowNotification
 object:nil];
```



# Generating Notifications

- An object that needs to generate a notification can post the notification to the `NSNotificationCenter` with the method:
  - `(void)postNotificationName:(NSString *)notificationName  
object:(id)notificationSender  
userInfo:(NSDictionary *)userInfo`
- For instance:

```
[[NSNotificationCenter defaultCenter] postNotificationName:@"MyNotification"
object:self
userInfo:nil];
```



# Keyboard notifications

- When the system shows or hides the keyboard, it posts several keyboard notifications
- Notifications contain information about the keyboard, such as its size
- Registering for these notifications is the only way to get some types of information about the keyboard
- System notifications for keyboard-related events (names are similar to delegate methods, but these are notifications, which implies a different communication paradigm):
  - **UIKeyboardWillShowNotification**
  - **UIKeyboardDidShowNotification**
  - **UIKeyboardWillHideNotification**
  - **UIKeyboardDidHideNotification**
- The selectors related to these events should be responsible to move the view to make the content visible when the view keyboard appears, and to reposition it when it disappears



# Mobile Application Development



Lecture 4  
UIKit views and controls



This work is licensed under a [Creative Commons Attribution-  
NonCommercial-ShareAlike 4.0 International License](#).



# Mobile Application Development



Lecture 5  
Controllers of View Controllers



This work is licensed under a [Creative Commons Attribution-  
NonCommercial-ShareAlike 4.0 International License](#).

# Lecture Summary

- Multiple MVCs
- UINavigationController
- Segues
- UITabBarController





# Multiple MVCs

- Complex iOS applications typically rely on multiple views to show content
- For instance, it might not be possible to display all the content inside a single view, so another view might be needed to display additional content
- Multiple views also make the flow of the application more natural and intuitive for the user
- Handling multiple views means that we also have multiple view controllers that must coordinate the flow of the application
- Each view is controlled by its own view controller, thus multiple MVCs come into play in an application

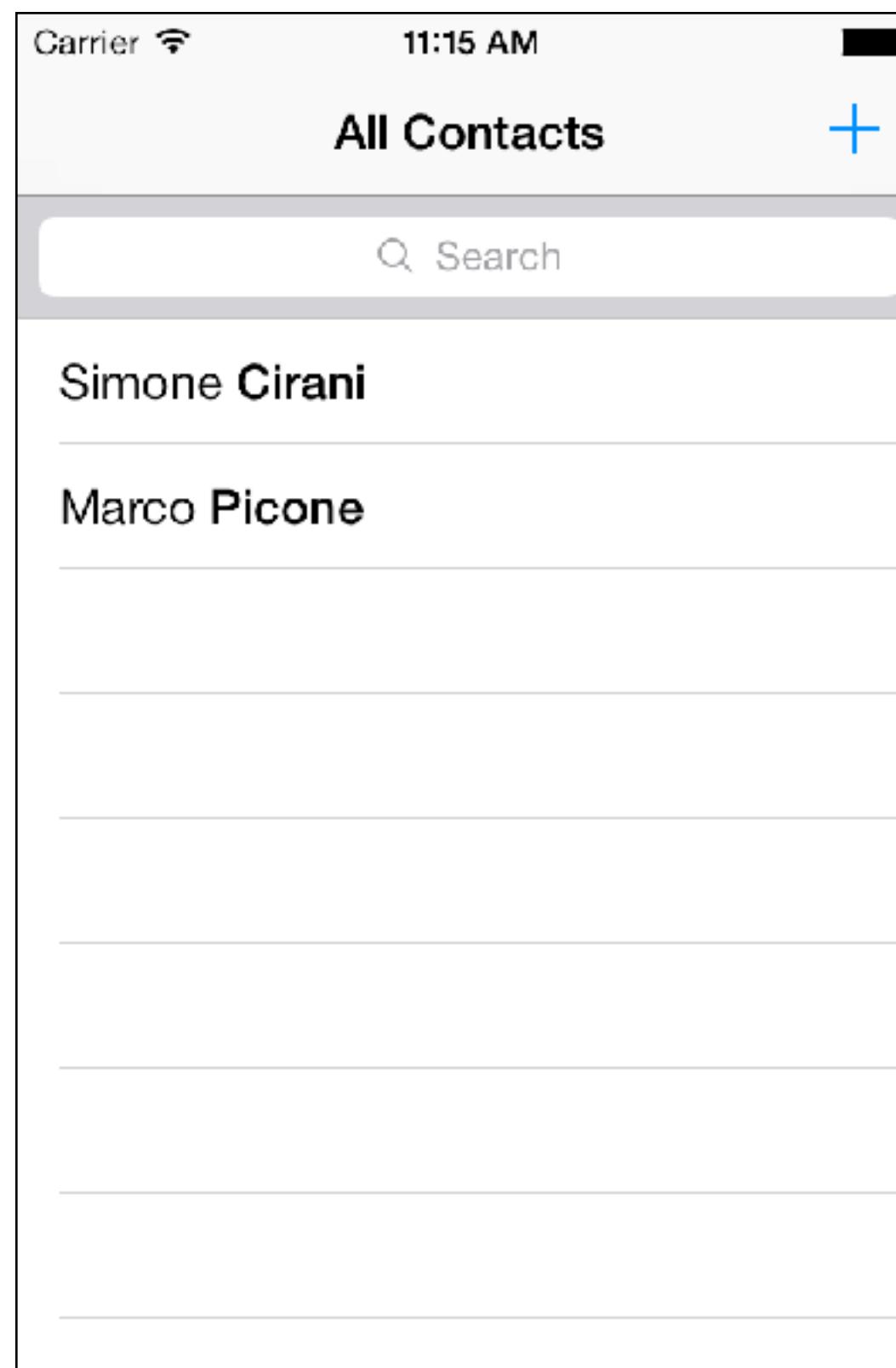


# Multiple MVCs

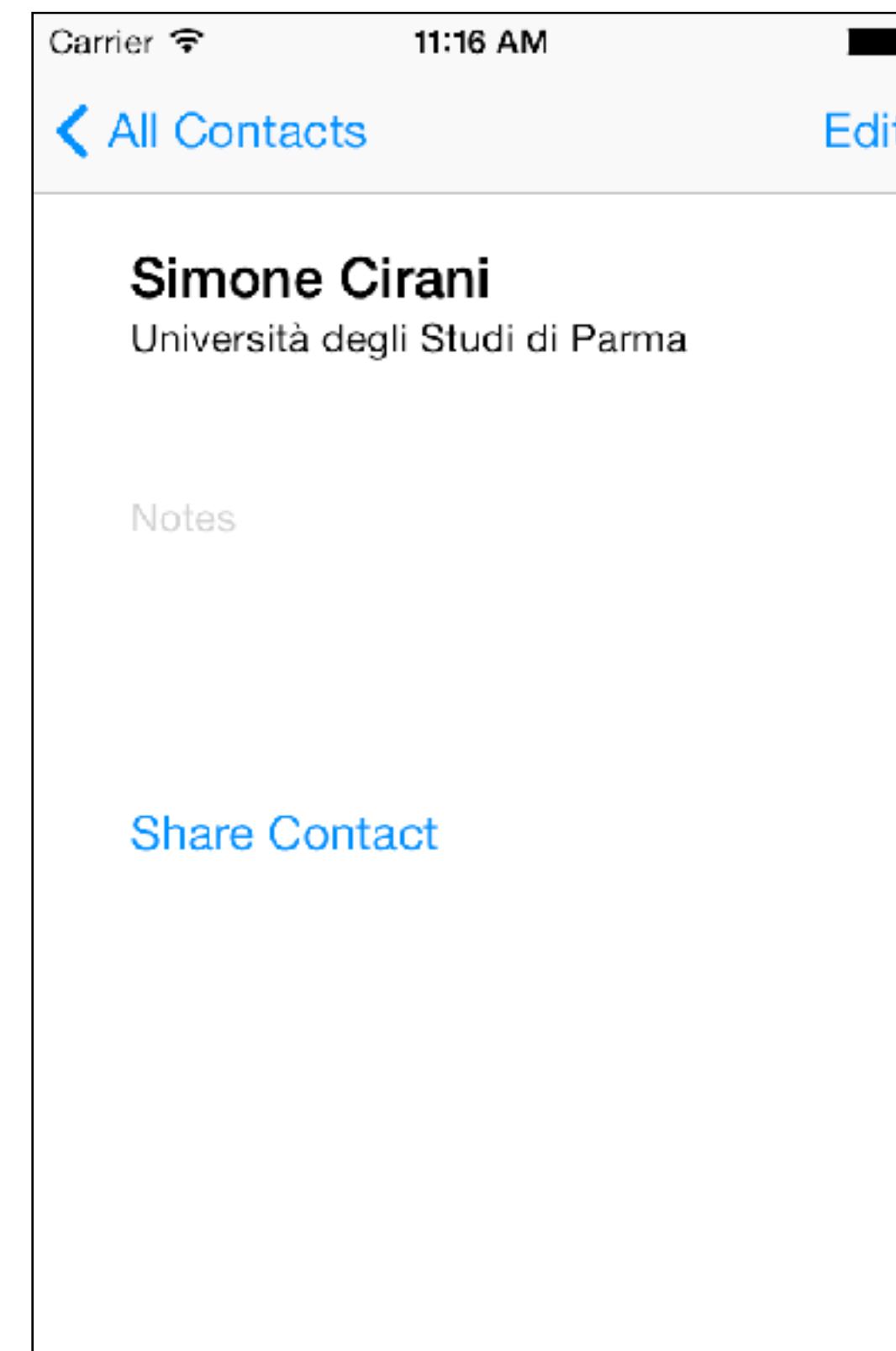
- As in good OOP design, view controllers should be highly specialized and independent from each other
- Each view controller should be responsible to manage a view that display some specific content (e.g. a contact list, a contact details)
- Apple provides great examples of apps with multiple MVCs:
  - Contacts app
  - Calendar app
  - Music app



# Multiple MVCs: Contact app



Contact list



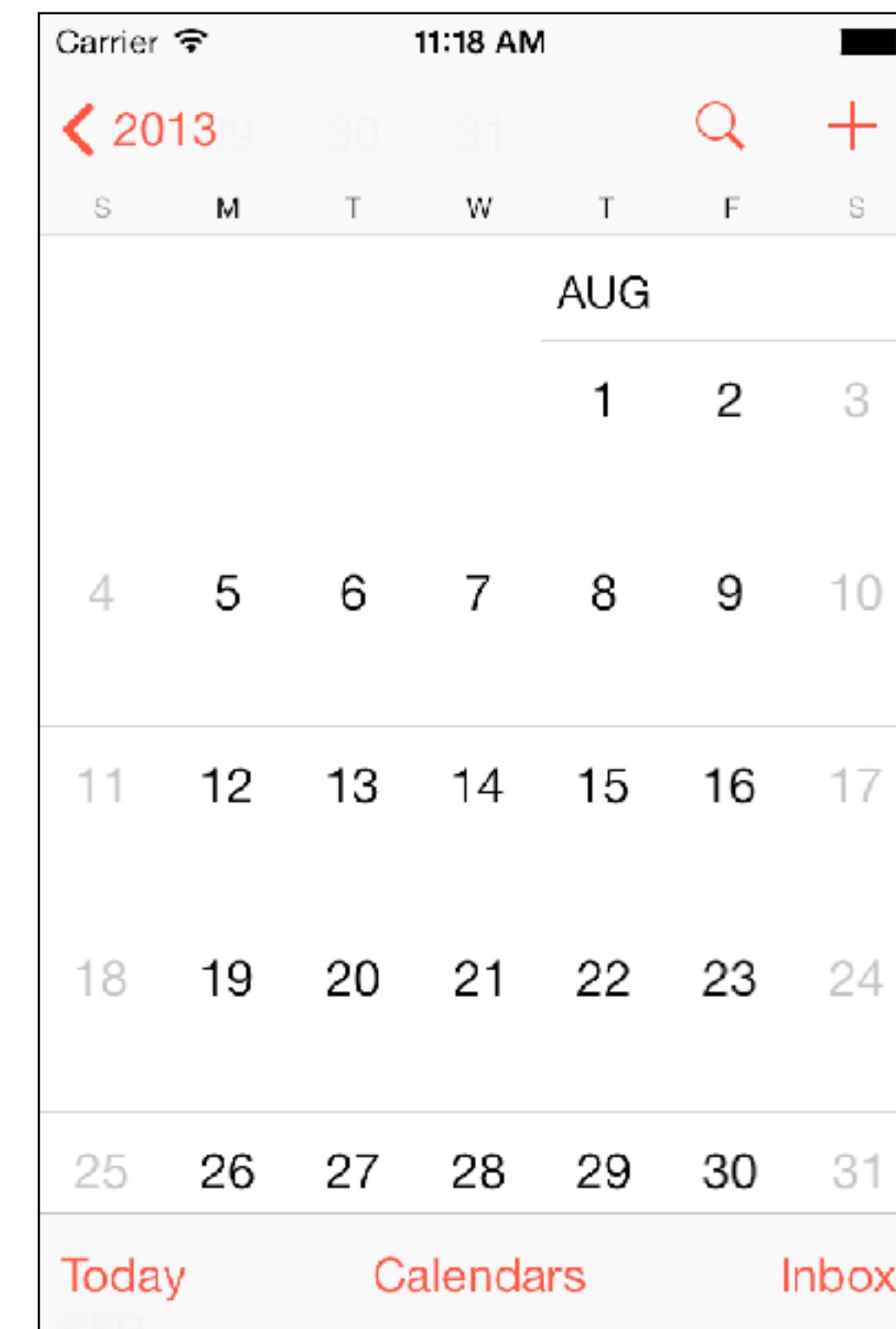
Contact details



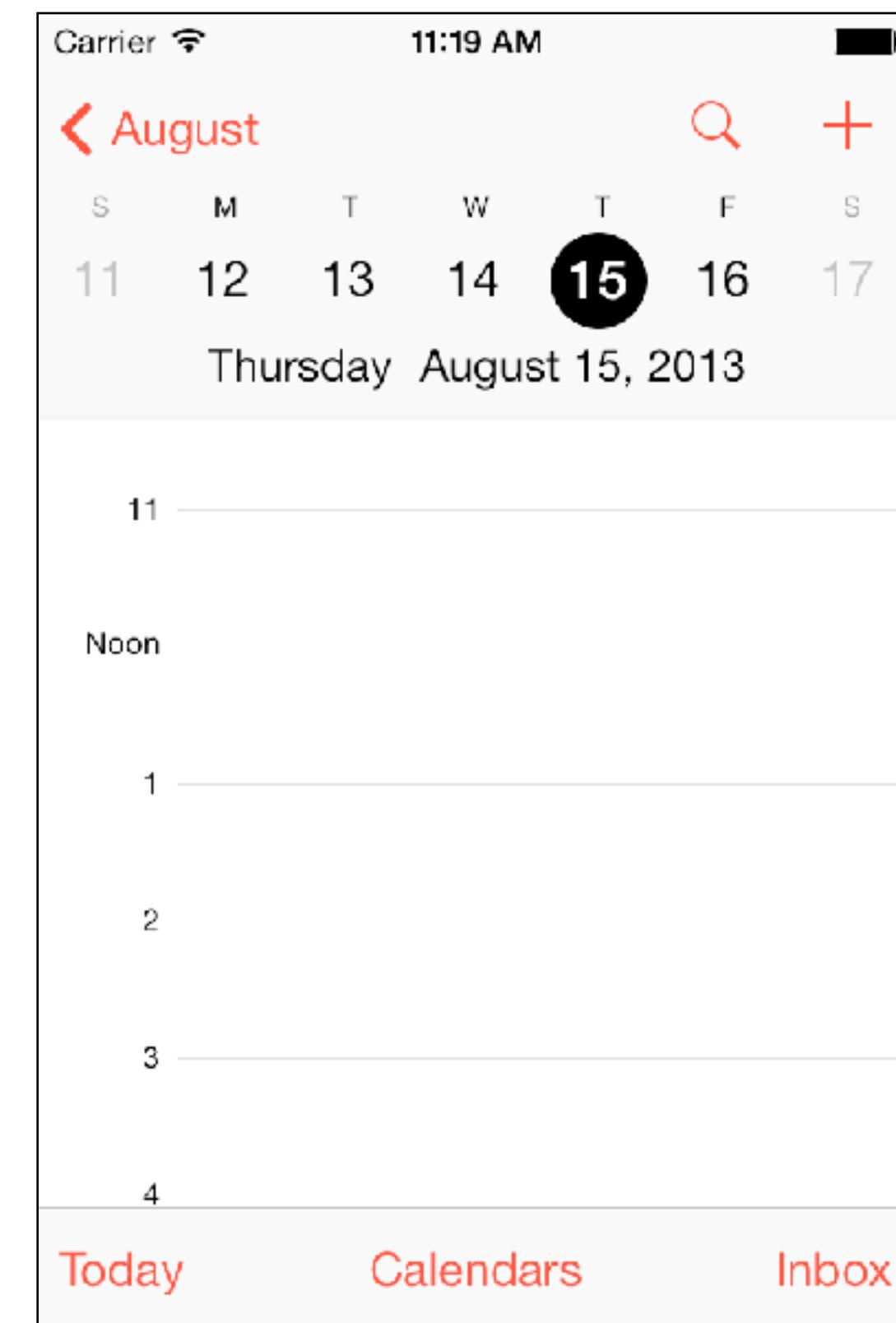
# Multiple MVCs: Calendar app



Year view



Month view



Day view



# Multiple MVCs

- Multiple view controllers can be rendered in different ways depending on what content is going to displayed throughout the app
- The management of the transition among different MVCs is provided by dedicated controllers
- Controllers of view controllers:
  - **UINavigationController**
  - **UITabBarController**

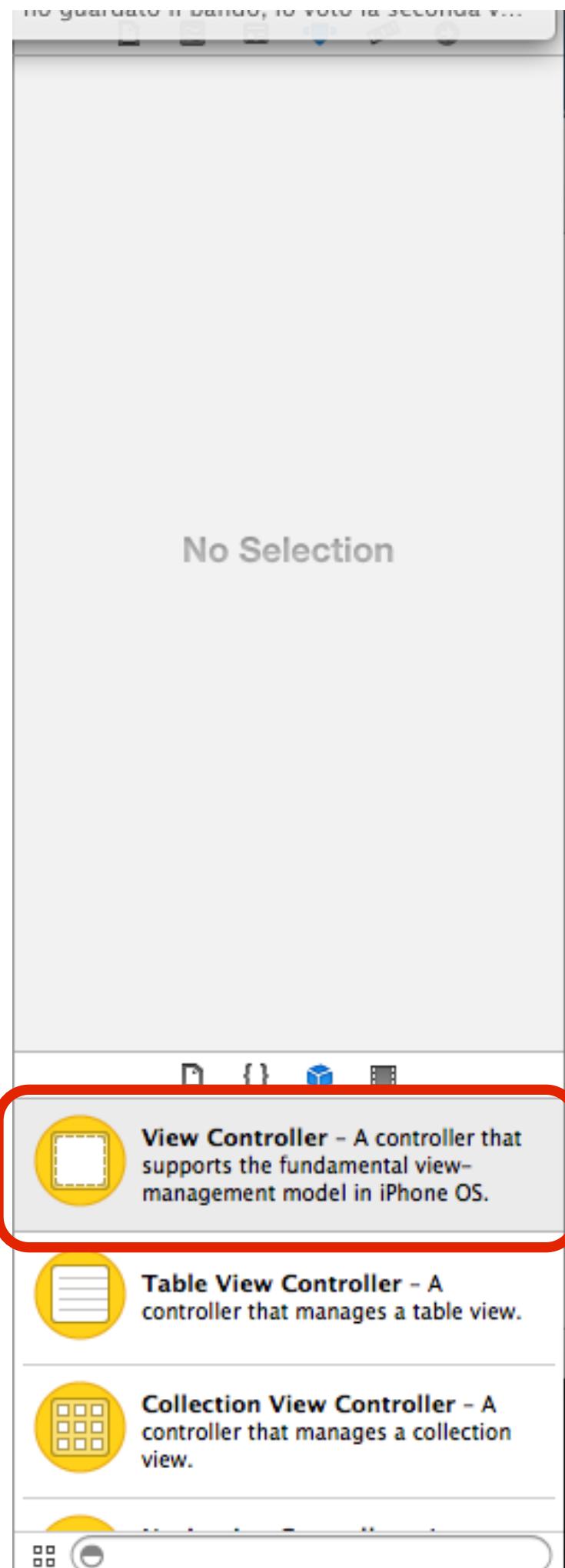


# Transitions between MVC: Segues

- Transitions among MVCs are called **segues**
- Segues can be triggered by controls in the view or by certain events that might occur in the app, depending on the application logic

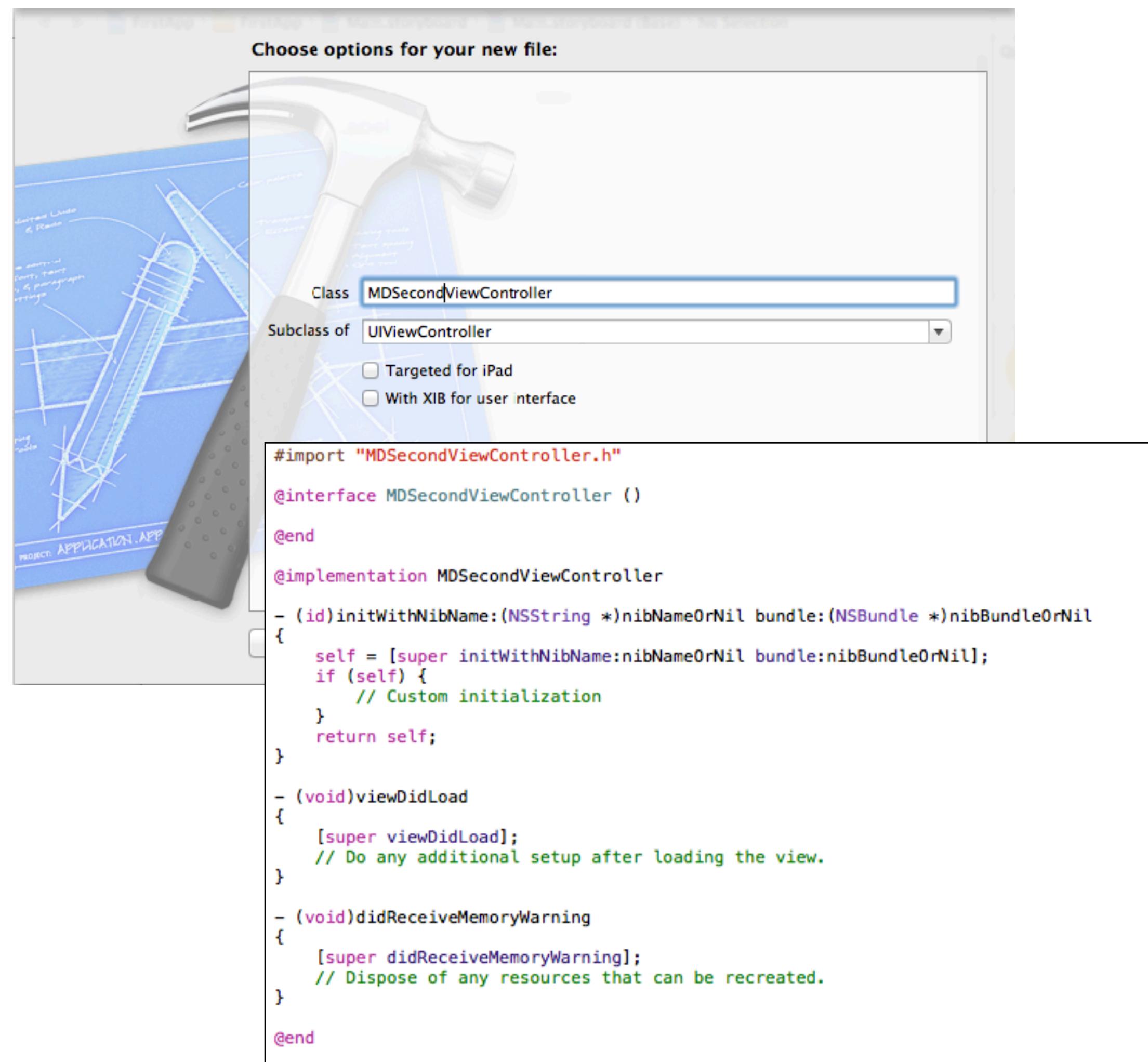
# Adding a view controller

- Other view controllers are added to the storyboard:
  1. drag a UIViewController from the object palette to the storyboard
  2. create a subclass of UIViewController using New File menu item
  3. in the **Identity** inspector, set the class of the UIViewController to the newly created subclass



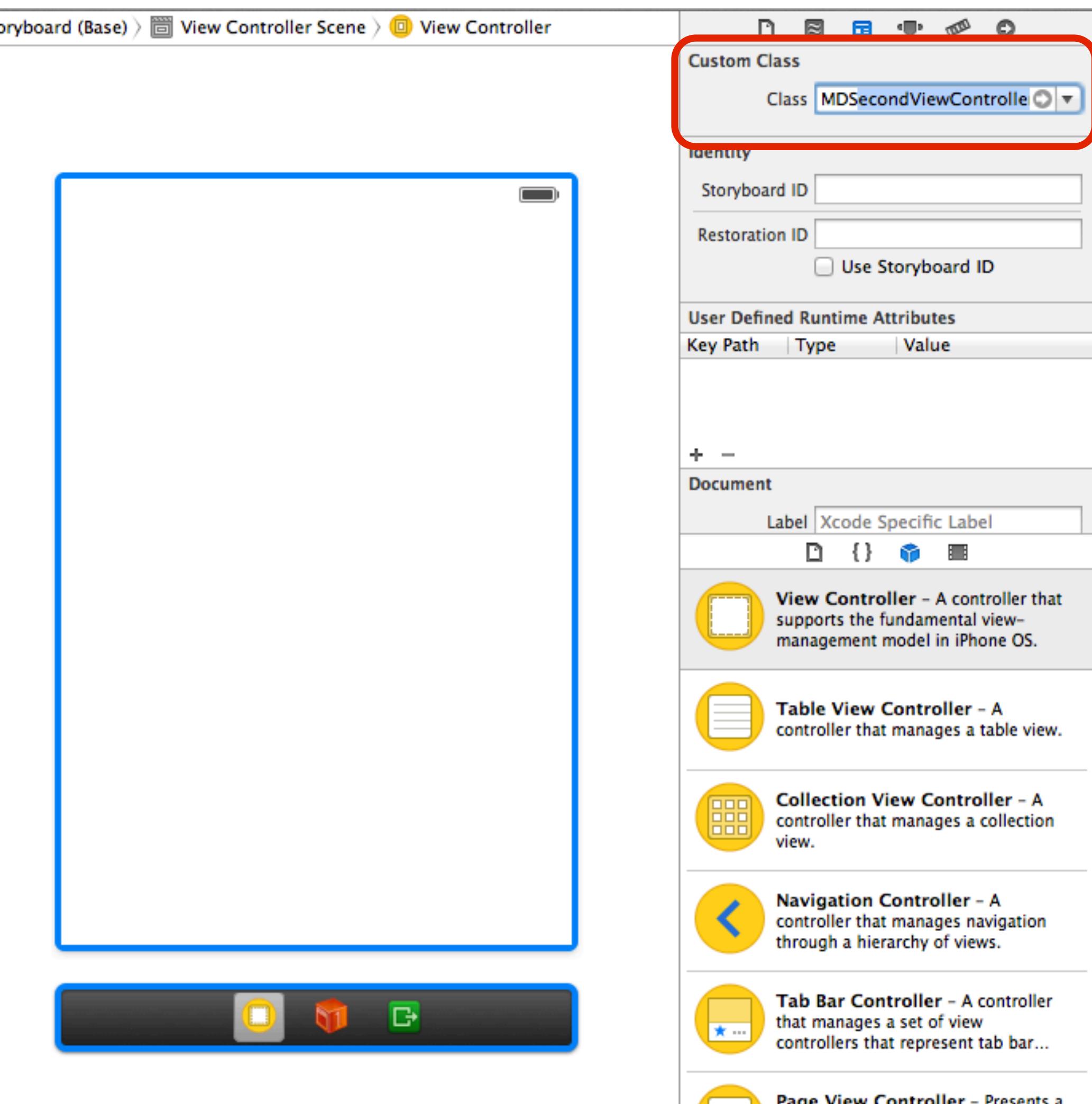
# Adding a view controller

- Other view controllers are added to the storyboard:
  1. drag a UIViewController from the object palette to the storyboard
  2. create a subclass of UIViewController using New File menu item
  3. in the **Identity** inspector, set the class of the UIViewController to the newly created subclass



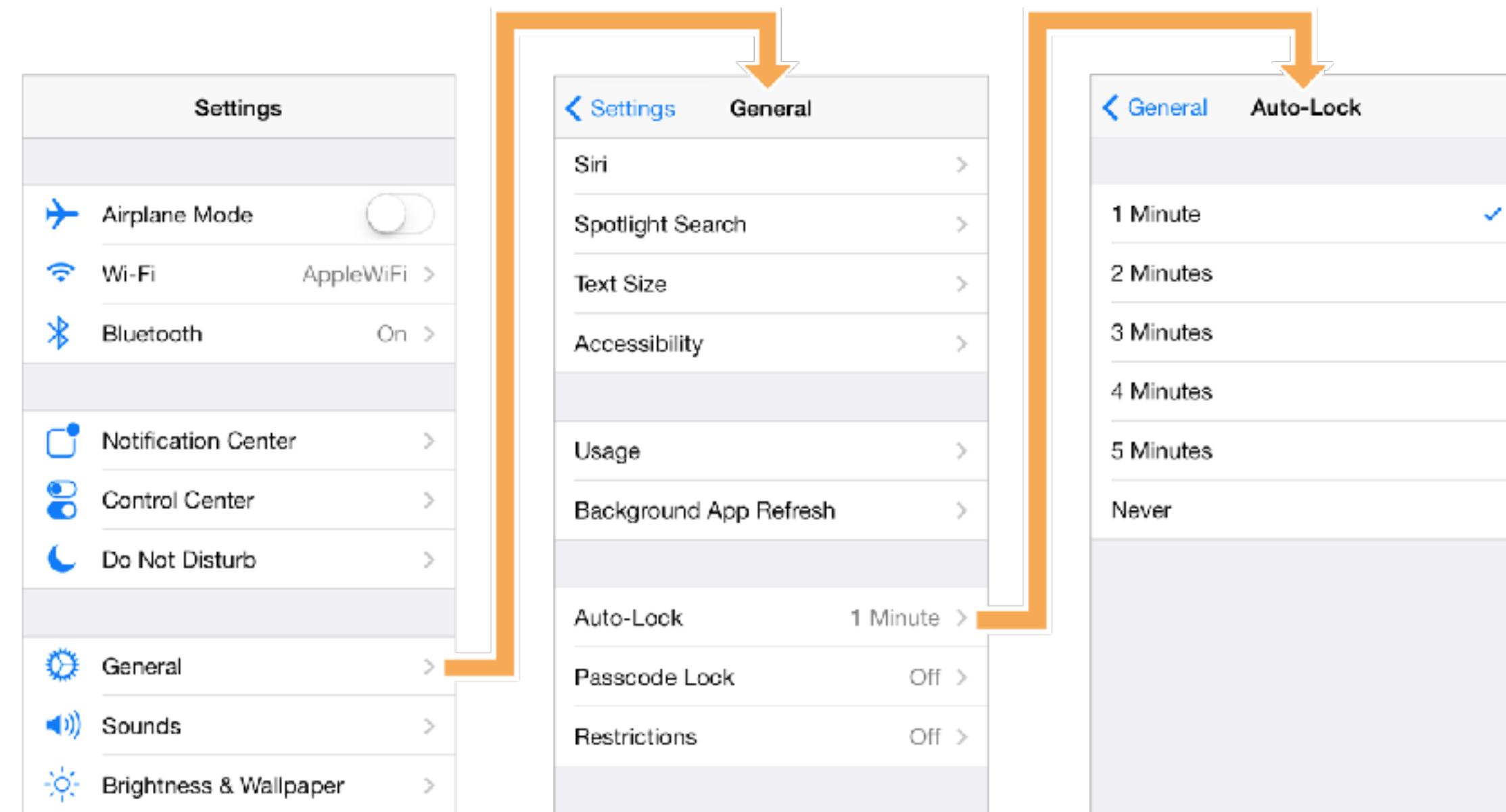
# Adding a view controller

- Other view controllers are added to the storyboard:
  1. drag a UIViewController from the object palette to the storyboard
  2. create a subclass of UIViewController using New File menu item
  3. in the **Identity** inspector, set the class of the UIViewController to the newly created subclass



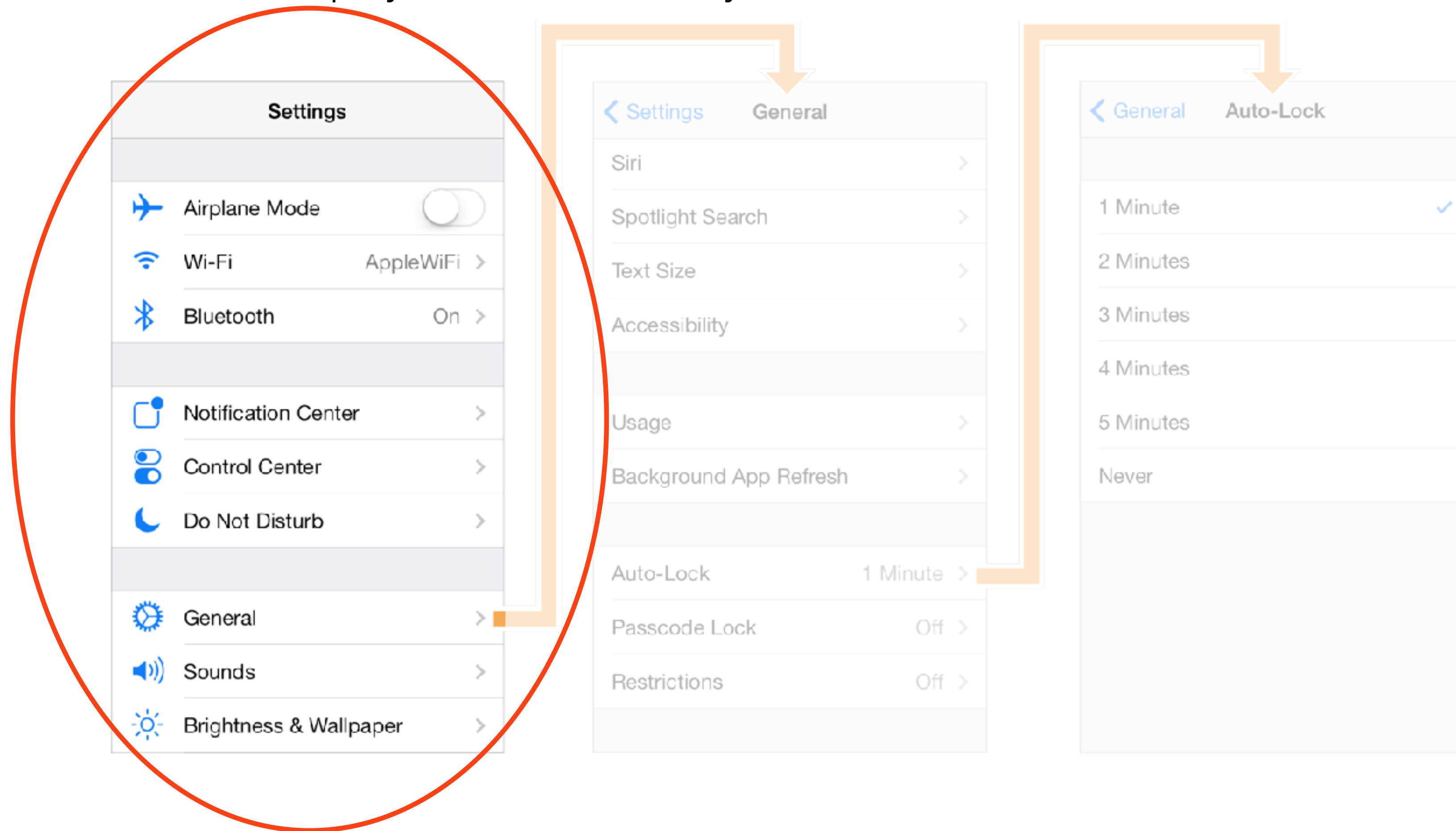
# UINavigationController

- UINavigationController is a class that implements a specialized view controller that manages the navigation of hierarchical content
- Typical usage: **drill-down to detailed content** (e.g. Calendar app: year → month → day)
- The screens presented by a navigation interface typically mimic the hierarchical organization of data
- Example of a navigation controller-based app: Settings app



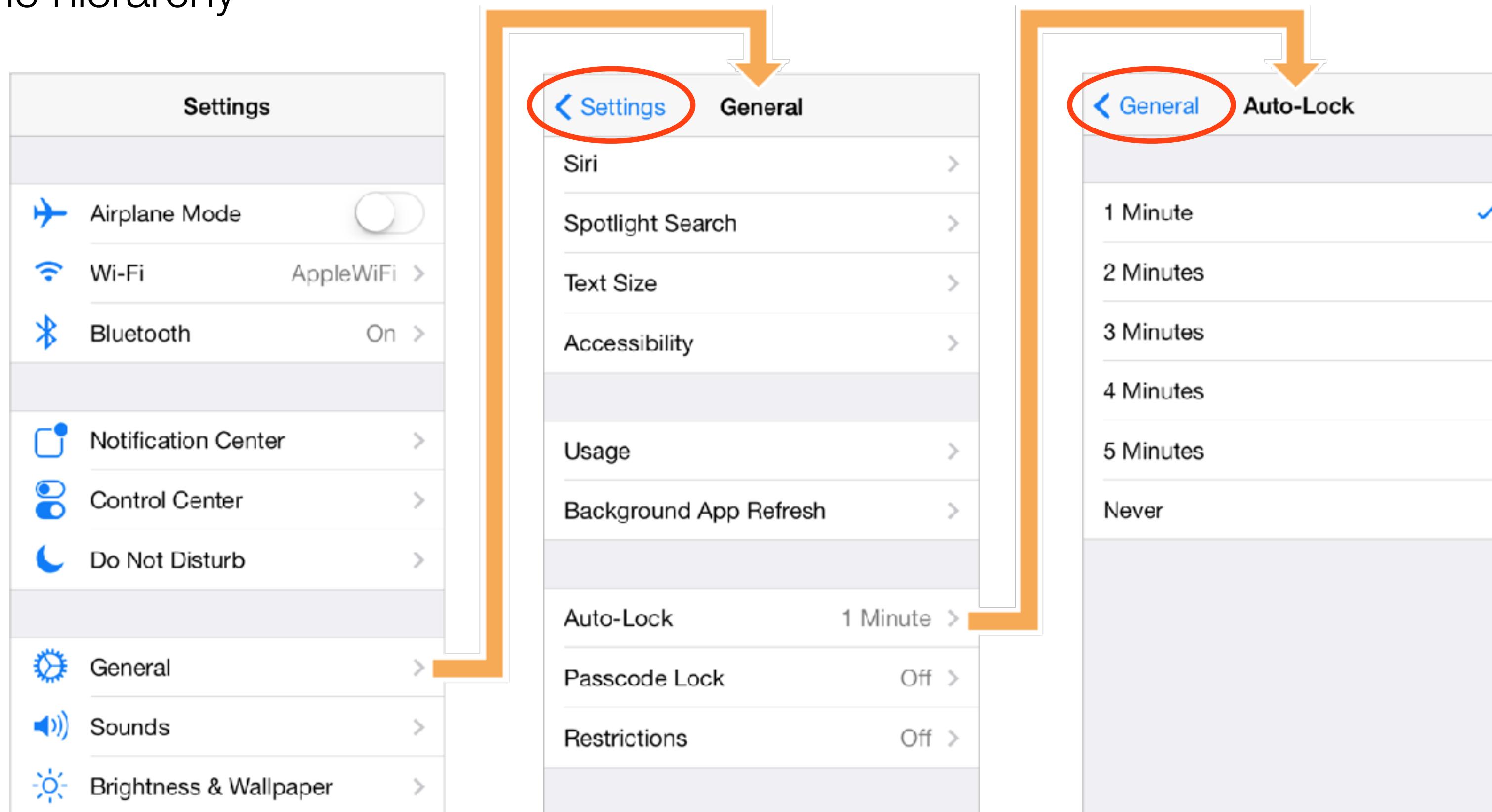
# UINavigationController

- The first view to be displayed in the hierarchy is called **root view controller**



# UINavigationController

- For all but the root view, the navigation controller provides a back button to allow the user to move back up the hierarchy

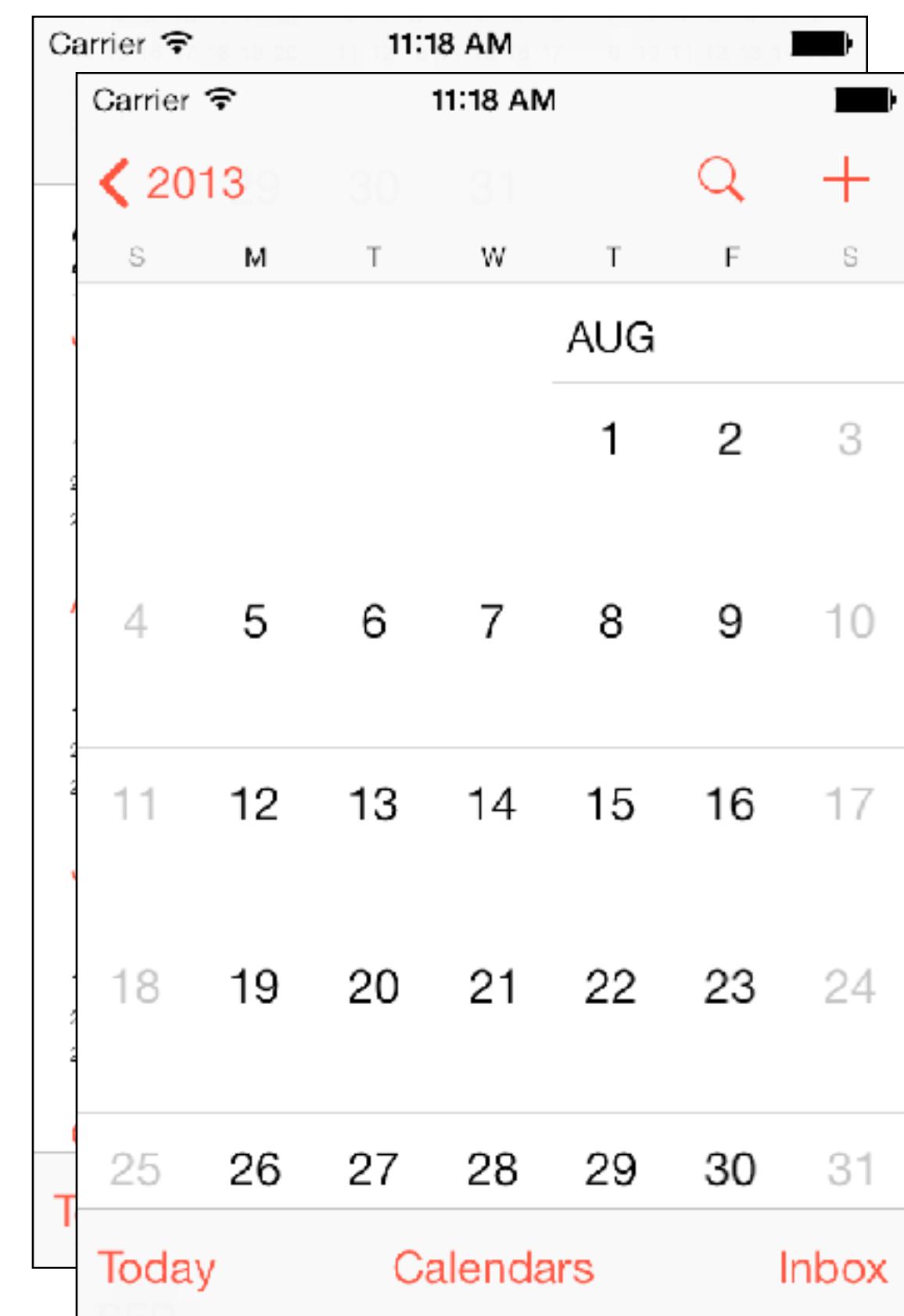
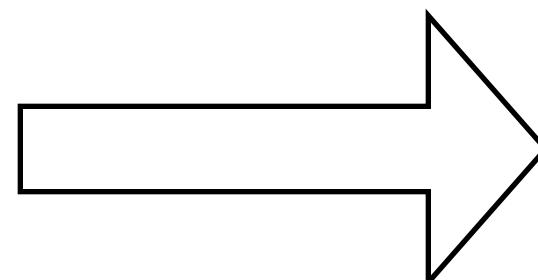


# UINavigationController

- The navigation controller manages a **stack of view controllers**

The year view is the root view controller

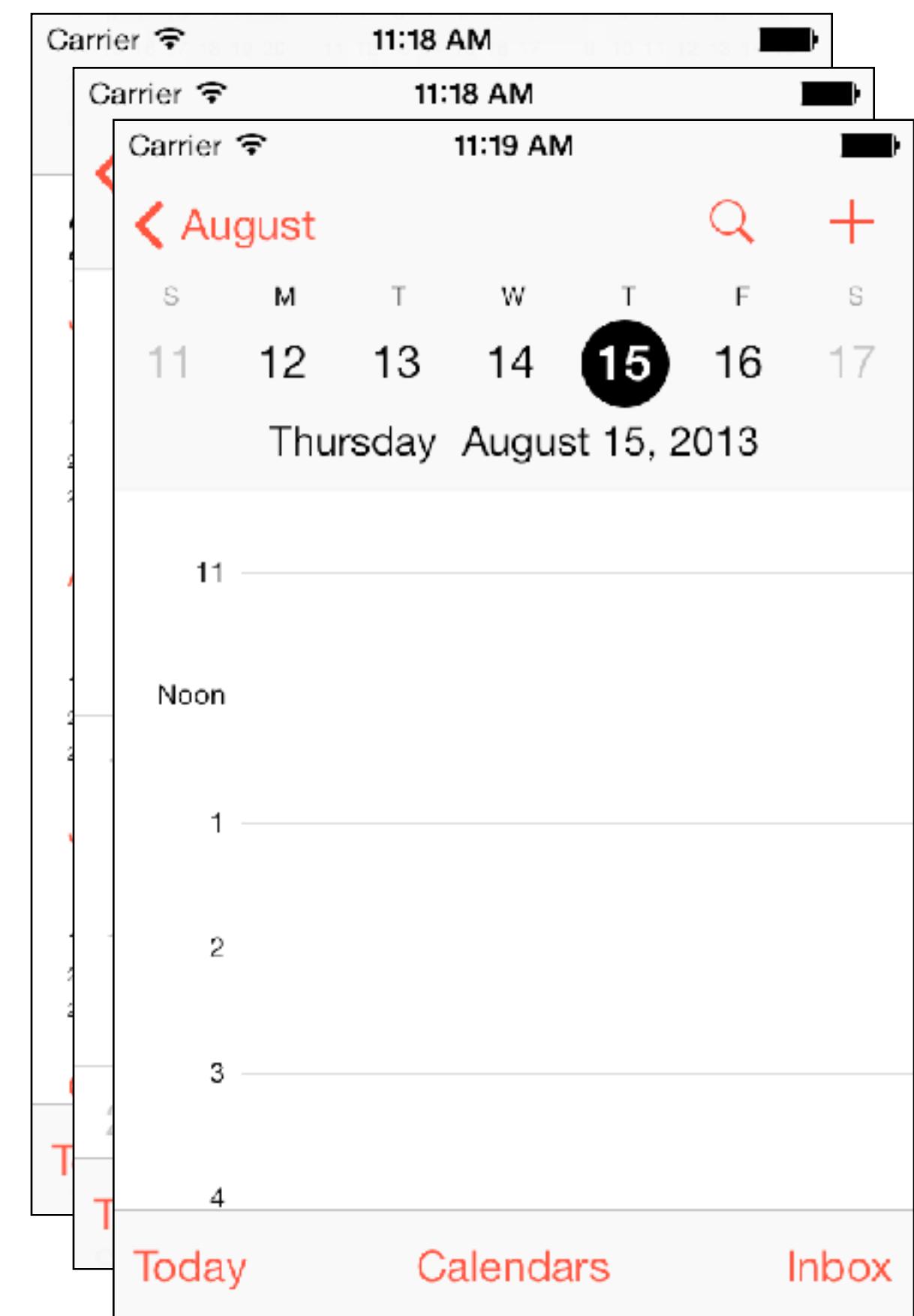
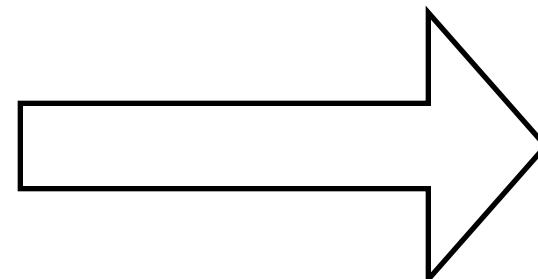
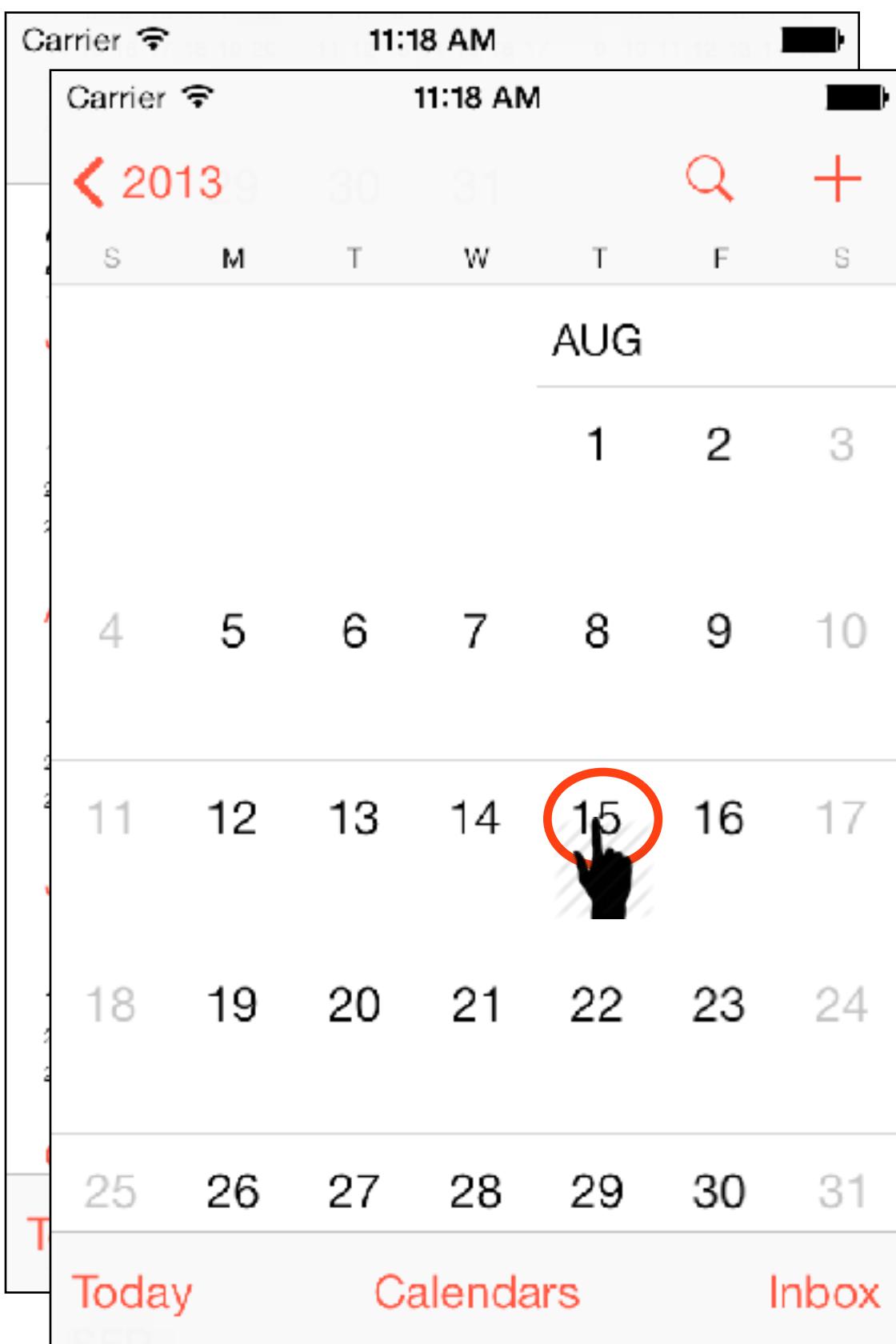
When a month is tapped, the month view is put at the top of the stack



# UINavigationController

- The navigation controller manages a **stack of view controllers**

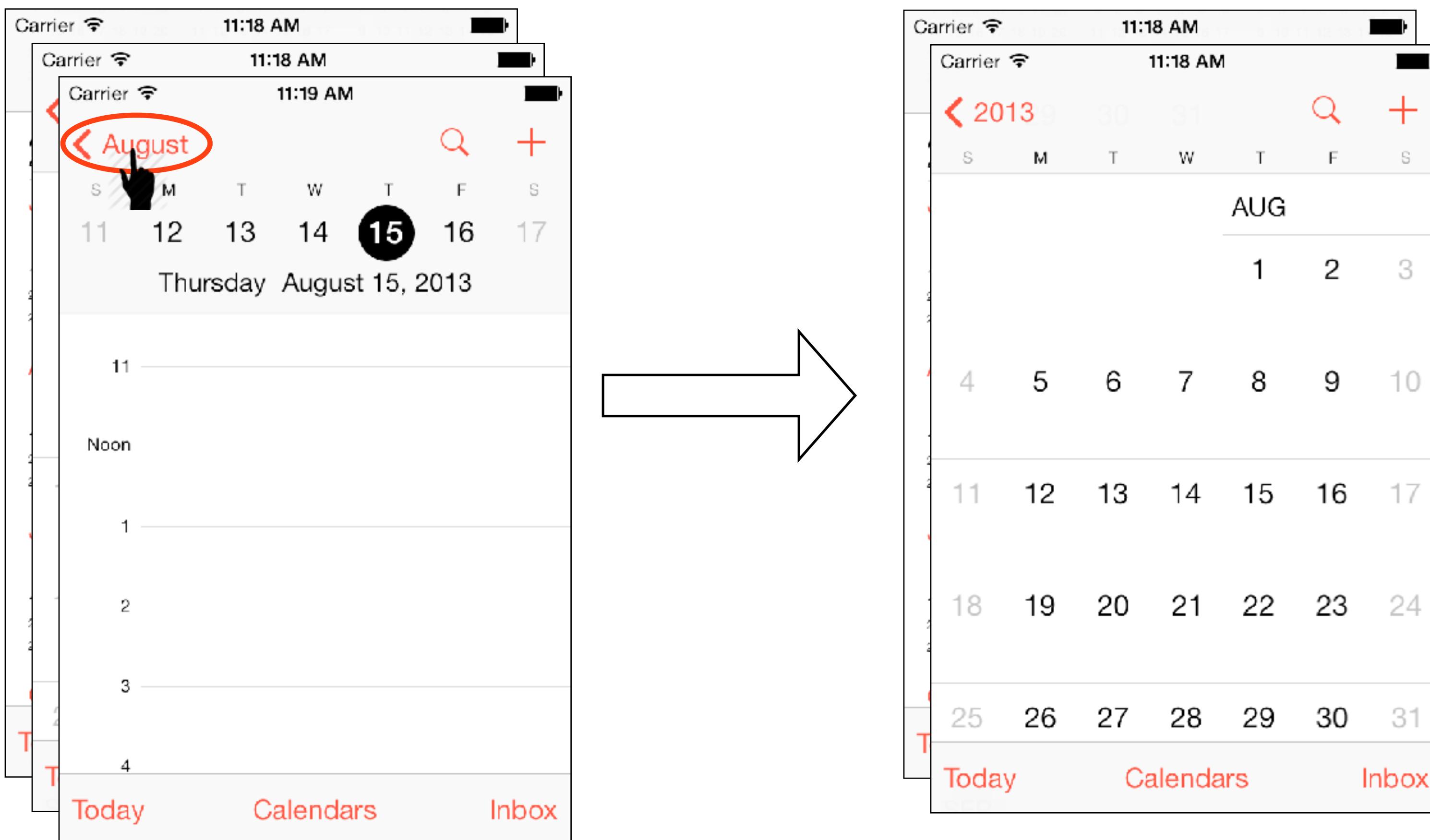
When a day is tapped, the day view is put at the top of the stack



# UINavigationController

- The navigation controller manages a **stack of view controllers**

Tapping the back button pops the view from the stack



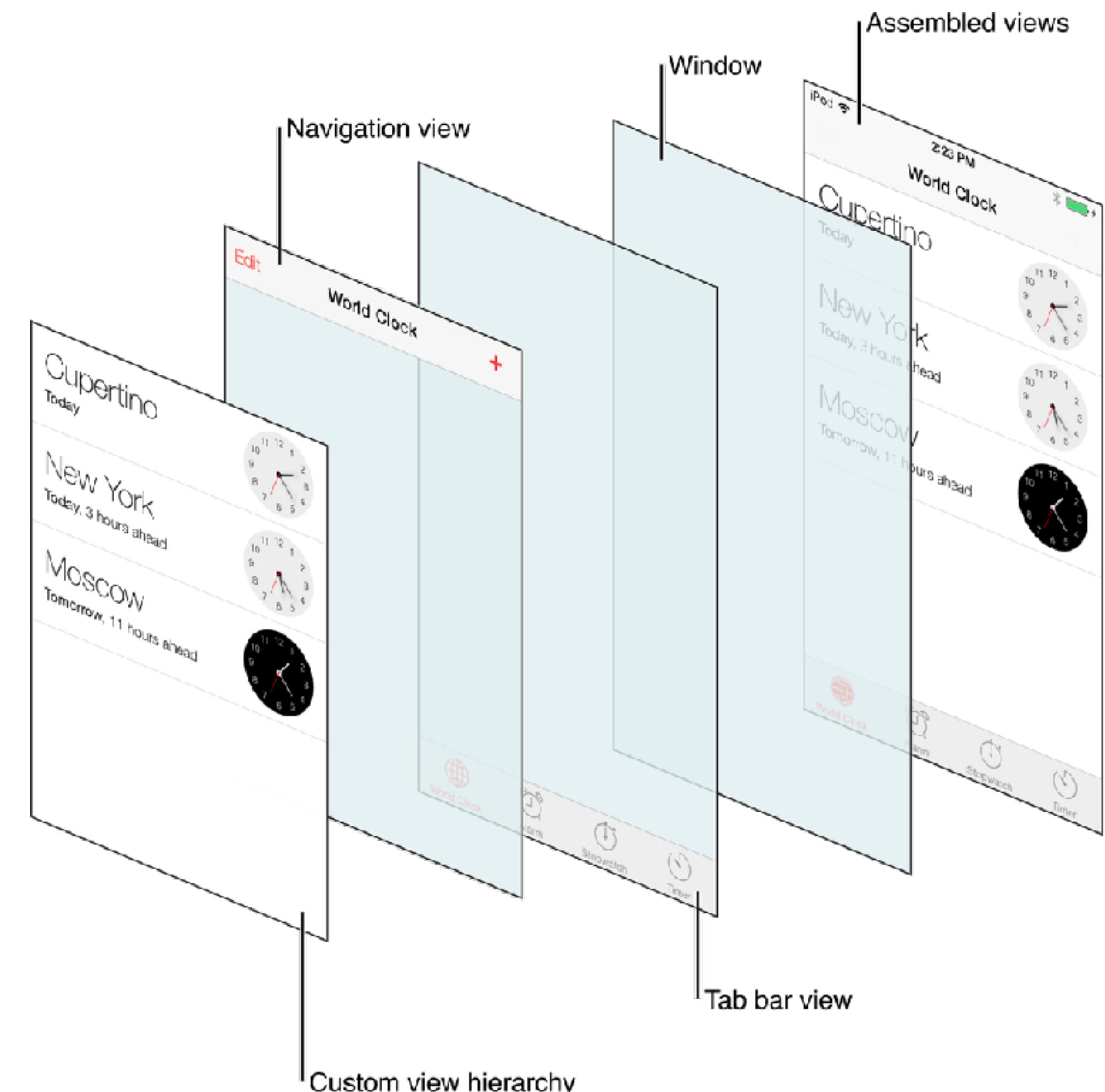


# UINavigationController

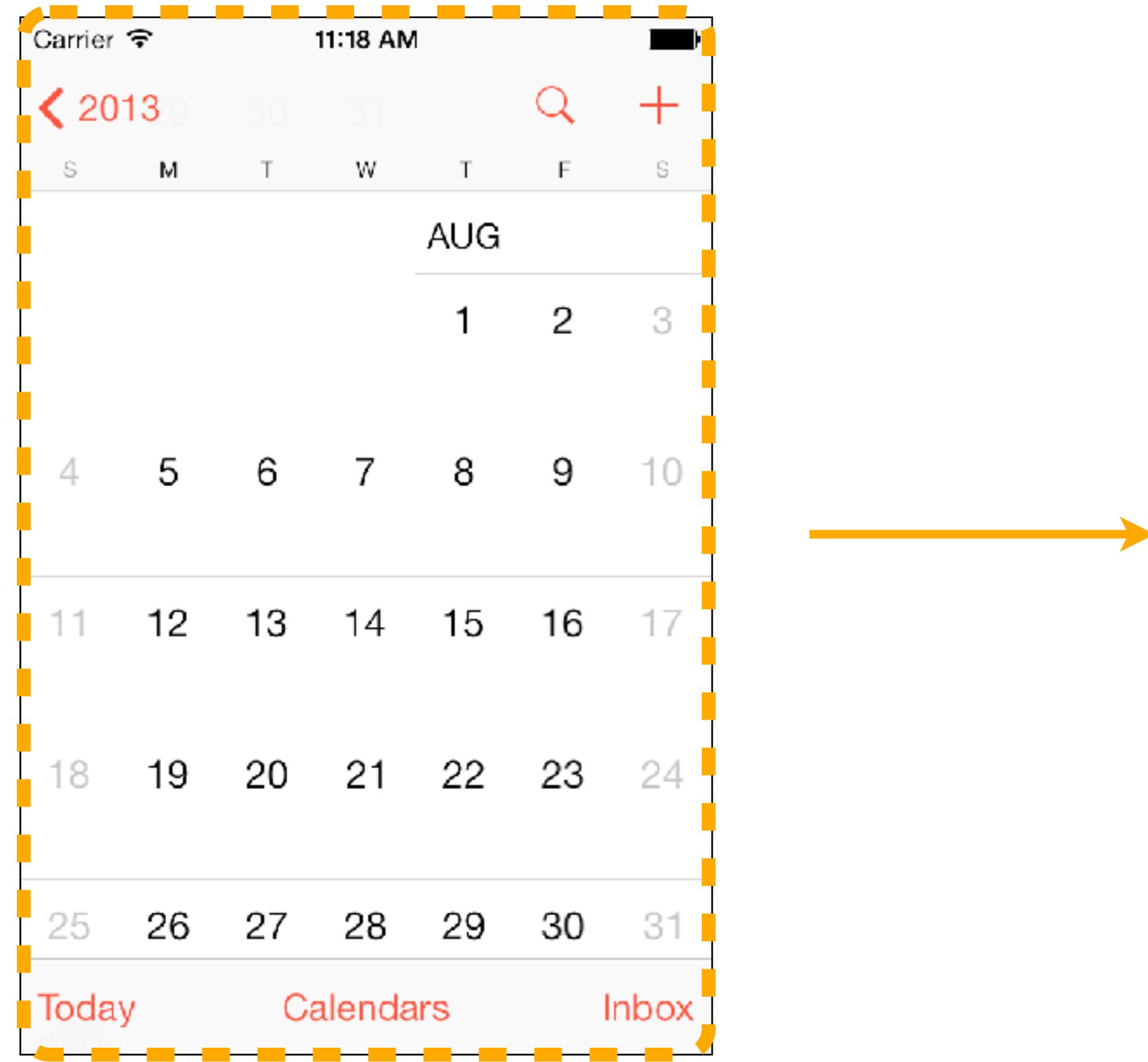
- The navigation controller manages a **stack of view controllers**
- The navigation controller object provides methods to modify the stack at runtime:
  - `(void)pushViewController:(UIViewController *)viewController animated:(BOOL)animated` is used to push a view controller at the top of the stack
  - `(UIViewController *)popViewControllerAnimated:(BOOL)animated` is used to pop a view controller from the top of the stack
  - `(NSArray *)popToViewController:(UIViewController *)viewController animated:(BOOL)animated` is used to pop all view controllers from the stack until the specified view controller is at the top of the stack
  - `(NSArray *)popToRootViewControllerAnimated:(BOOL)animated` is used to pop all the view controllers on the stack except the root view controller
- Every `UIViewController` has a `navigationController` property that can be used to access the `UINavigationController` it is embedded in

# UINavigationController

- The view for a navigation controller is just a container for several other views:
  - a navigation bar
  - an optional toolbar
  - the view containing custom content
- When moving from a MVC to another, the content of the custom view changes, as well as of the navigation bar and toolbar
- Only the custom content view changes to reflect the view controller that is at the top of the navigation stack

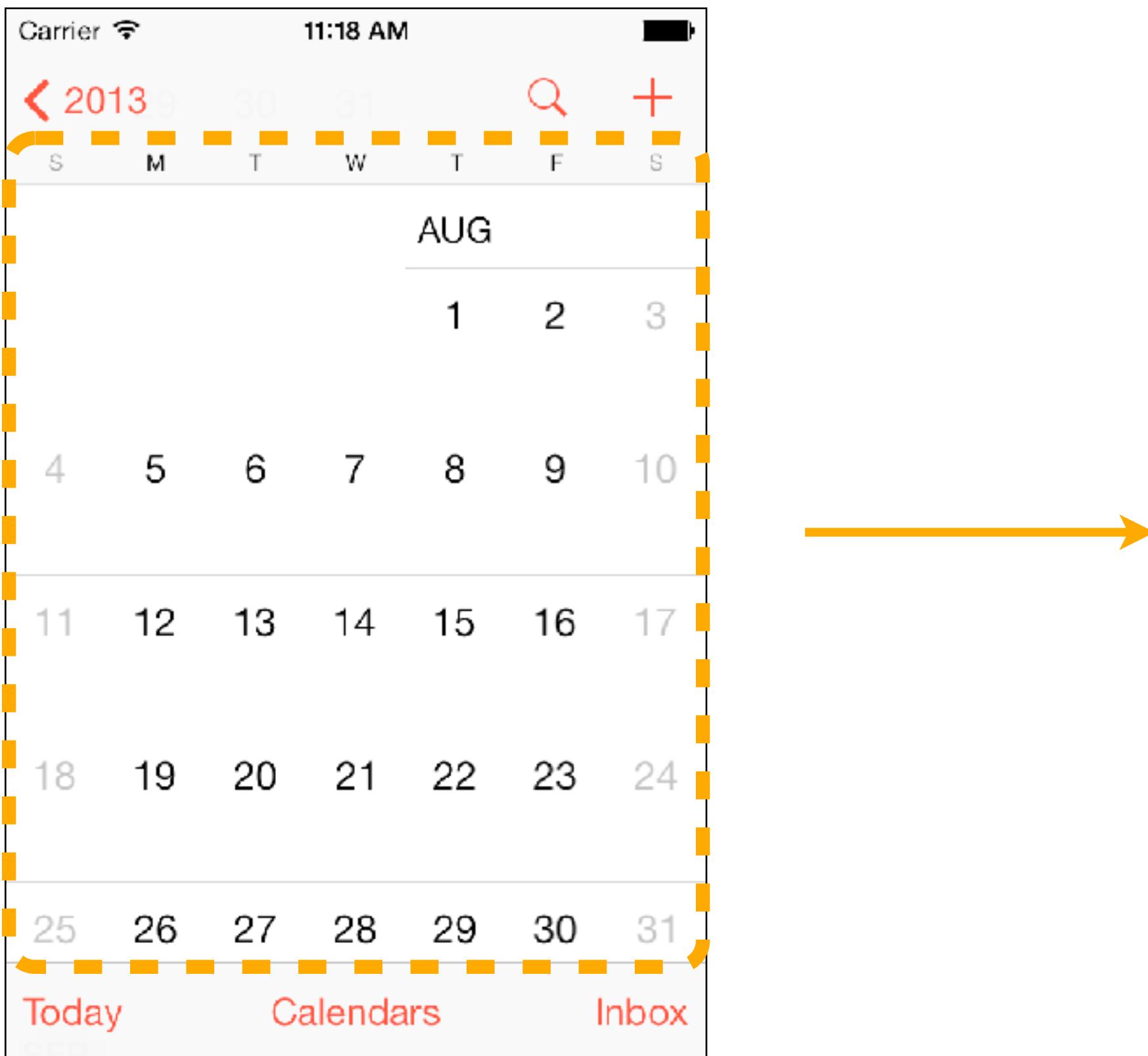


# View of a UINavigationController



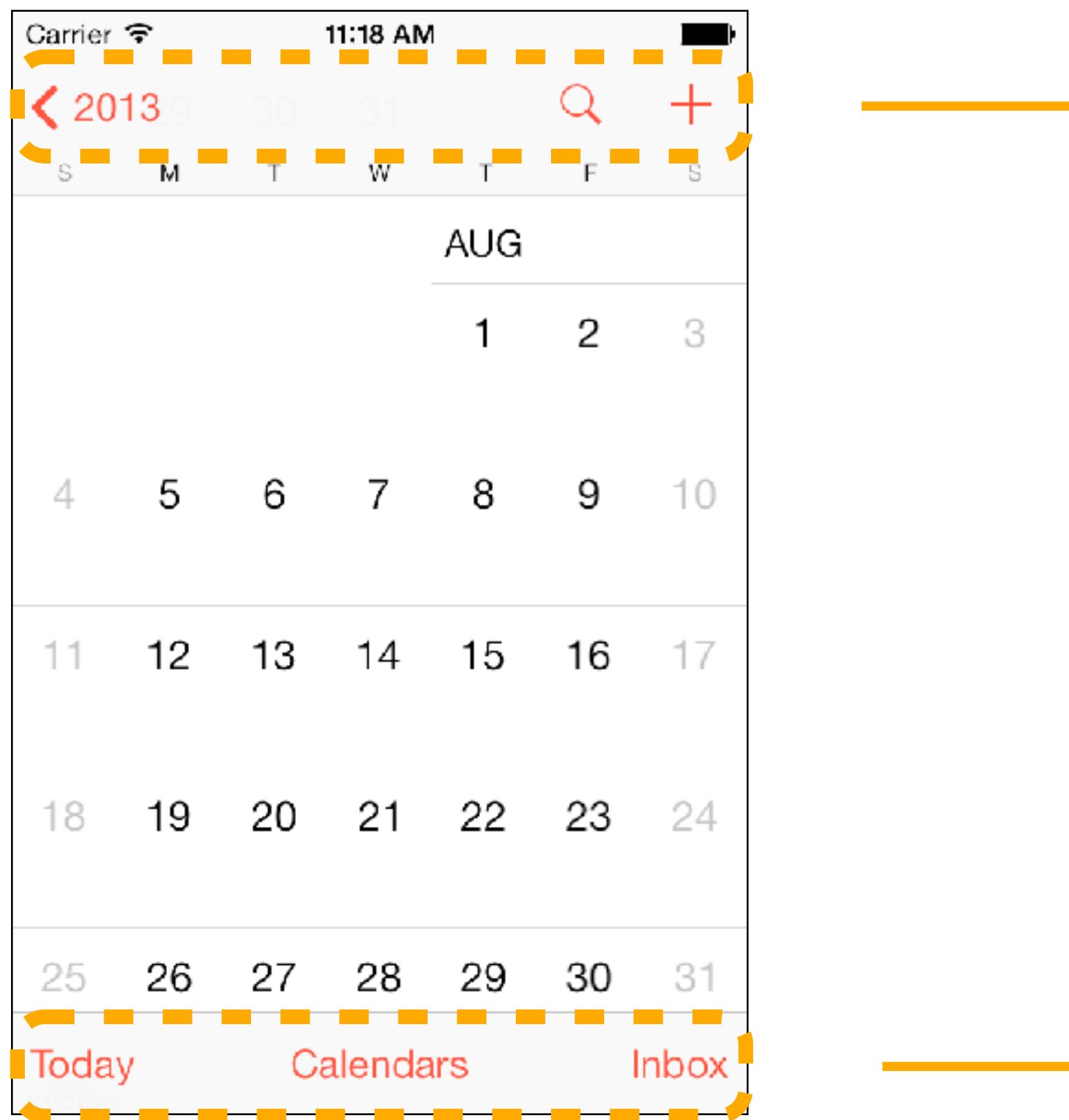
View of the navigation controller

# View of a UINavigationController



Custom content of the navigation controller managed by the view controller at the top of the stack

# View of a UINavigationController

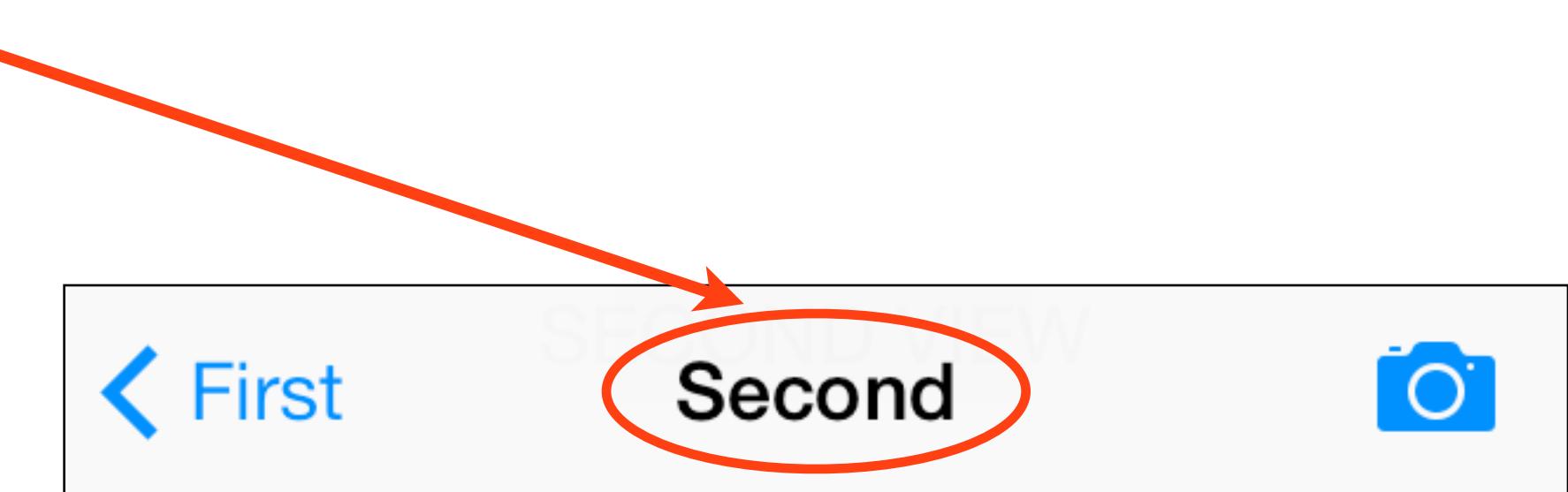


Navigation bar

Navigation toolbar (optional) -  
`toolbarItems` property of the  
embedded `UIViewController`

# UINavigationBar

- The **UINavigationBar** displays:
  - a title for the currently displayed view controller; it can be set using the **title** property of the embedded **UIViewController**
  - a back button which displays the **title** of the previous **UIViewController** in the navigation stack
  - an array of **UIBarButtonItem** objects (NOT **UIButton**!) accessible with the **UIViewController** property **navigationItem.rightBarButtonItem**



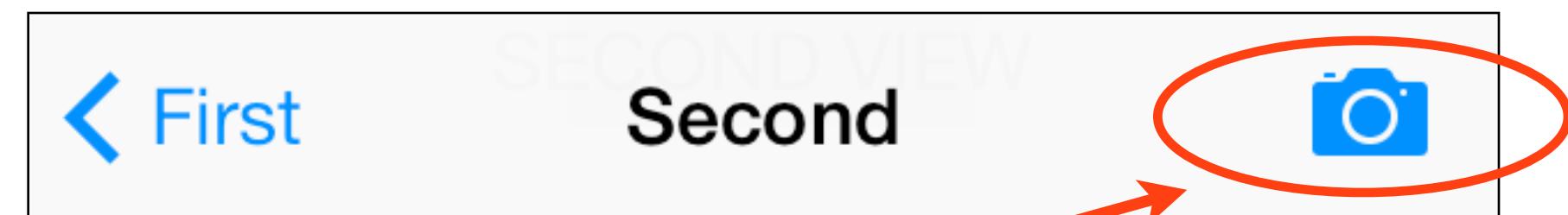
# UINavigationBar

- The **UINavigationBar** displays:
  - a title for the currently displayed view controller;  
it can be set using the **title** property of the embedded **UIViewController**
  - a back button which displays the **title** of the previous **UIViewController** in the navigation stack
  - an array of **UIBarButtonItem** objects (NOT **UIButton**!) accessible with the **UIViewController** property **navigationItem.rightBarButtonItem**



# UINavigationBar

- The **UINavigationBar** displays:
  - a title for the currently displayed view controller; it can be set using the **title** property of the embedded **UIViewController**
  - a back button which displays the **title** of the previous **UIViewController** in the navigation stack
  - an array of **UIBarButtonItem** objects (NOT **UIButton**!) accessible with the **UIViewController** property **navigationItem.rightBarButtonItem**





# UINavigationController with multiple MVCs

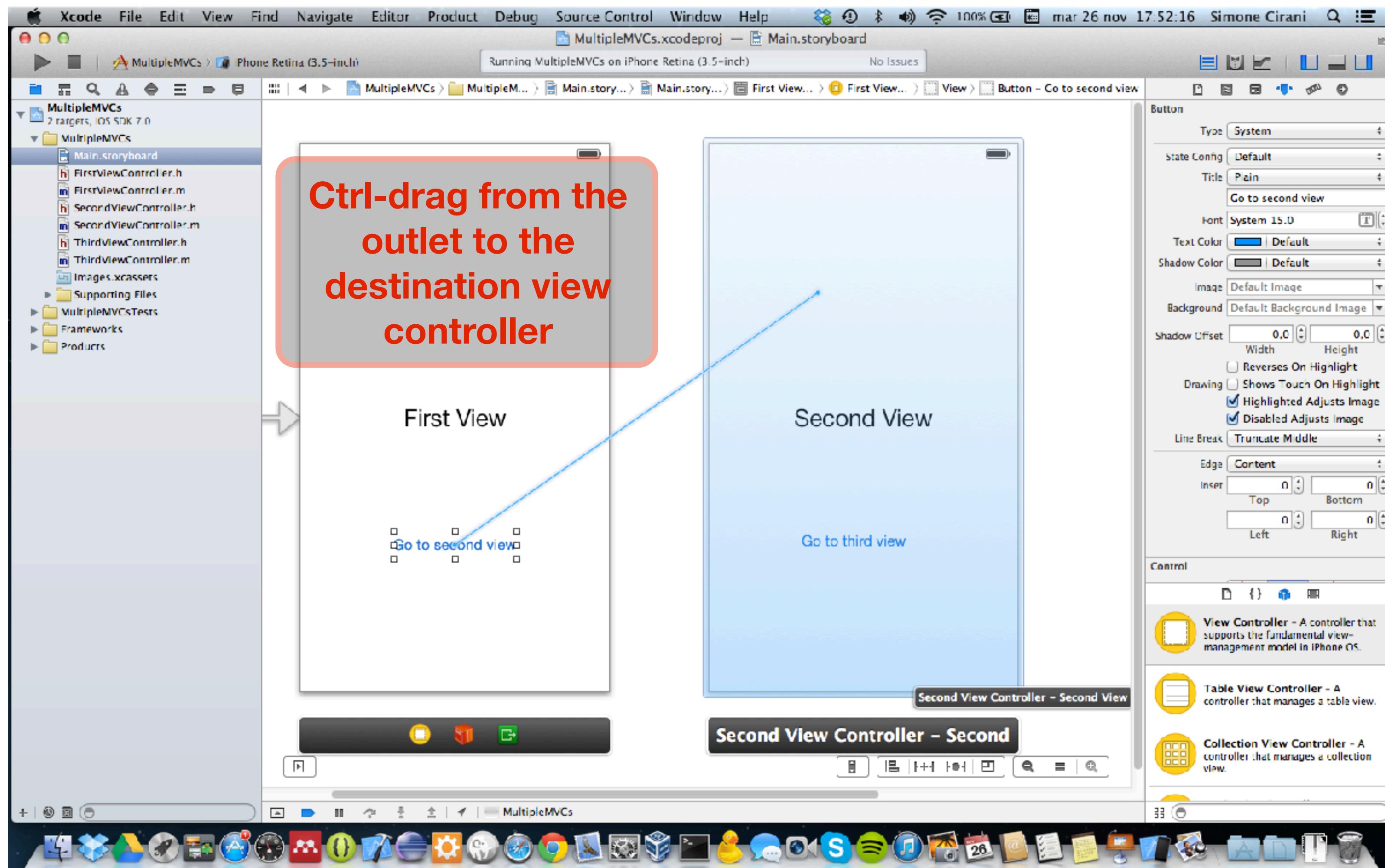
- A **UINavigationController** starts displaying no content
- When the **rootViewController** property of the **UINavigationController** is set, the **UINavigationController** displays the view managed by the view controller
- Segues define the transitions from a view controller to the next one



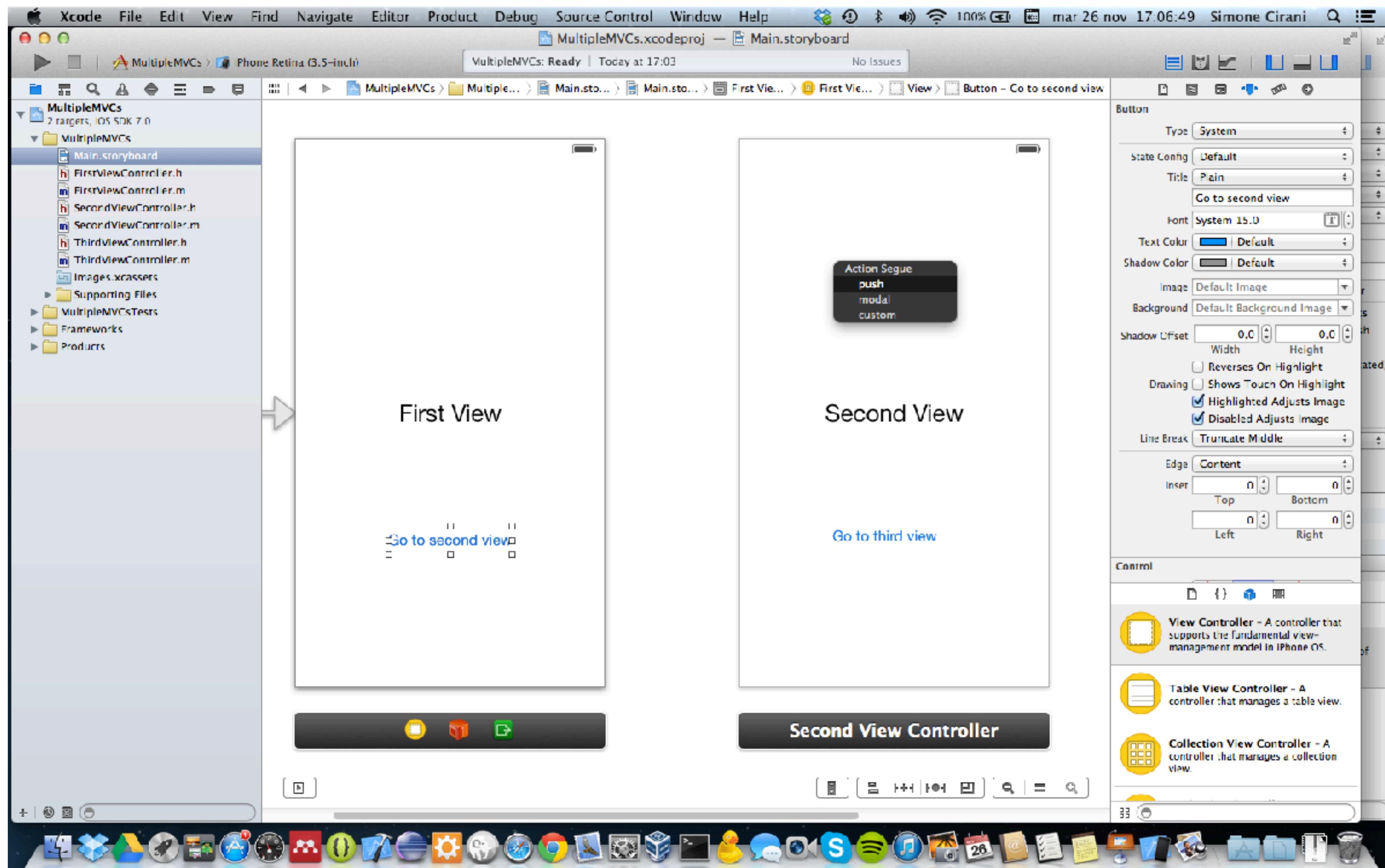
# Segues

- Segues are responsible for performing the visual transition between two view controllers
- A transition can be triggered:
  - from an outlet in storyboard (segue)
  - programmatically
- Segues can be of different types:
  - push
  - modal
  - custom

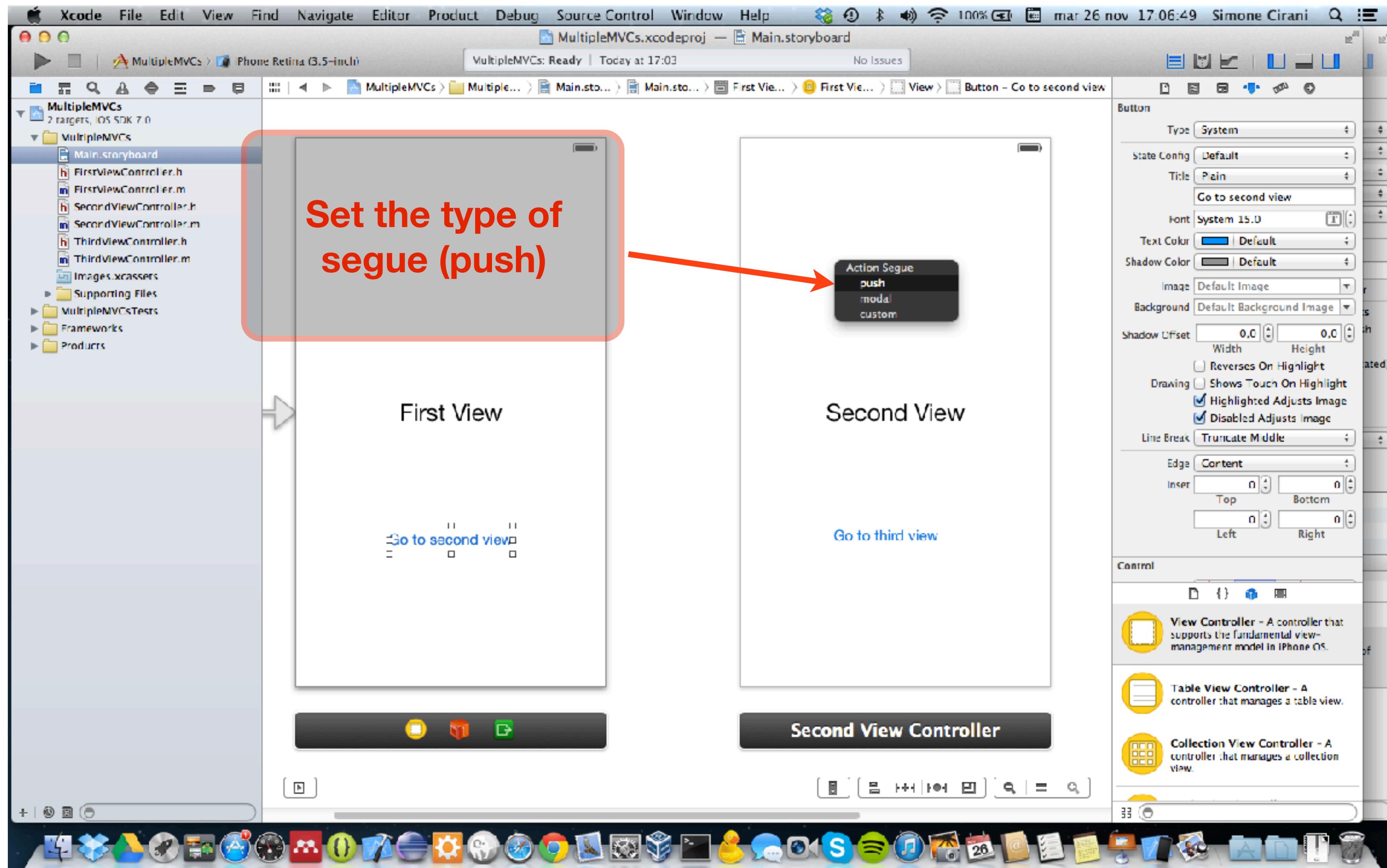
# Segues in storyboard



# Segues in storyboard

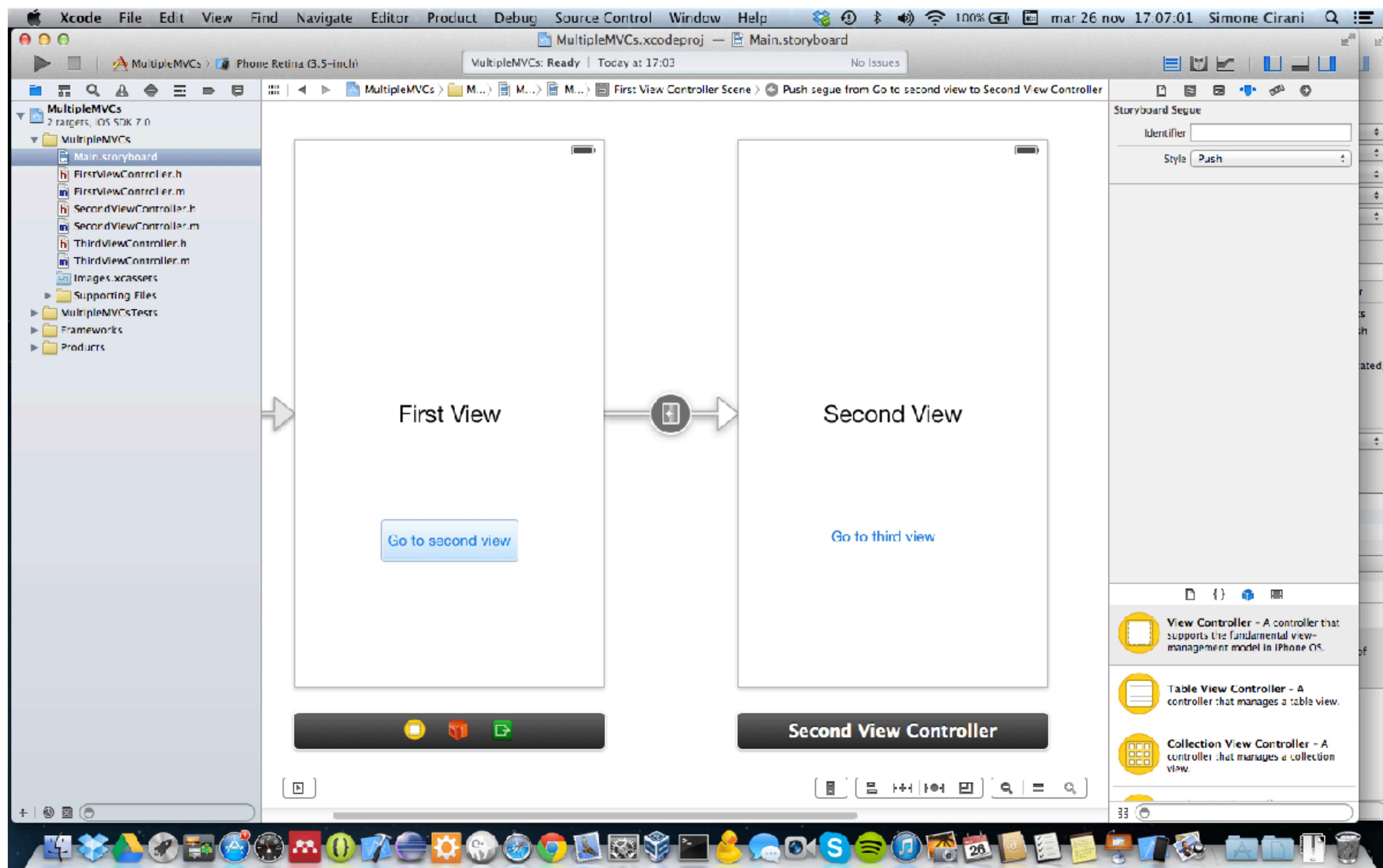


# Segues in storyboard

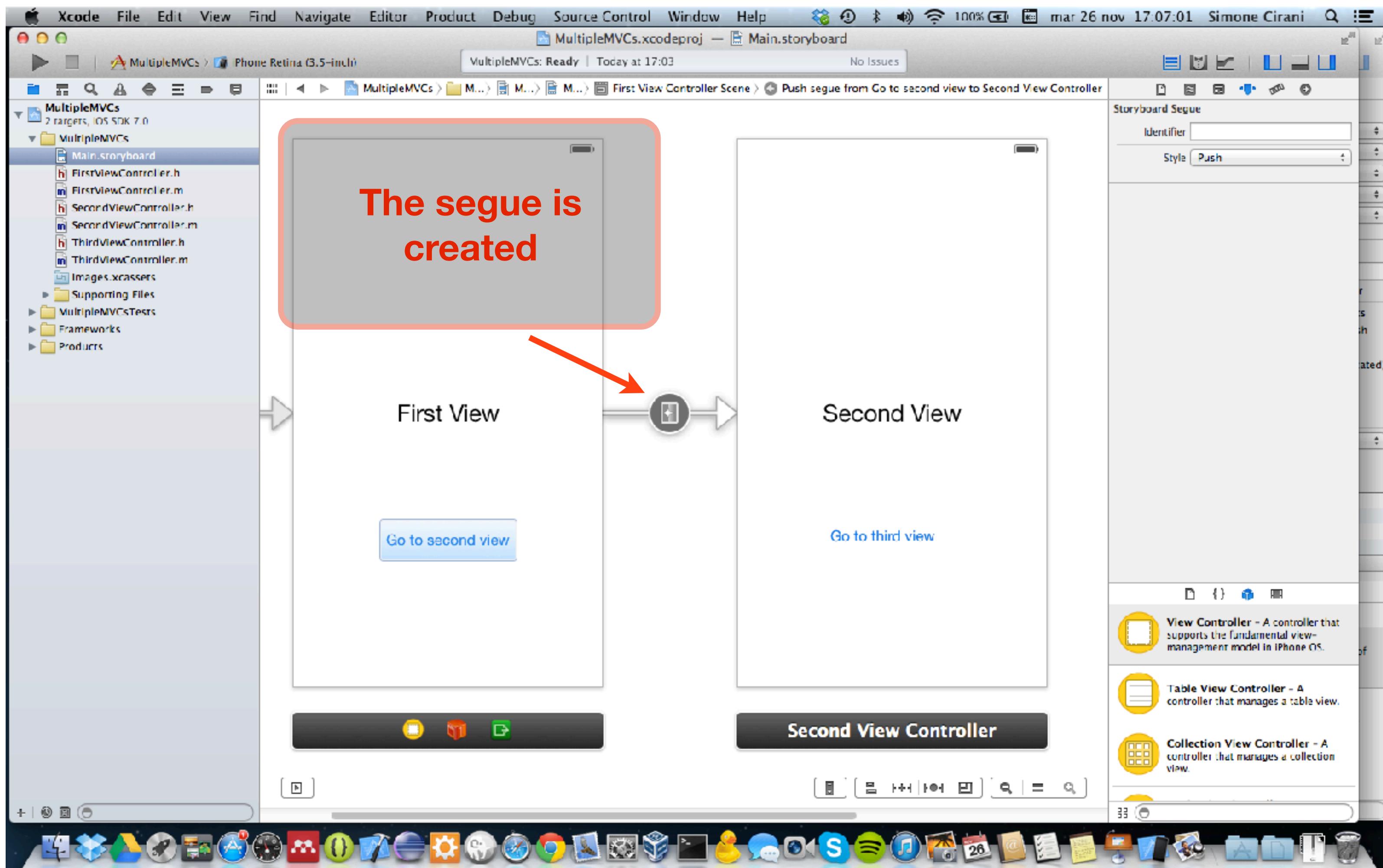




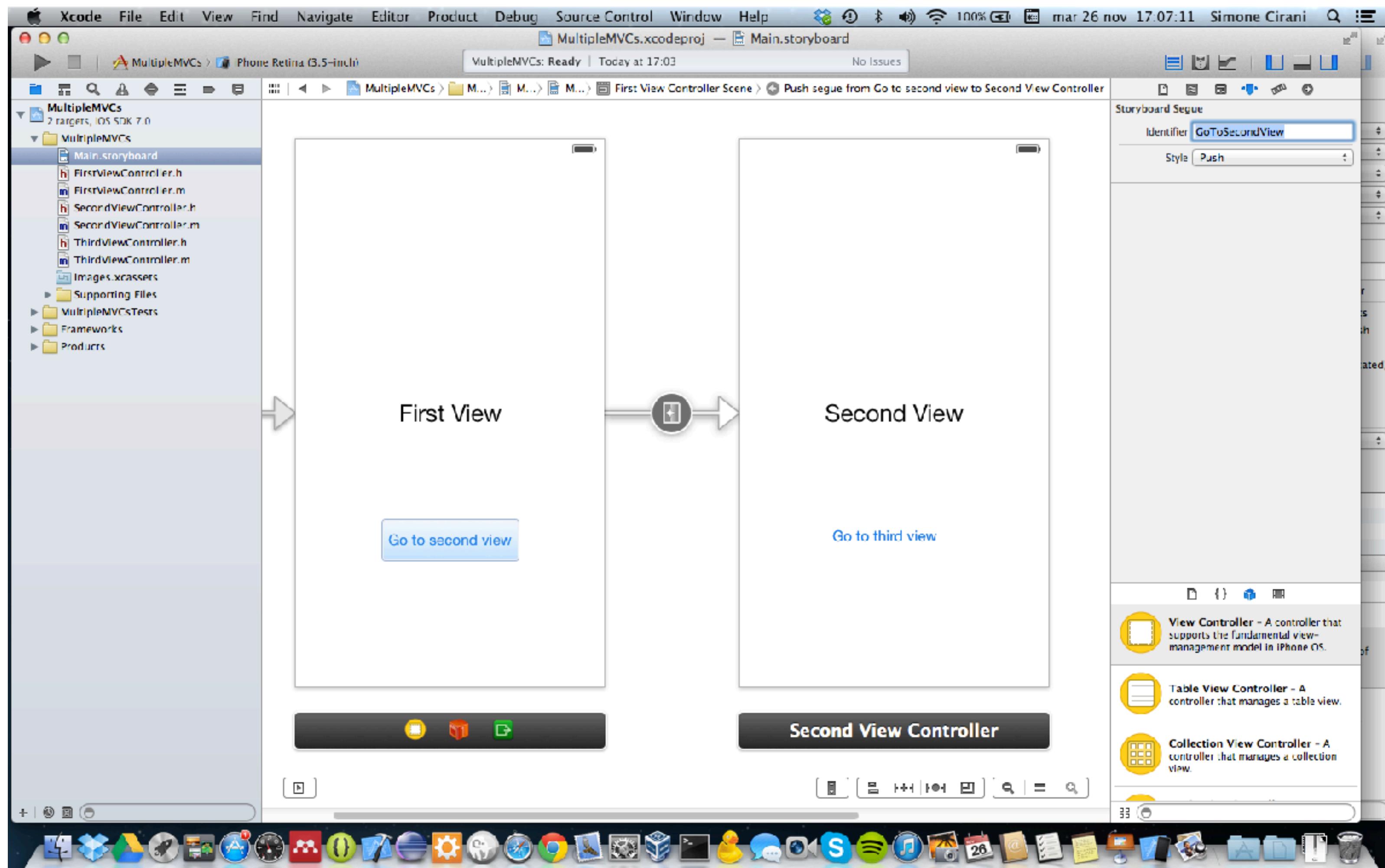
# Segues in storyboard



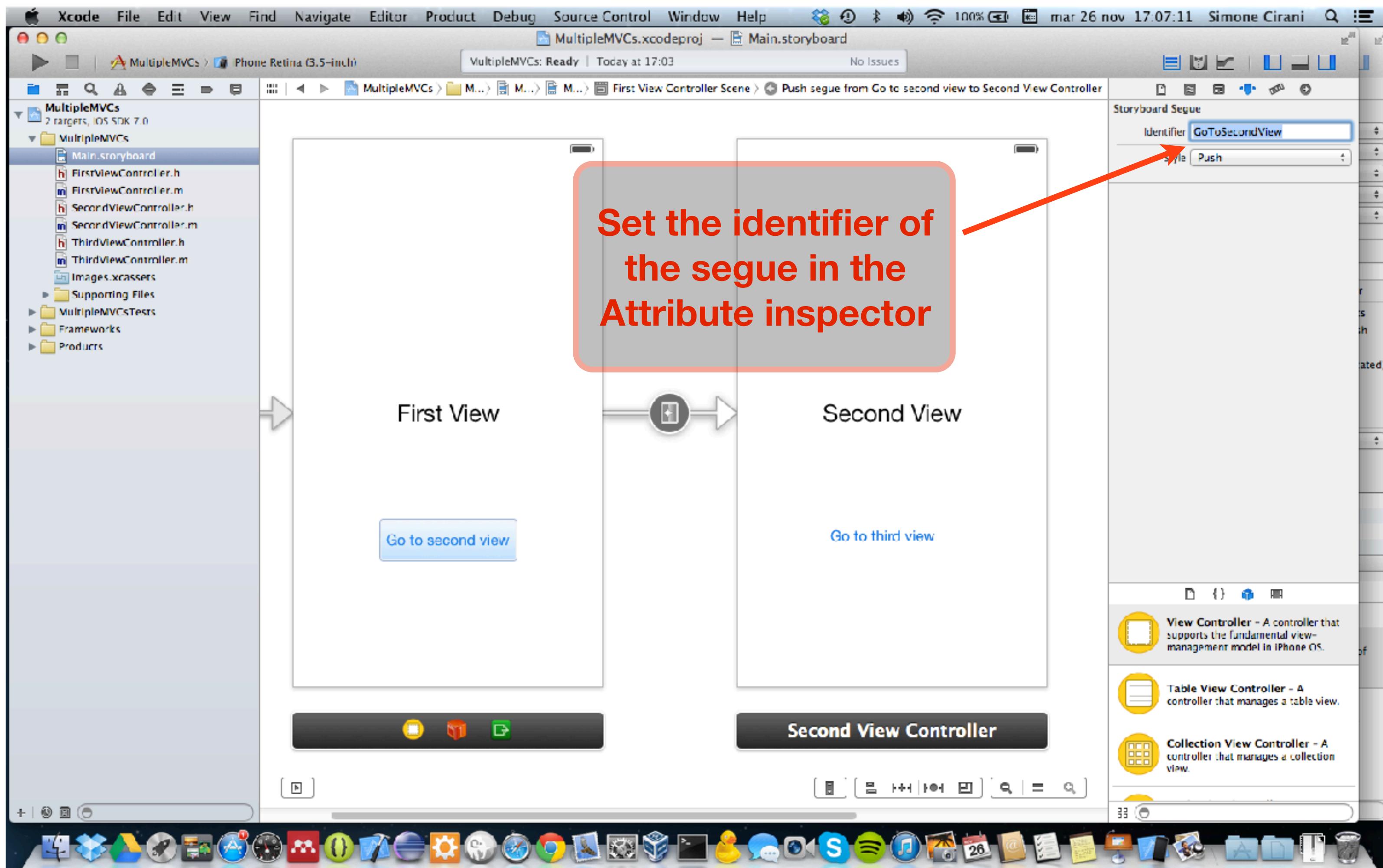
# Segues in storyboard



# Segues in storyboard



# Segues in storyboard





# Segues

- A view controller can perform additional setup before performing the segue (it should have the chance to be “prepared”)
- Preparing the view controller to perform the segue allows to **pass important data to the new view controller before performing the segue**
- Every UIViewController can override the following method to prepare for the segue:
  - `(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender`
- This method is called before the new view controller (destinationViewController of the segue) is presented
- The segue that will be executed is passed in as an argument; it can be identified through its **identifier** property (previously set in the Attribute inspector)



# Preparing for a segue

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
 if([segue.identifier isEqualToString:@"GoToSecondView"]) {
 if([segue.destinationViewController isKindOfClass:[SecondViewController class]]) {
 SecondViewController *sVC = (SecondViewController *)segue.destinationViewController;
 sVC.data = @"Some data";
 }
 }
}
```



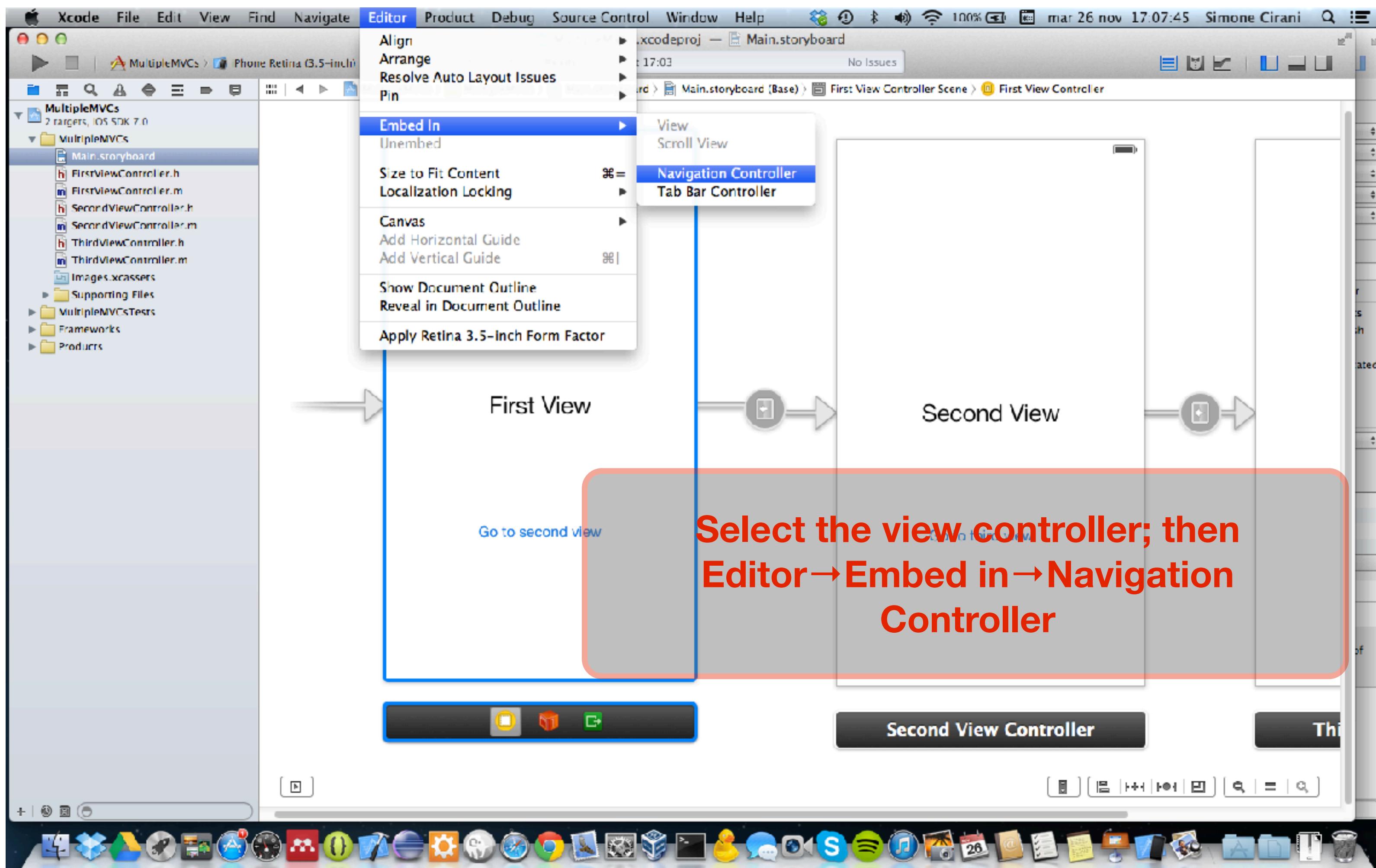
# Instantiating view controllers programmatically

- A view controller can be instantiated “by-hand” (not from the storyboard)
- For example, when the type of destination view controller depends on some information that can be resolved only at runtime (e.g. an error screen or a content screen)

```
NSString *vc = @"SecondViewController";
UIViewController *controller = [self.storyboard instantiateViewControllerWithIdentifier:vc];
```

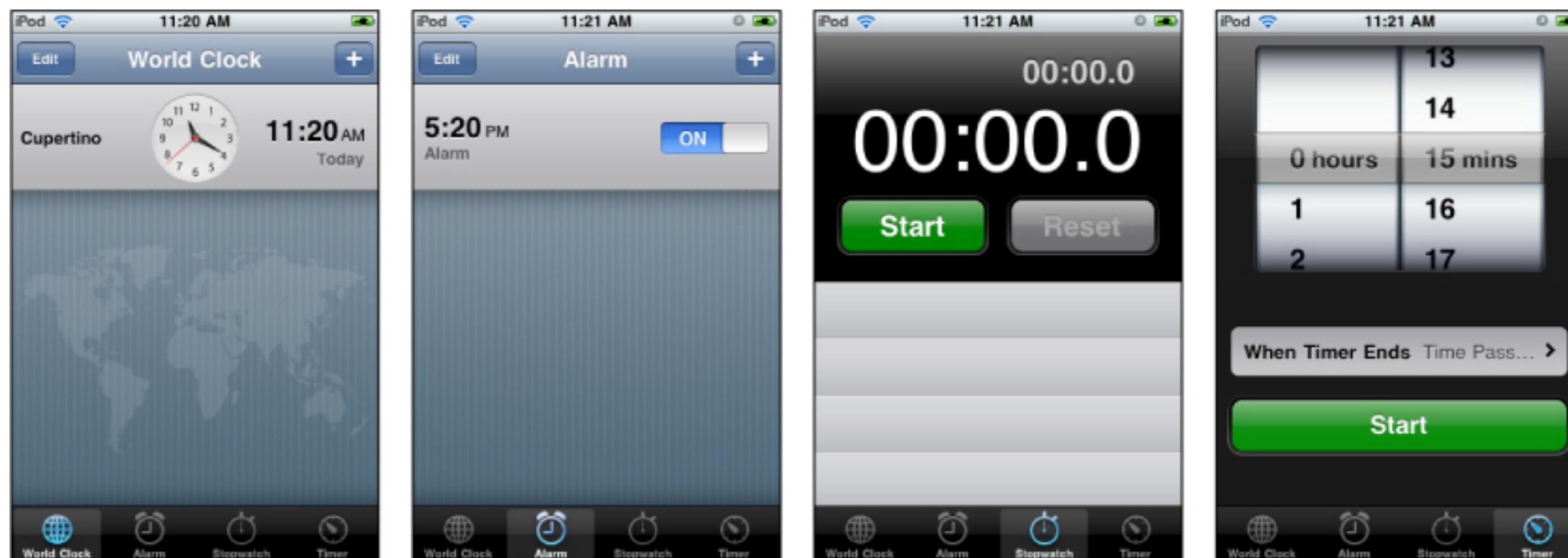
- The view controller is then pushed to the navigation stack with the **pushViewController:animated:** method

# Embedding view controllers in a UINavigationController



# UITabBarController

- **UITabBarController** is a class that implements a specialized view controller that manages a radio-style selection interface
- Typical usage: **multiple sections** (e.g. Clock app: Clock, Alarm, Stopwatch, Timer)
- Tabs are displayed at the bottom of the window (tab bar) for selecting between the different modes and for displaying the views for that mode
- Example of a navigation controller-based app: Clock app

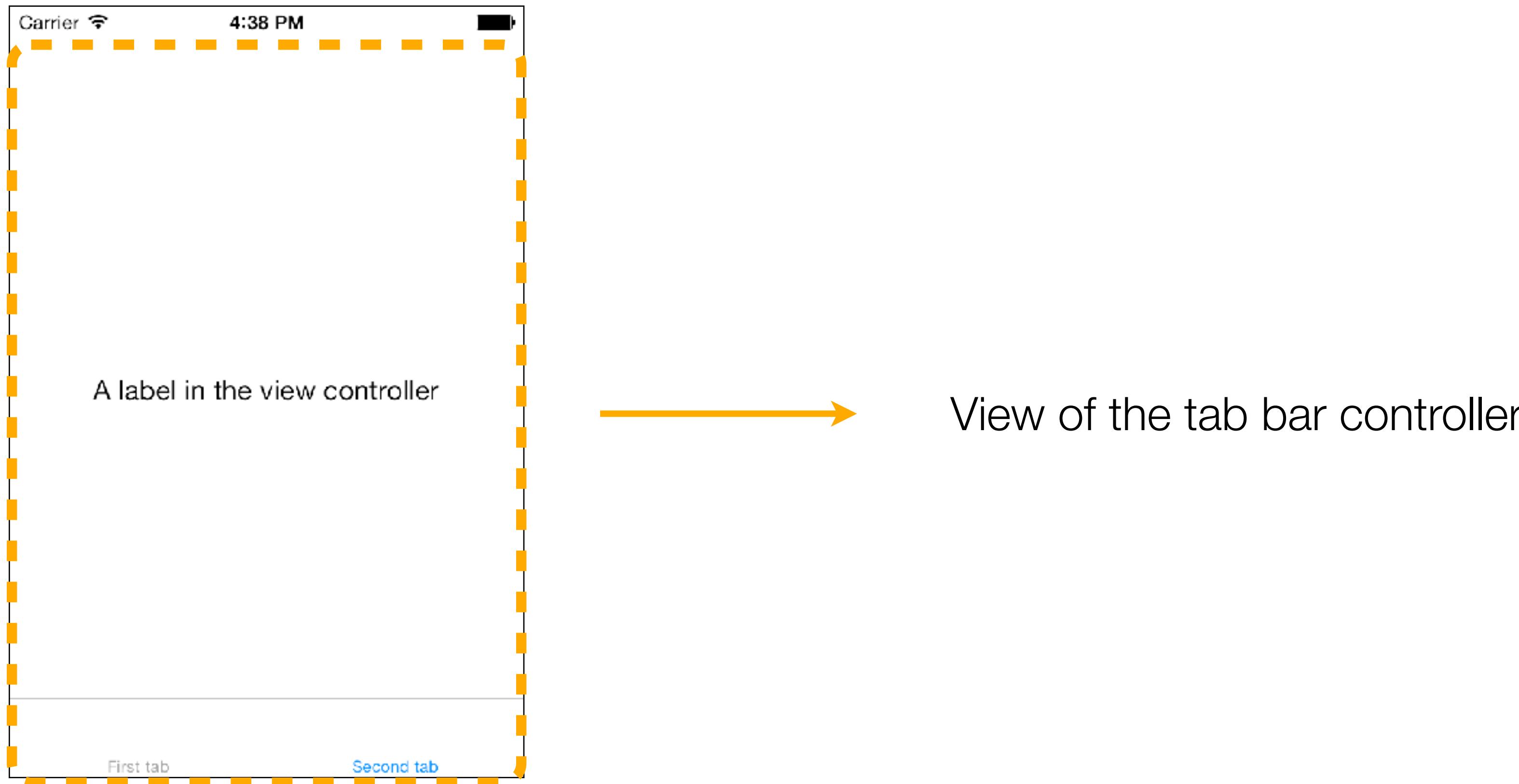


# UITabBarController

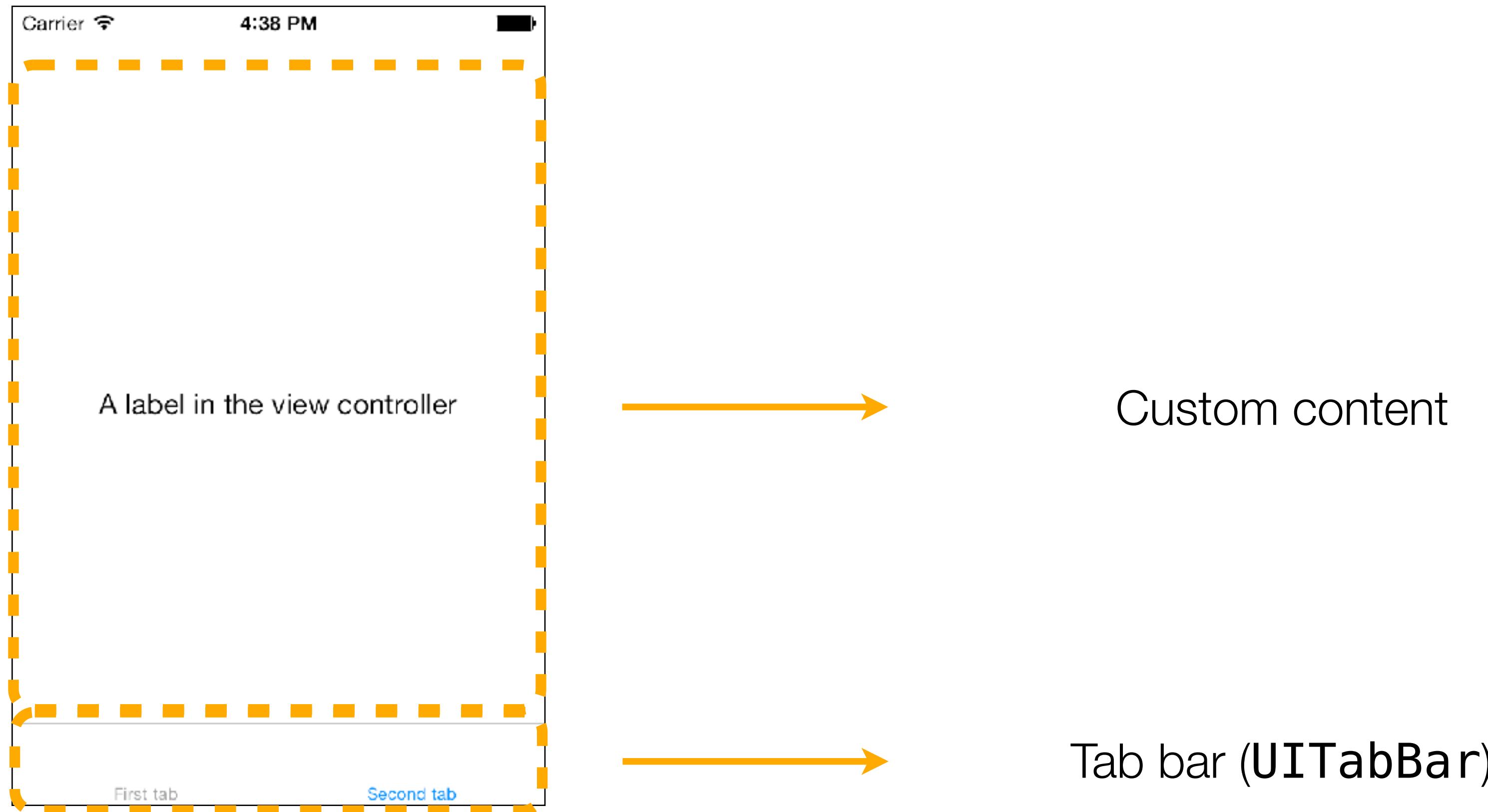
- Each tab of a tab bar controller interface is associated with a custom view controller
- When the user selects a specific tab, the tab bar controller displays the root view of the corresponding view controller, replacing any previous views
- View controllers are assigned to the tab bar controller by setting the `viewControllers` property of the `UITabBarController` (`NSArray` of `UIViewController`s); order is maintained



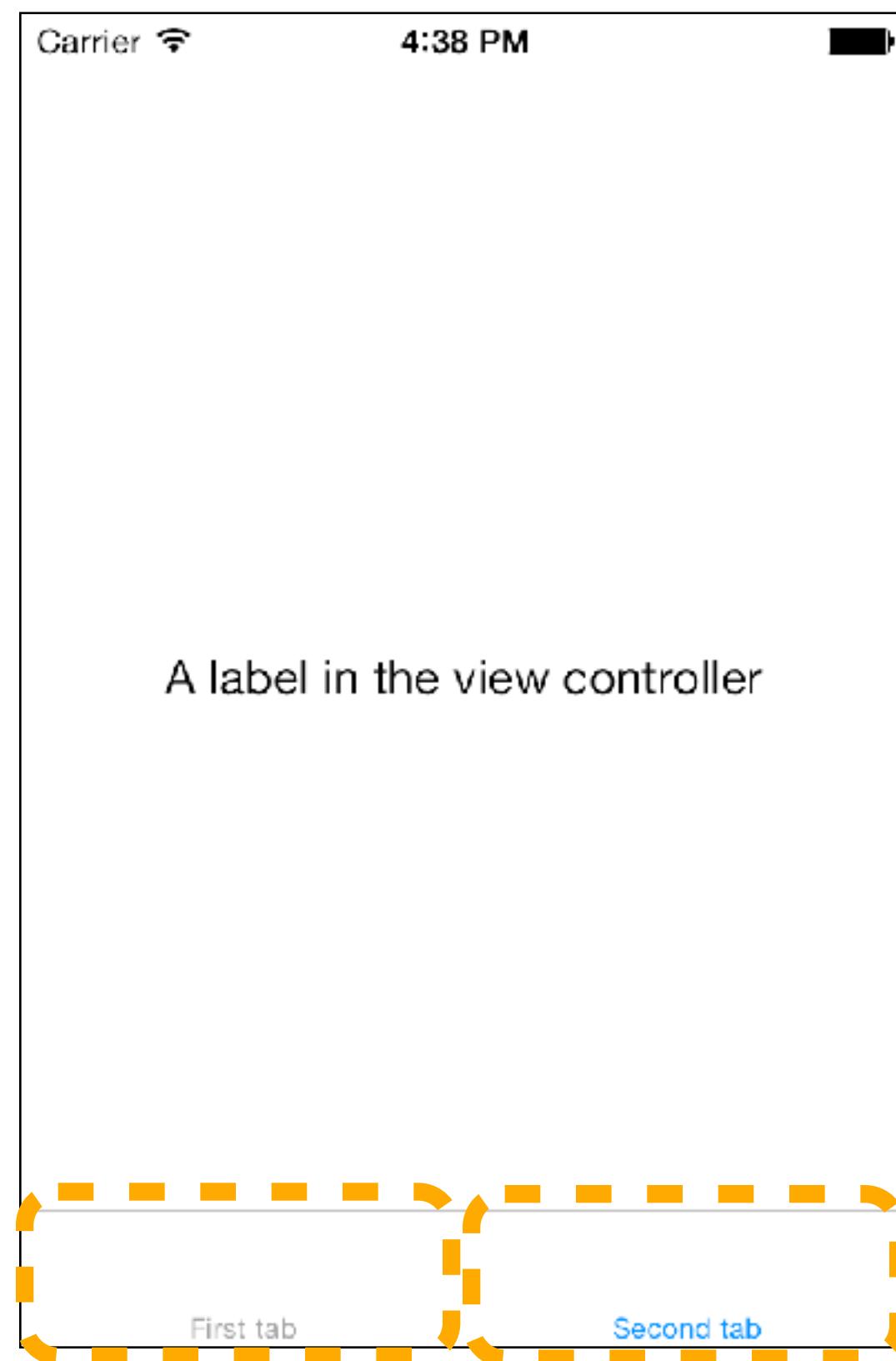
# Views of a UITabBarController



# Views of a UITabBarController



# Views of a UITabBarController



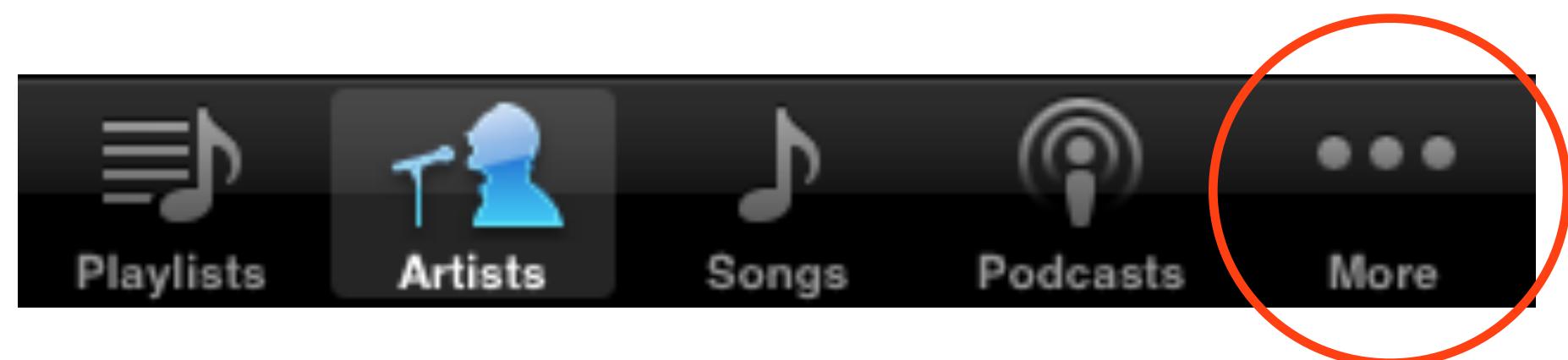


# Views of a UITabBarController

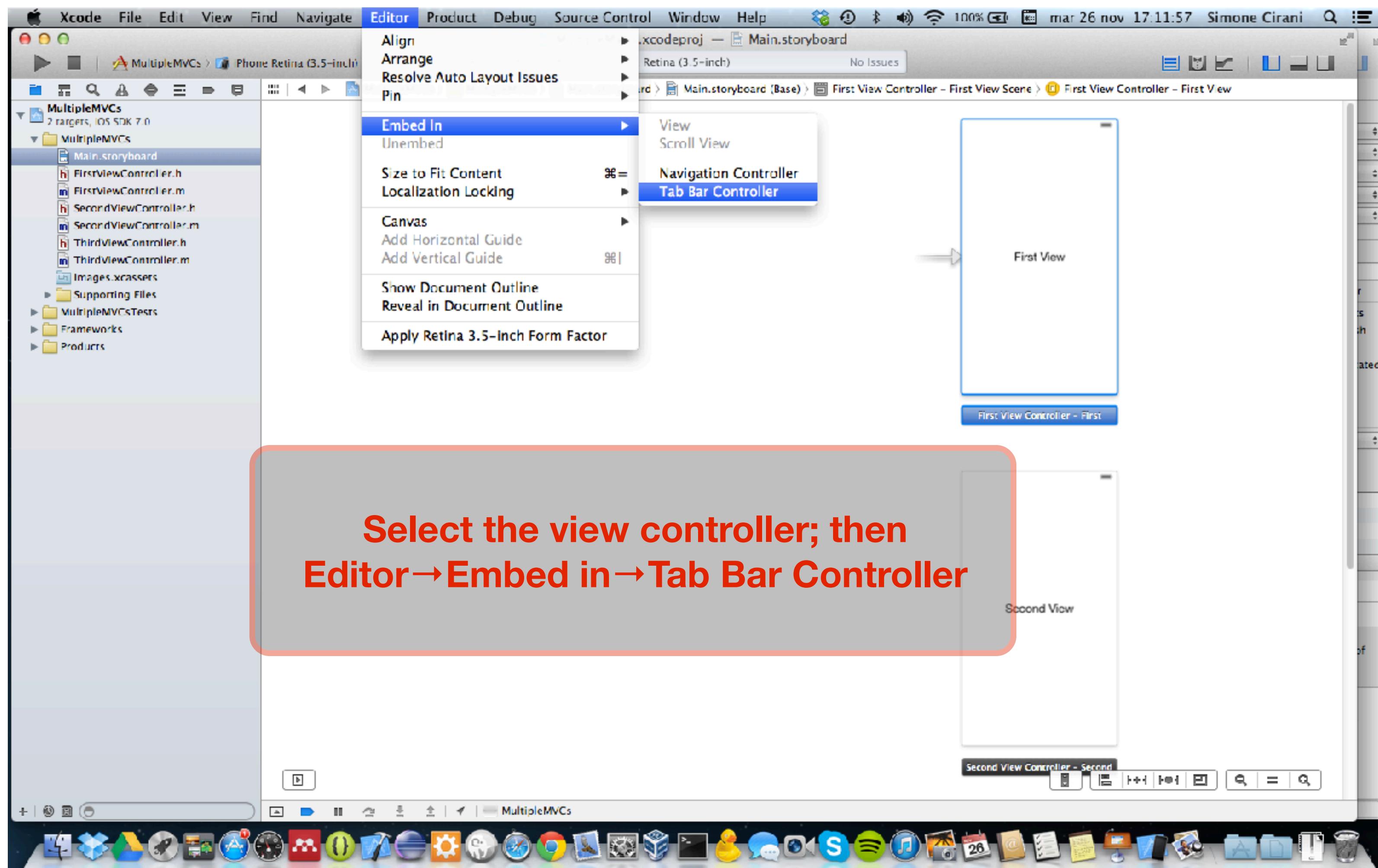
- The embedded view controller defines the appearance of its corresponding tab bar item
- The title of the tab bar item for the currently displayed view controller can be set using the **title** property of the embedded **UIViewController** (if there is no instance of **UITabBarItem** associated to the tab bar the title is displayed by default with no image)
- The appearance of the tab bar item (title and icon) can be configured in storyboard
- To associate a custom tab bar item programmatically:
  1. create an instance of **UITabBarItem** to configure its title and image
  2. set the **tabBarItem** property of the embedded view controller with the created item

# More navigation controller

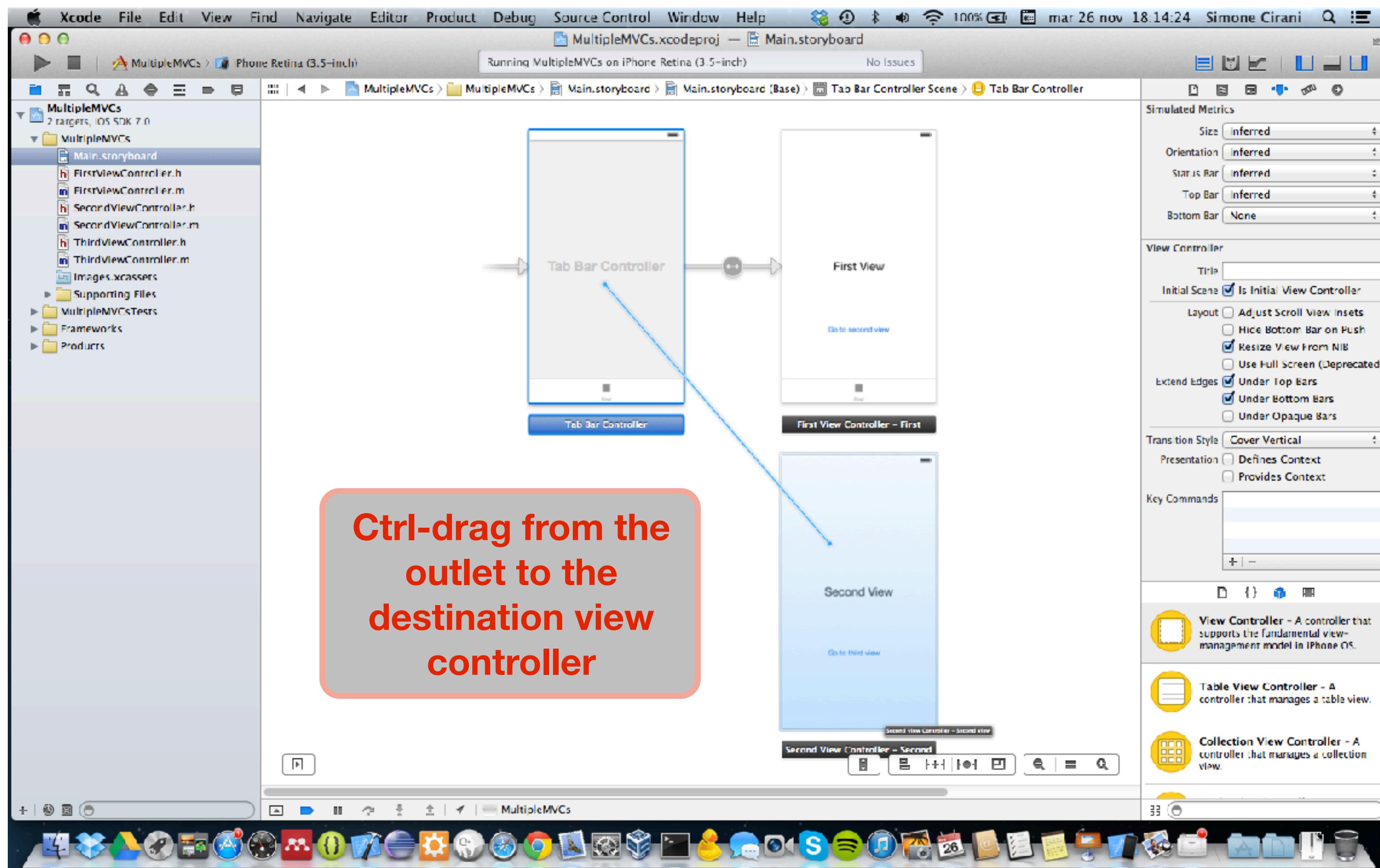
- The tab bar offers limited space to render tab items
- If 6 or more custom view controllers are added to a tab bar controller, the tab bar controller displays only the first four items plus the standard “More” item on the tab bar
- When the “More” tab is selected a view with all the remaining tabs is shown (automatically)



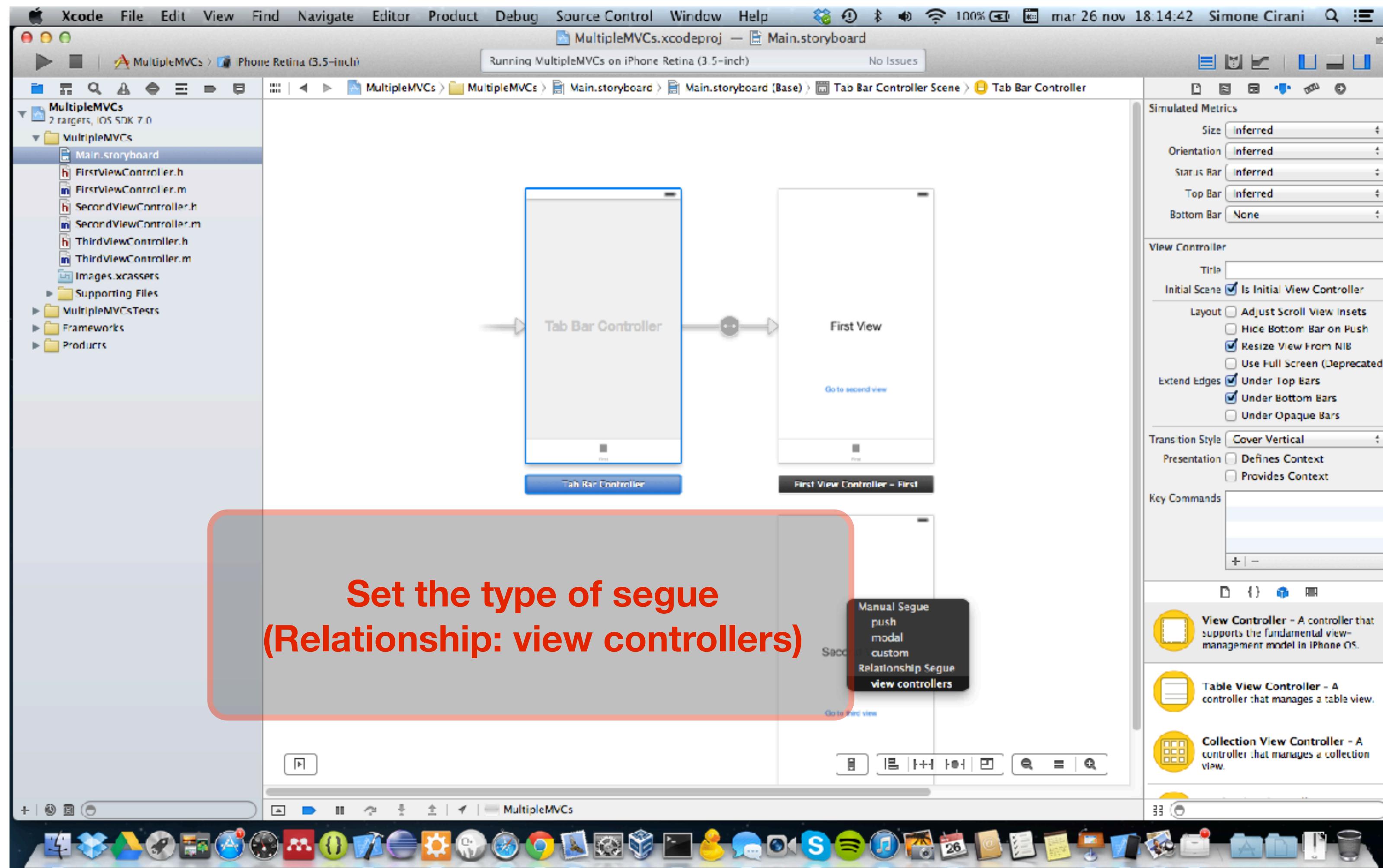
# Embedding view controllers in a UITabBarController



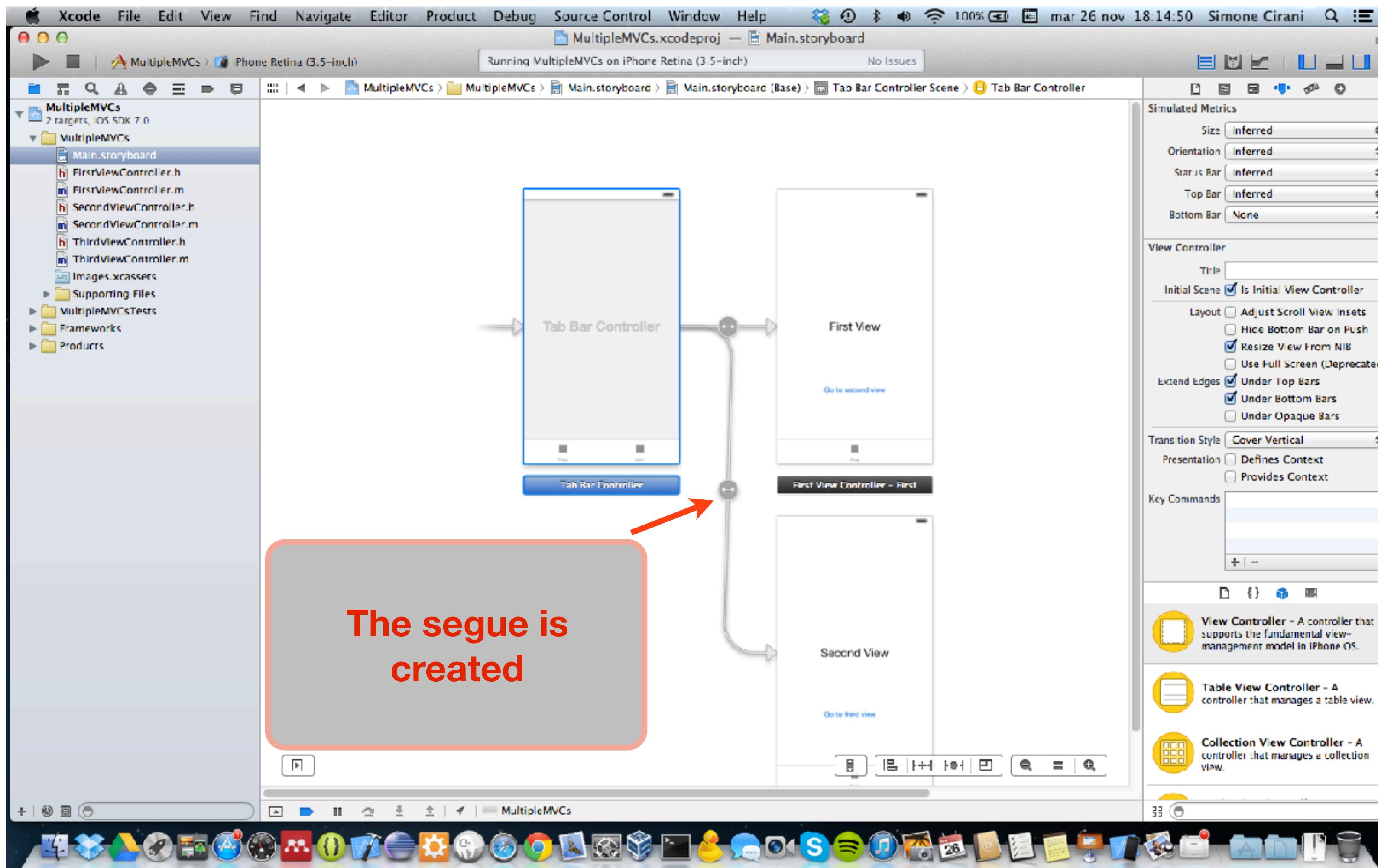
# Adding view controllers to a UITabBarController



# Adding view controllers to a UITabBarController

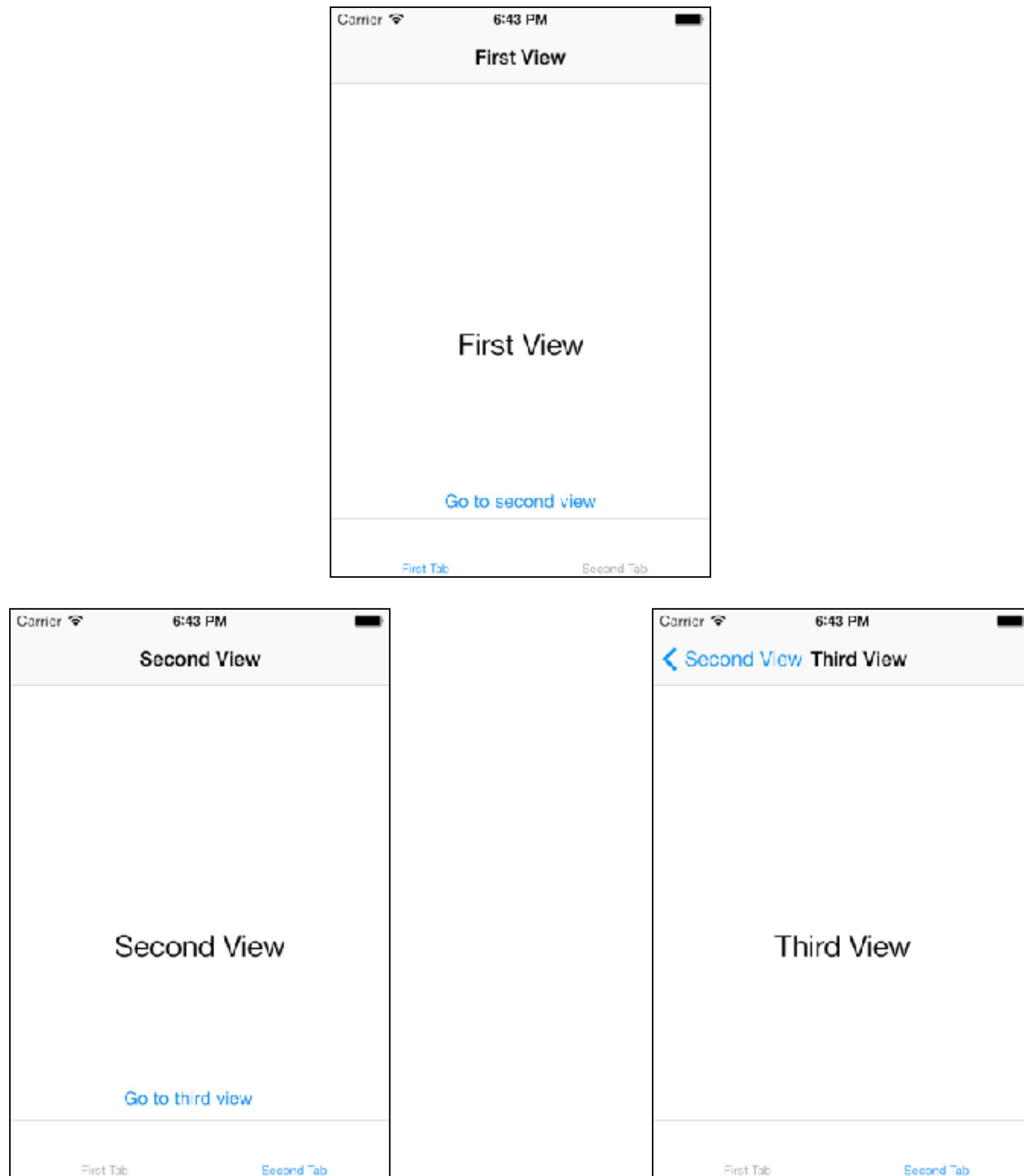


# Adding view controllers to a UITabBarController



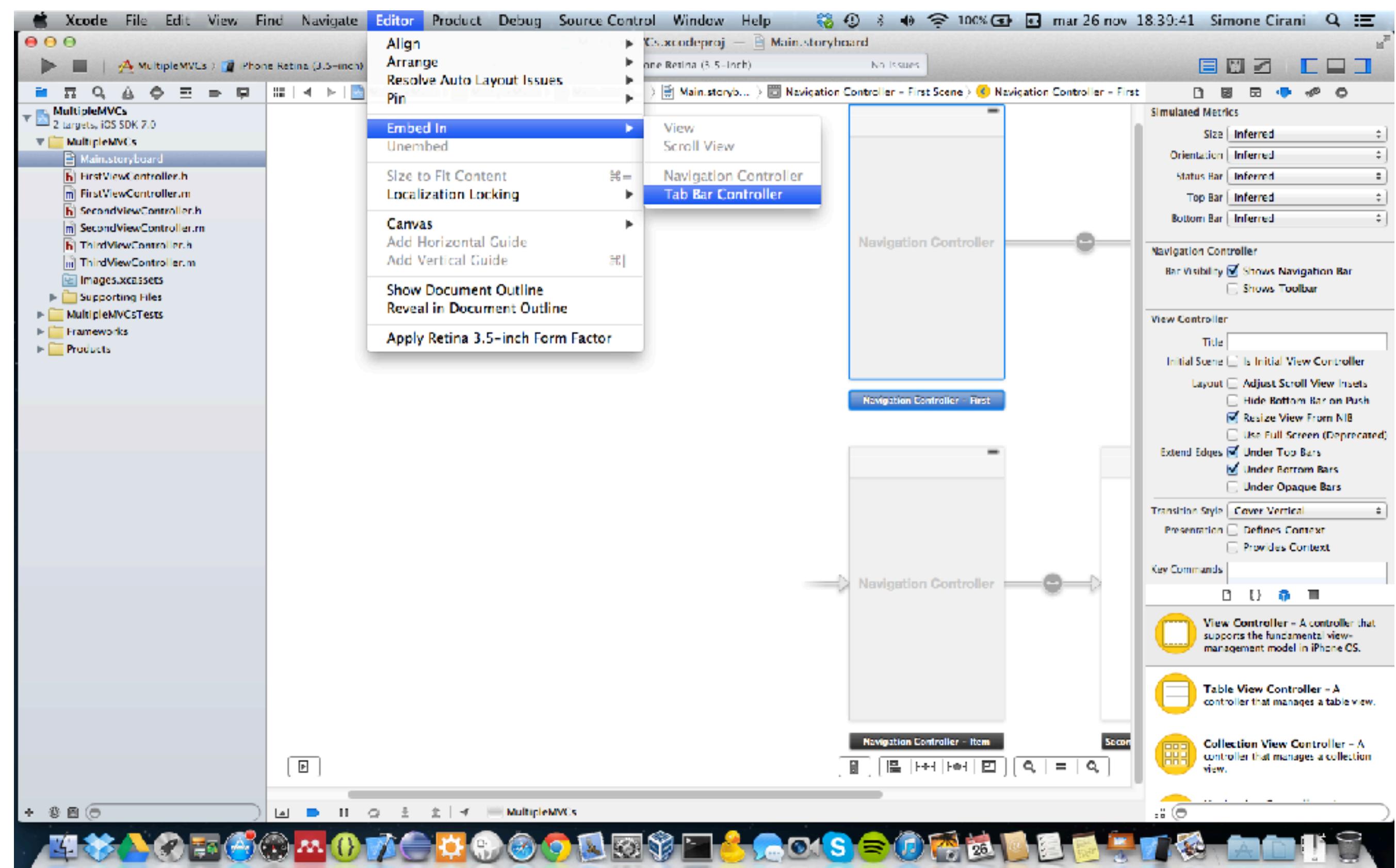
# Combining tab bar controllers and navigation controllers

- It is possible to combine tab bar controllers and navigation controllers
- A navigation controller can be embedded inside a tab bar controller, if one or more sections of the app require a drill-down navigation
- for example, the Music app has several sections (Artists, Albums, ...) and by getting into one of them we get into more detail
- Embedding a tab bar controller inside a navigation controller doesn't make much sense... probably redesigning the app would be better (indeed XCode does not allow you to do this)



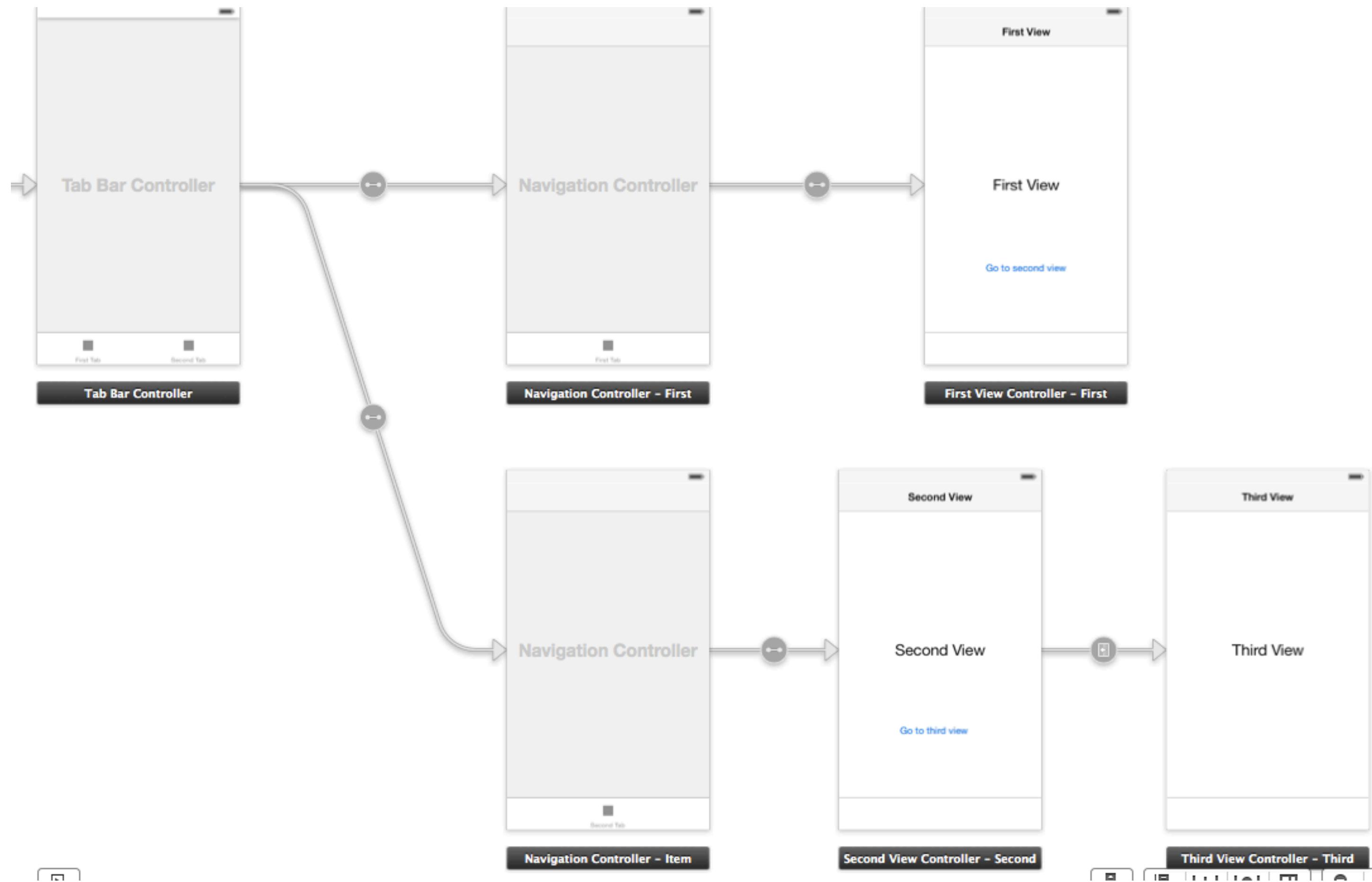
# Combining tab bar controllers and navigation controllers

- To embed a **UINavigationController** inside a **UITabBarController** in storyboard, select the navigation controller, then go to Editor → Embed in → Tab Bar Controller



# Combining tab bar controllers and navigation controllers

- It is possible to have complex storyboards for advanced user experience





# Mobile Application Development



Lecture 5  
Controllers of View Controllers



This work is licensed under a [Creative Commons Attribution-  
NonCommercial-ShareAlike 4.0 International License](#).



# Mobile Application Development



Lecture 6  
Blocks, Concurrency, Networking



This work is licensed under a [Creative Commons Attribution-  
NonCommercial-ShareAlike 4.0 International License](#).

# Lecture Summary

- Blocks
- Concurrency and multi-threading
- Grand Central Dispatch (GCD)
- Networking
- UIImage & UIImageView





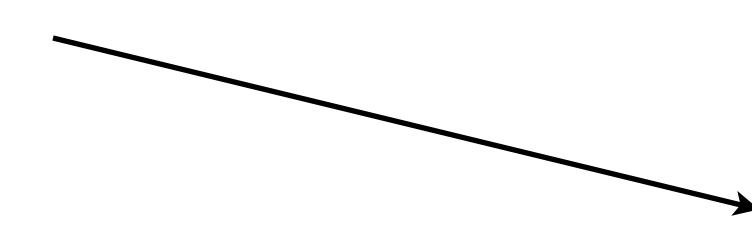
# Closures

- A **closure** is a function (or a reference to function) together with a referencing environment
- A referencing environment is an area where non-local variables are stored and can be accessed
- A function is different from a closure because it cannot access non-local variables (also called *upvalues* of the function)



# Closures

- A **closure** is a function (or a reference to function) together with a referencing environment
- A referencing environment is an area where non-local variables are stored and can be accessed
- A function is different from a closure because it cannot access non-local variables (also called *upvalues* of the function)
- Examples of closures:
  - in Java: local classes, anonymous classes



```
final int rounds = 100;
Runnable runnable = new Runnable() {
 public void run() {
 for (int i = 0; i < rounds; i++) {
 // ...
 }
 }
};
new Thread(runnable).start();
```



# Blocks

- **Blocks** are a language-level extension of C, Objective-C, and C++ introduced to support closures:
  - blocks can access non-local variables from the enclosing scope
  - Blocks are segments of code that can be passed around to methods or functions as if they were values
  - Blocks are Objective-C objects, thus they can be added to **NSArray** or **NSDictionary** instances

[https://developer.apple.com/library/ios/documentation/cocoa/conceptual/Blocks/Articles/00\\_Introduction.html#/apple\\_ref/doc/uid/TP40007502](https://developer.apple.com/library/ios/documentation/cocoa/conceptual/Blocks/Articles/00_Introduction.html#/apple_ref/doc/uid/TP40007502)



# Defining blocks

- A block is defined with a caret (^) and its start and end are marked by curly braces:

```
^{
 // block body ...
 NSLog(@"Within block...");
```

- Blocks can be assigned to variables:

```
block = ^{
 NSLog(@"Within block...");
```

- The above block does not return any value and takes no argument
- A block can be executed, as if it were a C function:

```
block();
```



# Defining blocks

- Blocks can return a value and take arguments, just like functions
- General syntax of a **block signature**:  
`return_type(^block_name)(arguments);`
- Some examples of block definition:

| Block type definition               | Input arguments         | Return type          |
|-------------------------------------|-------------------------|----------------------|
| <code>void(^BlockA)(void);</code>   | -                       | -                    |
| <code>int(^BlockB)(int);</code>     | one <code>int</code>    | one <code>int</code> |
| <code>int(^BlockC)(int,int);</code> | two <code>int</code> s  | one <code>int</code> |
| <code>void(^BlockD)(double);</code> | one <code>double</code> | -                    |



# Defining blocks

- **typedef** can be used to define blocks with the same signature to simplify code

```
typedef int(^ReturnIntBlock)(int);

ReturnIntBlock square = ^(int val){

 return val * val;

};

int a = square(10);

// ...

ReturnIntBlock cube = ^(int val){

 return val * val * val;

};

int b = cube(10);
```



# Working with blocks

```
int(^max)(int, int) = ^(int a, int b){
 return (a > b ? a : b);
};

int maximum = max(first,second);
```



# Accessing variables within blocks

- Blocks can access values from the enclosing scope (blocks are closures)
  - A block literal declared from within a method can access the local variables accessible in the local scope of the method
- 
- Only the value of non-local variables is captured, so inside the block it is unaffected
  - The value of non-local variables cannot be changed (read-only access)

```
- (void)aMethod{
 // ...
 int var = 10;
 void(^example)(void) = ^{
 NSLog(@"Accessing variable %d", var);
 };
 // ...
}
```

```
- (void)aMethod{
 // ...
 int var = 10;
 void(^example)(void) = ^{
 NSLog(@"Accessing variable %d", var);
 };
 // ...
 var = 100;
 example(); // var is still 10
}
```



# Accessing variables within blocks

- If a non-local variable should be changed inside a block, it must be properly marked when it is declared with the `__block` directive
- The `__block` directive also marks the variable for shared storage between the enclosing scope and the block, so its value might change as it is not passed in by value only

```
- (void)aMethod{
 // ...
 __block int var = 10;
 void(^example)(void) = ^{
 NSLog(@"Accessing variable %d", ++var);
 };
 // ...
 var = 100;
 example(); // prints 101
}
```



# Blocks as arguments

- Blocks are objects: they can be passed in as arguments to methods and functions
- Similar to passing `void*`, except that blocks are much more than function pointers
- Typically, blocks are used to create callback methods or execute them in other parts of the code that can be parametrized



# Blocks as arguments

- `(void)downloadFromUrl:(NSURL *)url completion:(void(^)(NSData*))callback;`
- The last argument of this method is a block, which takes a `NSData*` argument and returns nothing, that will be executed as callback
- It is equivalent to the following declaration:

```
typedef void(^DownloadBlock)(NSData*);
```

- `(void)downloadFromUrl:(NSURL *)url completion:(DownloadBlock)callback;`



# Blocks as arguments

- The method's implementation might look like

```
- (void)downloadFromUrl:(NSURL *)url completion:(void(^)(NSData*))callback{
 // download from the URL
 NSData *data = ...;
 // ...
 callback(data);
}
```



# Blocks as arguments

- To invoke the method, we must pass in a `void(^)(NSData*)` block

```
void(^DownloadCompleteHandler)(NSData *) = ^(NSData *data){
 NSLog(@"Download complete [%d bytes]!", [data length]);
};
// ...
[self downloadFromUrl:url completion:DownloadCompleteHandler];
```



# Concurrency

- It is very important to be able to execute many tasks in parallel
- The main thread is the application's thread responsible for handling user-generated events and managing the UI
- A long-running task (or a task that might take an undefined amount of time) should never be performed in the main thread
- Background threads are needed in order to keep the main thread ready for handling event, and therefore to keep the application responsive
- Multiple threads perform different tasks simultaneously
- Threads must be coordinated to avoid that the program reaches an inconsistent state

<https://developer.apple.com/library/ios/documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html>



# Concurrency

- How does iOS handle the execution of many tasks at the same time?
- iOS provides mechanisms to work with threads (UNIX-based)
- Threads are a lightweight low-level feature that can be used to provide concurrency in a program
- However, using threads introduces complexity to programs because of several issues, such as:
  - synchronization
  - scalability
  - shared data structures
- Threads introduce uncertainty to code and overhead at runtime for context switch
- Alternative methods have been introduced in order to progressively eliminate the need to rely on threads



# Threads in iOS

- iOS provides several methods for creating and managing threads
- POSIX threads can be used (C interface)
- The Cocoa framework provides an Objective-C class for threads, **NSThread**

```
[NSThread detachNewThreadSelector:@selector(threadMainMethod:) toTarget:self withObject:nil];
```

- **NSObject** also allows the spawning of threads

```
[obj performSelectorInBackground:@selector(aMethod) withObject:nil];
```
- Using threads is discouraged, iOS provides other technologies that can be used to effectively tackle concurrency in programs



# Alternative to threads

- Instead of relying on threads, iOS adopts an **asynchronous design approach**:
  1. acquire a background thread
  2. start the task on that thread
  3. send a notification to the caller when the task is done (callback function)
- An asynchronous function does some work behind the scenes to start a task running but returns before that task might actually be complete
- iOS provides technologies to perform asynchronous tasks without having to manage the threads directly



# Grand Central Dispatch

- **Grand Dispatch Central** (GCD) is one of the technologies that iOS provides to start tasks asynchronously
- GCD is a C library which moves the thread management code down to the system level
- GCD takes care of creating the needed threads and of scheduling tasks to run on those threads
- GCD is based on **dispatch queues**, which are a C-based mechanism for executing custom tasks



# Dispatch queues

- A dispatch queue executes tasks, either synchronously or asynchronously, always in a FIFO order (a dispatch queue always dequeues and starts tasks in the same order in which they were added to the queue)
- Serial dispatch queues run only one task at a time, each time waiting that the current task completes before executing the next one
- Concurrent dispatch queues can execute many tasks without waiting for the completion of other tasks
- With GCD, tasks to be executed are first defined and then added to an appropriate dispatch queue
- Dispatch queues provide a straightforward and simple programming interface
- The tasks submitted to a dispatch queue must be encapsulated inside either a function or a block object



# Operation queues

- **Operation queues** are an Objective-C equivalent to concurrent dispatch queues provided by Cocoa (implemented by the class **NSOperationQueue**)
- Operation queues do not necessarily take a FIFO ordering, as they take into account also the possible dependencies among operations
- Operations added to an operation queue are instances of the **NSOperation** class
- Operations can be configured to define dependencies, in order to affect the order of their execution

# Working with queues

- Getting the main queue

```
dispatch_queue_t mainQueue = dispatch_get_main_queue();
```

- Creating a queue

```
dispatch_queue_t queue = dispatch_queue_create("queue_name", NULL);
```

- Executing a task defined in a block asynchronously on a queue

```
dispatch_async(queue, block);
```

- Executing a task on the main queue

```
dispatch_async(dispatch_get_main_queue(), block);
```

- If not using ARC, the queue must be released

```
dispatch_release(queue);
```



# Working with queues

```
dispatch_queue_t queue = dispatch_queue_create("queue_name", NULL);

dispatch_async(queue, ^{
 // long running task...

 dispatch_async(dispatch_get_main_queue(), ^{
 // update UI in main thread...
 });
});
```



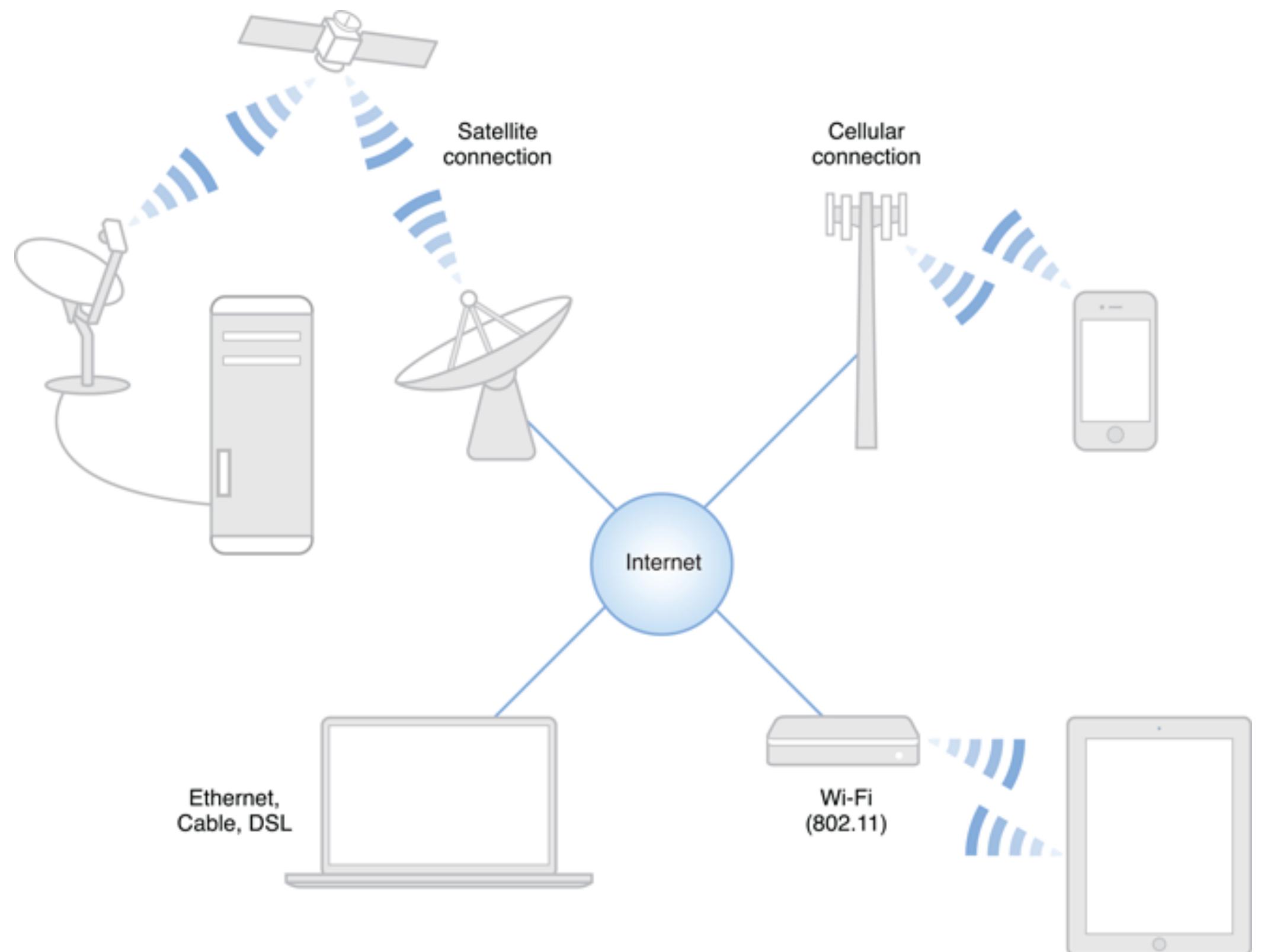
# Networking

- iOS provides APIs for networking at many levels, which allow developers to:
  - perform HTTP/HTTPS requests, such as GET and POST requests
  - establish a connection to a remote host, with or without encryption or authentication
  - listen for incoming connections
  - send and receive data with connectionless protocols
  - publish, browse, and resolve network services with Bonjour

[https://developer.apple.com/library/ios/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/Introduction/Introduction.html#/apple\\_ref/doc/uid/TP40010220-CH12-BBCFIHFH](https://developer.apple.com/library/ios/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/Introduction/Introduction.html#/apple_ref/doc/uid/TP40010220-CH12-BBCFIHFH)

# Networking

- Many applications rely on data and services provided by remote hosts, thus networking plays a central role in today's app development
- As a general rule, networking should be dynamic and asynchronous... and NOT on the main thread!
- Connectivity status might change abruptly, so apps should be designed to be robust to maintain usability and user experience
- It is a bad design to make assumptions on the speed of the connection (even though the type of connection may be desumed)





# NSURL

- **NSURL** objects are used to manipulate URLs and the resources they reference
- **NSURL** objects are also used to refer to a file
- Typically, an instance of **NSURL** can be created with either of the following methods:

```
NSURL *aUrl = [NSURL URLWithString:@"http://mobdev.ce.unipr.it"];
```

```
NSURL *bUrl = [[NSURL alloc] initWithString:@"http://mobdev.ce.unipr.it"];
```

```
NSURL *cUrl = [NSURL URLWithString:@"/path" relativeToURL:aURL];
```



# Loading URLs to NSString and NSData

- **NSString** and **NSData** instances can be created with information coming from a remote URL:

```
NSURL *url = ...;
NSError *error;

NSString *str = [NSString stringWithContentsOfURL:url encoding:NSUTF8StringEncoding error:&error];

NSData *data = [NSData dataWithContentsOfURL:url options:NSDataReadingUncached error:&error];
```

- **stringWithContentsOfURL:encoding:error:** returns a string created by reading data from a given URL interpreted using a given encoding
- **dataWithContentsOfURL:options:error:** creates and returns a data object containing the data from the location specified by the given URL
- These methods are synchronous, so it is better to execute them inside a background dispatch queue to keep an asynchronous design



# NSURLRequest

- **NSURLRequest** objects represent a URL load request independent of protocol and URL scheme
- The mutable subclass of **NSURLRequest** is **NSMutableURLRequest**
- An **NSURLRequest** is constructed with a given **NSURL** argument:

```
NSURLRequest *req = [NSURLRequest requestWithURL:url];
```

```
NSURLRequest *req2 = [[NSURLRequest alloc] initWithURL:url];
```

- **NSMutableURLRequest** instances can be configured to set protocol independent properties and HTTP-specific properties:
  - URL, timeout
  - HTTP method, HTTP body, HTTP headers



# NSURLConnection

- NSURLConnection objects provide support to perform the loading of a URL request
- An NSURLConnection can load a URL request either synchronously or asynchronously
- Synchronous URL loading:

```
NSError *error;
NSURLResponse *response = nil;
NSData *data = [NSURLConnection sendSynchronousRequest:req
 returningResponse:&response
 error:&error];
```

- Asynchronous URL loading:

```
[NSURLConnection
 sendAsynchronousRequest:req
 queue:[NSOperationQueue currentQueue]
 completionHandler:^(NSURLResponse *response, NSData *data, NSError *connectionError) {
 // ...
 }
];
```



# NSURLConnectionDelegate

- The **NSURLConnectionDelegate** protocol provides methods for receive informational callbacks about the asynchronous load of a URL request
- Delegate methods are called on the thread that started the asynchronous load operation for the associated **NSURLConnection** object
- **NSURLConnection** with specified delegate:

```
NSURLConnection *conn = [[NSURLConnection alloc] initWithRequest:req delegate:self];
```

- Starting and stopping the loading of a URL with **NSURLConnection**:

```
[conn start];
```

```
[conn cancel];
```



# NSURLConnectionDelegate

- The **NSURLConnectionDelegate** and **NSURLConnectionDataDelegate** protocols define methods that should be implemented by the delegate for an instance of **NSURLConnection**
- In order to keep track of the progress of the URL loading the protocol methods should be implemented
- ```
(void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data{
    NSLog(@"%@", @"received %ld bytes", [data length]);
}

-(void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)response{
    NSLog(@"%@", @"received response");
}

-(void)connectionDidFinishLoading:(NSURLConnection *)connection{
    NSLog(@"%@", @"loading finished");
}

-(void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error{
    NSLog(@"%@", @"loading failed");
}
```



NSURLSession

- The **NSURLSession** class and related classes provide an API for downloading content via HTTP
- This API provides delegate methods for:
 - supporting authentication
 - giving the app the ability to perform background downloads when the app is not running or while your app is suspended
- The **NSURLSession** API is highly asynchronous: a completion handler block (that returns data to the app when a transfer finishes successfully or with an error) must be provided
- The **NSURLSession** API provides status and progress properties, in addition to delivering this information to delegates
- It supports canceling, restarting (resuming), and suspending tasks, and it provides the ability to resume suspended, canceled, or failed downloads where they left off
- The behavior of a **NSURLSession** depends on its session type and its session task



Types of sessions

- There are three types of sessions, as determined by the type of configuration object used to create the session:
 1. **Default sessions** use a persistent disk-based cache and store credentials in the user's keychain
 2. **Ephemeral sessions** do not store any data to disk; all caches, credential stores, and so on are kept in RAM and tied to the session; when your app invalidates the session, they are purged automatically
 3. **Background sessions** are similar to default sessions, except that a separate process handles all data transfers
- Another kind of session exists: **shared session** is a singleton NSURLConnection which cannot be configured and can be used for basic requests. It can be retrieved using the class method:

`[NSURLSession sharedSession]`



Types of tasks

- There are three types of tasks supported by a session:
 1. **Data tasks** send and receive data using NSData objects
 - data tasks are intended for short, often interactive requests from your app to a server
 - data tasks can return data to your app one piece at a time after each piece of data is received, or all at once through a completion handler
 - data tasks do not store the data to a file, so they are not supported in background sessions
 2. **Download tasks** retrieve data in the form of a file, and support background downloads while the app is not running
 3. **Upload tasks** send data (usually in the form of a file), and support background uploads while the app is not running

Shared NSURLConnection examples

Data task

```
NSURLSessionTask *task =  
    [[NSURLSession sharedSession] dataTaskWithRequest:req  
                                                completionHandler:^(NSData * data, NSURLResponse * response, NSError * error){  
  
        NSLog(@"%@", [NSString stringWithUTF8String:[data bytes]]);  
  
    }];  
  
[task resume];
```

Download task

```
NSURLSessionTask *task =  
    [[NSURLSession sharedSession] downloadTaskWithRequest:req  
                                                completionHandler:^(NSURL *location, NSURLResponse *response, NSError *error){  
  
        NSLog(@"%@", location.absoluteURL);  
  
    }];  
  
[task resume];
```



Working with NSURLConnection

- The **NSURLSession** instance can be created to execute its completion handler block in the main queue or in another queue



Working with NSURLConnection

- The **NSURLSession** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **in the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];

NSURLSession *session = [NSURLSession sessionWithConfiguration:conf
                                                       delegate:nil
                                                       delegateQueue:[NSOperationQueue mainQueue]];

NSURLSessionDownloadTask *task;

task = [session
        downloadTaskWithRequest:request
        completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {

            /* this block is executed on the main queue */
            /* UI-related code ok, but avoid long-running operations */
    }];

[task resume];
```

Working with NSURLConnection

- The **NSURLSession** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **in the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];  
  
NSURLSession *session = [NSURLSession sessionWithConfiguration:conf  
delegate:nil  
delegateQueue:[NSOperationQueue mainQueue]];  
  
NSURLSessionDownloadTask *task;  
task = [session  
downloadTaskWithRequest:request  
completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {  
    /* this block is executed on the main queue */  
    /* UI-related code ok, but avoid long-running operations */  
}  
];  
  
[task resume];
```

setting the main queue as the delegate queue to execute the completion handler



Working with NSURLConnection

- The **NSURLSession** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **in the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];

NSURLSession *session = [NSURLSession sessionWithConfiguration:conf
                                                       delegate:nil
                                                       delegateQueue:[NSOperationQueue mainQueue]];

NSURLSessionDownloadTask *task;

task = [session
        downloadTaskWithRequest:request
        completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {

            /* this block is executed on the main queue */
            /* UI-related code ok, but avoid long-running operations */
        }
];

[task resume];
```



Working with NSURLConnection

- The **NSURLSession** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **off the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];  
  
NSURLSession *session = [NSURLSession sessionWithConfiguration:conf];  
  
NSURLSessionDownloadTask *task;  
  
task = [session  
    downloadTaskWithRequest:request  
    completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {  
  
        /* non-UI-related code here (not in the main thread) */  
  
        dispatch_async(dispatch_get_main_queue(), ^{  
            /* UI-related code (in the main thread) */  
        });  
    }];
```



Working with NSURLConnection

- The **NSURLSession** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **off the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];  
  
NSURLSession *session = [NSURLSession sessionWithConfiguration:conf];  
  
NSURLSessionDownloadTask *task; no delegate queue  
  
task = [session  
        downloadTaskWithRequest:request  
        completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {  
  
            /* non-UI-related code here (not in the main thread) */  
  
            dispatch_async(dispatch_get_main_queue(), ^{  
                /* UI-related code (in the main thread) */  
            });  
        }];
```

Working with NSURLConnection

- The **NSURLSession** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **off the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];  
  
NSURLSession *session = [NSURLSession sessionWithConfiguration:conf];  
  
NSURLSessionDownloadTask *task;  
  
task = [session  
    downloadTaskWithRequest:request  
    completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {  
        /* non-UI-related code here (not in the main thread) */  
        dispatch_async(dispatch_get_main_queue(), ^{  
            /* UI-related code (in the main thread) */  
        });  
    }];
```

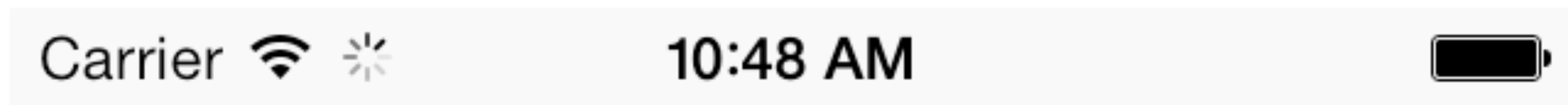


Sockets and streams

- **NSStream** is an abstract class for objects representing streams
- **NSStream** objects provide a common way to read and write data to and from:
 - memory
 - files
 - network (using sockets)
- **NSInputStream** are used to read bytes from a stream, while **NSOutputStream** are used to write bytes to a stream
- With streams it is possible to write networking code that works at the TCP level, instead of application level (such as HTTP/HTTPS URL loading)

Network Activity Indicator

- When performing network-related tasks (typically, for more than a couple of seconds), such as uploading or downloading contents, feedback should be provided to the user
- A **network activity indicator** appears in the status bar and shows that network activity is occurring

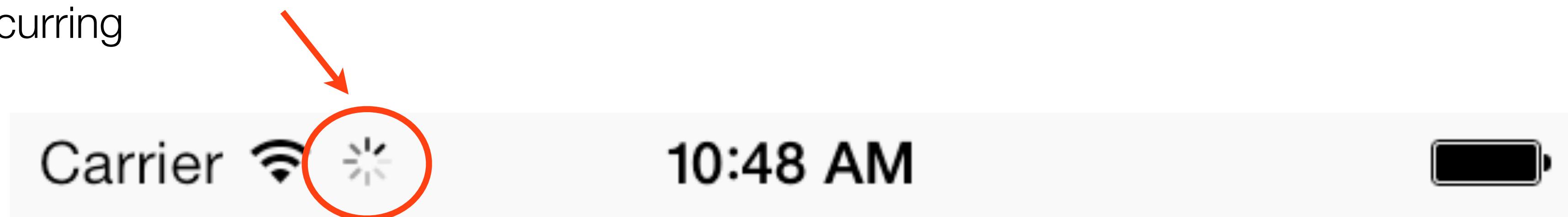


- Use the `UIApplication` method `networkActivityIndicatorVisible` to control the indicator's visibility:

```
[UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
```

Network Activity Indicator

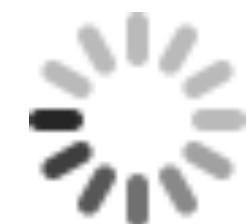
- When performing network-related tasks (typically, for more than a couple of seconds), such as uploading or downloading contents, feedback should be provided to the user
- A **network activity indicator** appears in the status bar and shows that network activity is occurring



- Use the `UIApplication` method `networkActivityIndicatorVisible` to control the indicator's visibility:

```
[UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
```

UIActivityIndicatorView



- An activity indicator is a spinning wheel that indicates a task is being executed
- An activity indicator should be displayed to provide feedback to the user that your app is not stalled or frozen, typically when performing long-running tasks (CPU-intensive or network tasks)
- The activity indicator view is provided by the **UIActivityIndicatorView** class
- A **UIActivityIndicatorView** can be started and stopped using the **startAnimating** and **stopAnimating** methods
- The **hidesWhenStopped** property can be used to configure whether the receiver is hidden when the animation is stopped
- A **UIActivityIndicatorView**'s appearance can be customized with the **activityIndicatorViewStyle** or **color** properties



UIImage

- The **UIImage** class is used to display image data
- Images can be created from files (typically PNG and JPEG files) and (network) data

```
UIImage *imgA = [UIImage imageNamed:@"image.png"];
```

```
NSData *data = ...;
UIImage *imgB = [UIImage imageWithData:data];
```

```
UIImage *imgC = [UIImage imageWithContentsOfFile:@"/path/to/file"];
```

- Image objects are immutable
- Supported image file formats: TIFF, JPEG, PNG, GIF
- Image properties: **imageOrientation**, **scale**, **size**

UIImageView

- A **UIImageView** displays an image or an animated sequence of images
- To display a single image, the image view must be configured with the appropriate **UIImage**
- For multiple images, also the animations among images must be configured
- A UIImageView object can be dragged into a view using the object palette
- The image to be displayed can be configured in storyboard
- The image property can be used to set programmatically the **image** property





UIImageView

- Image views can perform expensive operations that can affect the performance of the application, such as scaling the image and setting the transparency to mix the image with lower layers
- Possible countermeasures that can have a positive impact on performance are:
 - provide pre-scaled images where possible (e.g. thumbnails for previews)
 - limit image size (e.g. pre-scaling or tiling large images)
 - disable alpha blending except where needed



Mobile Application Development



Lecture 6
Blocks, Concurrency, Networking



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).



Mobile Application Development



Lecture 7
ScrollView, TableView, WebView



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).

Lecture Summary

- Scroll views
- Table views and table view cells
- Web views





Scroll views

- **Scroll views** allow users to see content that is larger than the view's boundaries
- Vertical or horizontal scroll indicators show users that there is more content
- Scroll views are implemented in the **UIScrollView** class
- **UIScrollView** also enables users:
 - to scroll within that content by making swiping gestures
 - to zoom in and back from portions of the content by making pinching gestures
- A **UIScrollView** is a view whose origin is adjustable over the content view
- A scroll view clips the content to its frame
- The content of a scroll view is one or more views of any kind



Adding content to scroll views

- The steps to work with a scroll view are the following:

1. Add a view to a scroll view:

```
UIImageView *imageView = [[UIImageView alloc] initWithImage:image];  
[scrollView addSubview:imageView];
```

2. Set the size of the scroll view's content:

```
scrollView.contentSize = CGSizeMake(3000, 3000);
```

```
// or ...
```

```
scrollView.contentSize = imageView.bounds.size;
```



Adding content to scroll views

- The **contentOffset** property returns the upper left corner of the currently portion of the view that is being displayed relatively to the scroll view

`scrollView.contentOffset.x, scrollView.contentOffset.y`

- The **bounds** property returns the rectangle which is currently visible

`scrollView.bounds.origin.x, scrollView.bounds.origin.y`

`scrollView.bounds.size.width, scrollView.bounds.size.height`

- The visible area in the subview's coordinates may be different from that returned by the bounds property because of scaling (zooming), different subview's frame origin

`CGRect visibleRect = [scrollView convertRect:scrollView.bounds toView:subview];`



Adding scroll views

- The recommended way to use scroll views is to add to a view by dragging it from the object palette and then add the subviews programmatically (with `addSubview:`)
- The scroll view can also be inserted in code with alloc/init
- It is also possible to embed an existing view from (Editor → Embed in... → Scroll View), but you should not use this option
- All the subviews added to a scroll view are automatically located at the origin (upper left corner) of the scroll view; to change the location, a new frame for the view should be set

```
subview.frame = CGRectMake(200, 200, 100, 100);
```
- It is essential to set the `contentSize` property of the scroll view, or it will not work!
- To have a scroll view to be of exactly of the same size of its subview

```
scrollView.contentSize = subview.bounds.size;
```



Working with scroll views

- Scroll views natively support the scrolling made by the user but it is also possible to scroll programmatically:
 - `(void)scrollRectVisible:(CGRect)rect animated:(BOOL)animated`
- It is possible to enable and disable scrolling using the `scrollEnabled` property
- The `directionalLockEnabled` property can be used to lock the scrolling direction to the user's first move (for the current drag)

Zooming in scroll views

- Scroll views support zooming of the content by pinching-in or pinching-out
- To enable zooming in a scroll view some preliminary actions must be taken:
 1. set the minimum and maximum zoom scale:

```
self.scrollView.minimumZoomScale = 0.5; // 50%
self.scrollView.maximumZoomScale = 3.0; // 300%
```



Zooming in scroll views

- Scroll views support zooming of the content by pinching-in or pinching-out
- To enable zooming in a scroll view some preliminary actions must be taken:
 1. set the minimum and maximum zoom scale:

```
self.scrollView.minimumZoomScale = 0.5; // 50%
self.scrollView.maximumZoomScale = 3.0; // 300%
```

2. declare the view controller as **UIScrollViewDelegate** and set it as **delegate** of the scroll view

```
@interface MyViewController : UIViewController<UIScrollViewDelegate>

// or ...

@interface MyViewController ()<UIScrollViewDelegate>

self.scrollView.delegate = self;
```



Zooming in scroll views

- Scroll views support zooming of the content by pinching-in or pinching-out
- To enable zooming in a scroll view some preliminary actions must be taken:
 1. set the minimum and maximum zoom scale:

```
self.scrollView.minimumZoomScale = 0.5; // 50%
self.scrollView.maximumZoomScale = 3.0; // 300%
```



2. declare the view controller as **UIScrollViewDelegate** and set it as **delegate** of the scroll view

```
@interface MyViewController : UIViewController<UIScrollViewDelegate>
```

```
// or ...
```

```
@interface MyViewController ()<UIScrollViewDelegate>
```

```
self.scrollView.delegate = self;
```

3. implement the delegate method to specify the subview that must be zoomed:

```
- (UIView *)viewForZoomingInScrollView:(UIScrollView *)sender
```



UIScrollViewDelegate

- The **UIScrollViewDelegate** protocol provides several methods to handle scrolling, zooming, and dragging events in the scroll view:
 - `(void)scrollViewDidScroll:(UIScrollView *)scrollView`
 - `(void)scrollViewDidZoom:(UIScrollView *)scrollView`
 - `(void)scrollViewWillBeginZooming:(UIScrollView *)scrollView withView:(UIView *)view`
 - `(void)scrollViewDidEndZooming:(UIScrollView *)scrollView withView:(UIView *)view atScale:(CGFloat)scale`
 - `(void)scrollViewWillBeginDragging:(UIScrollView *)scrollView`
 - `(void)scrollViewDidEndDragging:(UIScrollView *)scrollView willDecelerate:(BOOL)decelerate`



Table views

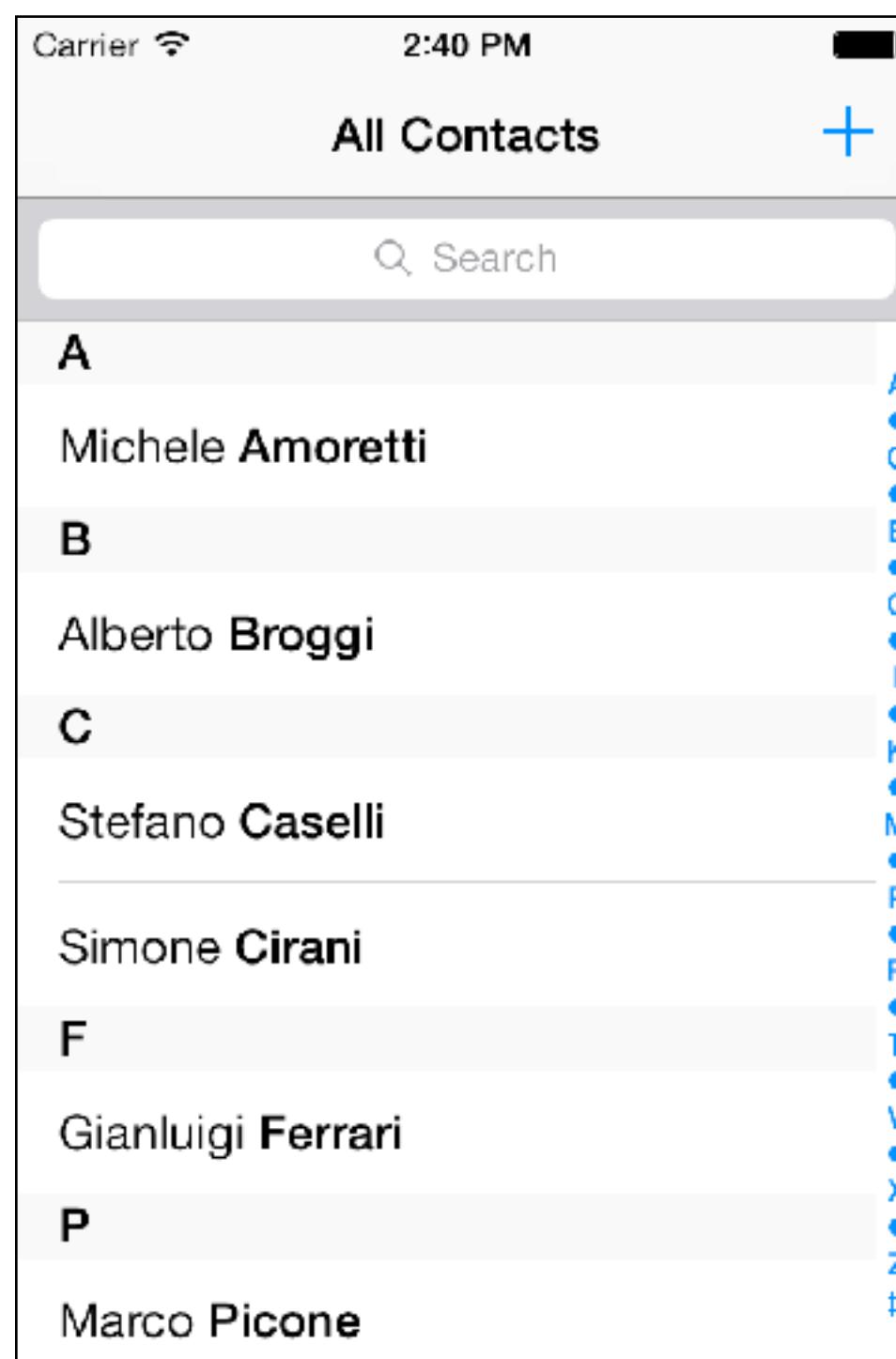
- **Table views** are used to display and edit hierarchical lists of information
- A table view displays a list of items in a single column (one dimensional: just rows, no columns)
- Multiple rows may be grouped into sections
- Multi-dimensional data are usually displayed combining table views into a navigation controller
- **UITableView** is the class that implements a table view
 - it is a subclass of **UIScrollView**
 - only vertical scrolling is permitted
- **UITableView** can efficiently handle large amounts of data



Table view styles

- A table view can have a plain style or a grouped style

UITableViewStylePlain



UITableViewStyleGrouped

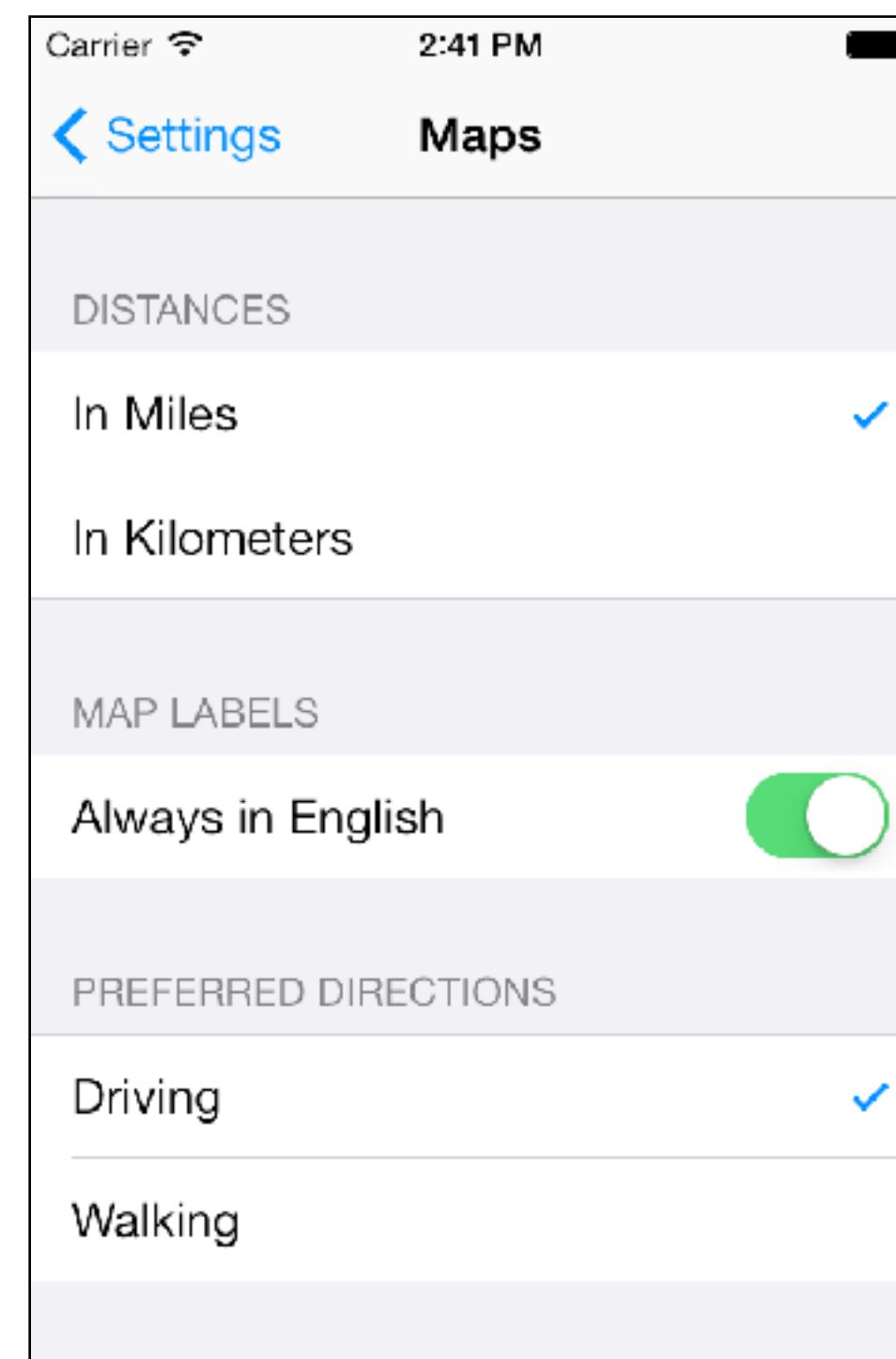
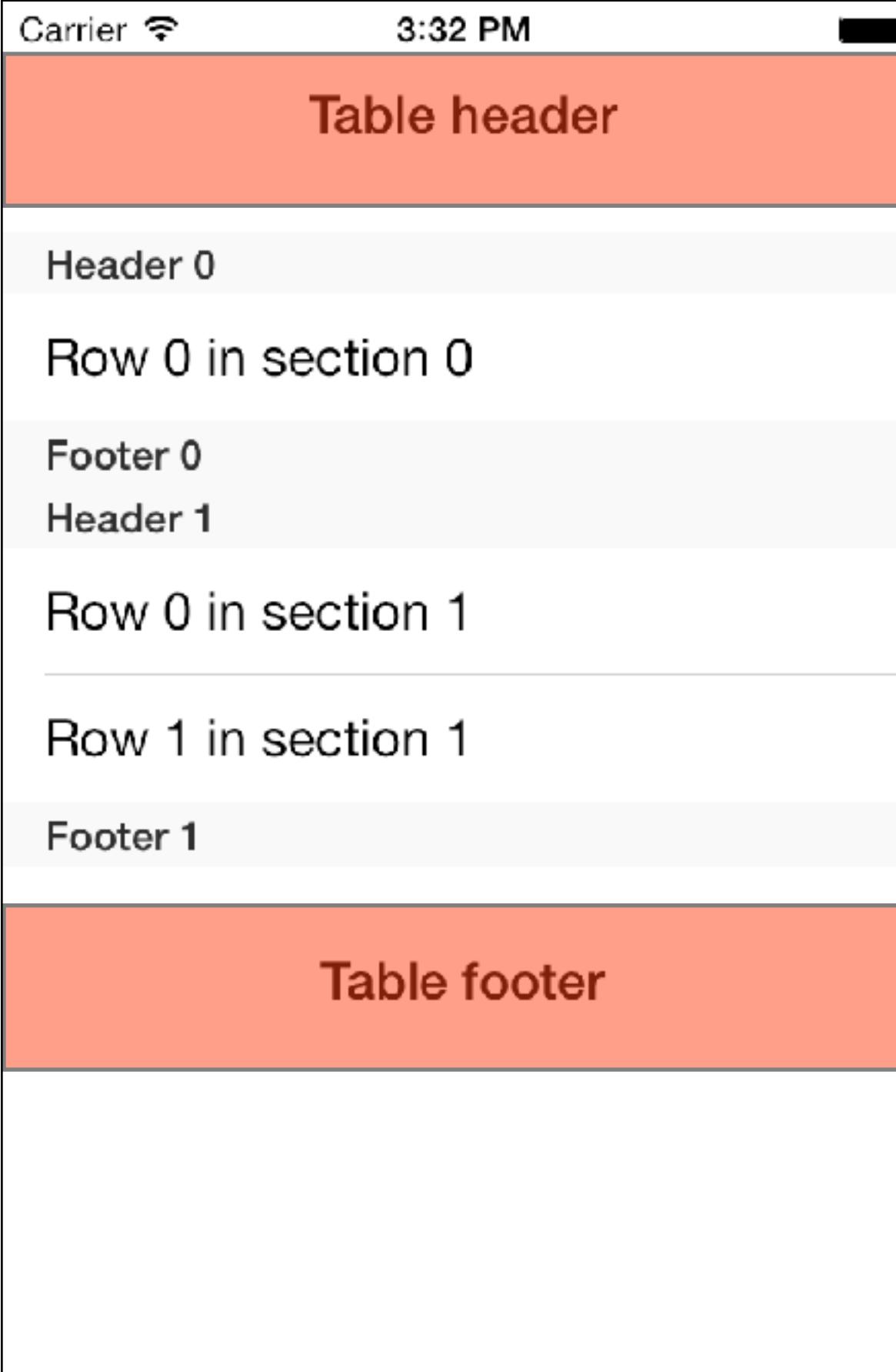
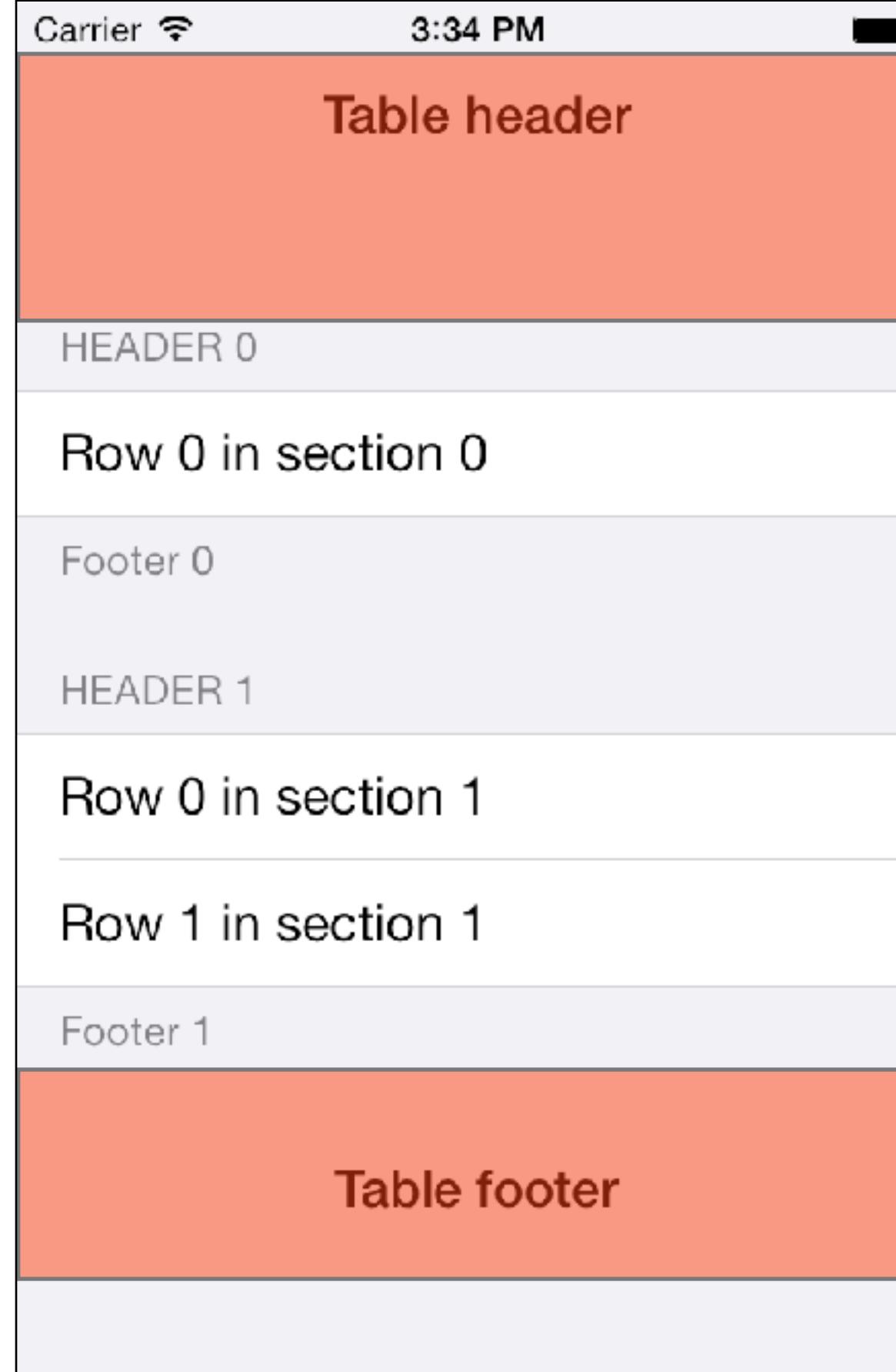




Table view styles

- A table view can be dynamic or static
- Dynamic table views have content that is mutable and unpredictable (e.g. database entries, RSS feeds, ...)
- Static table views have fixed content (defined by the developer), e.g. for settings

UITableView

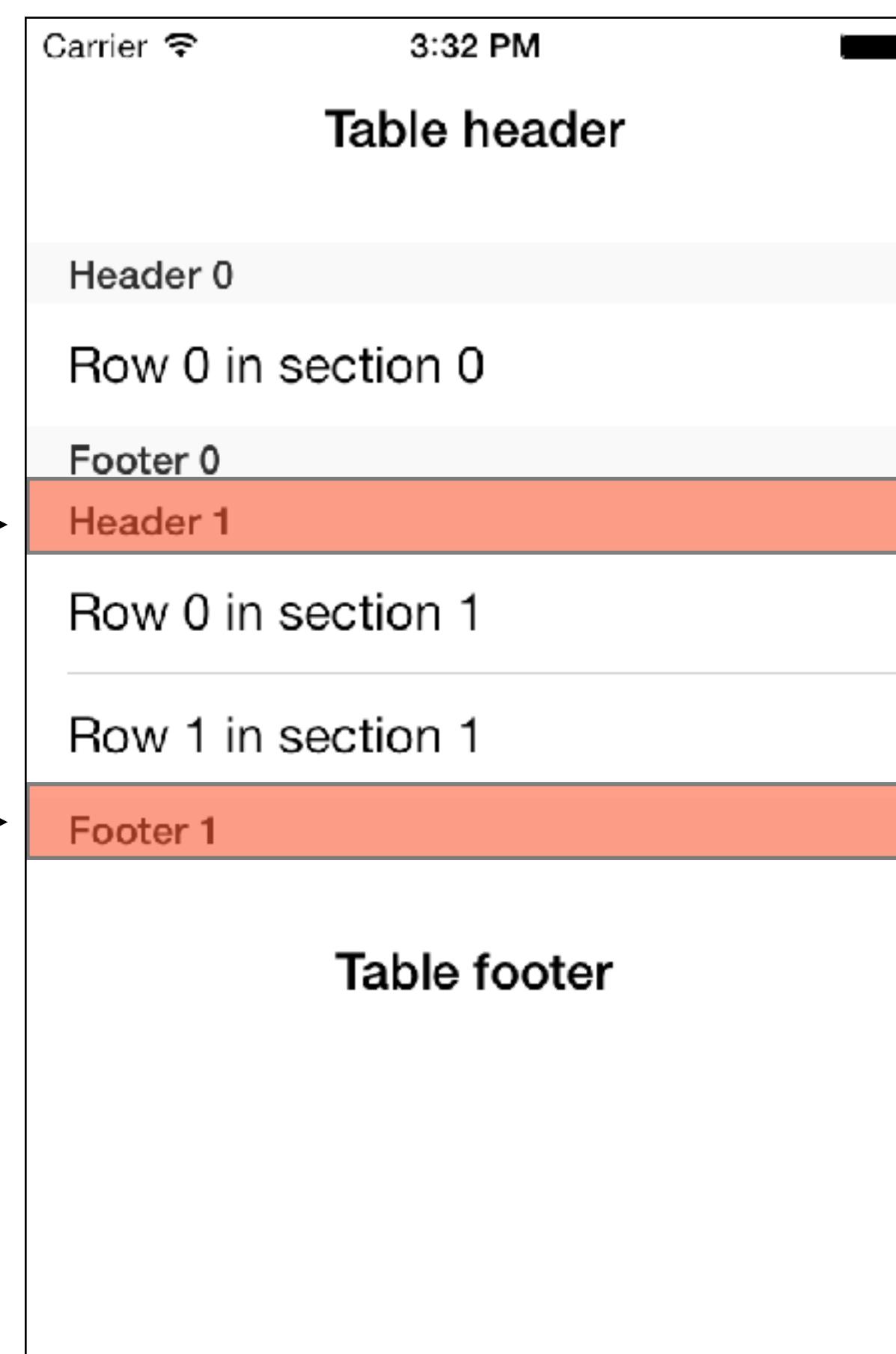
Table header	UITableViewStylePlain	Table footer
<pre>@property UIView *tableHeaderView;</pre>	 <p>The screenshot shows a UITableView with the following structure:</p> <ul style="list-style-type: none">Table header: A solid orange header row labeled "Table header".Rows: Two sections of data.<ul style="list-style-type: none">Section 0: Contains a light gray header row "Header 0", a white data row "Row 0 in section 0", a light gray footer row "Footer 0", a white header row "Header 1", a white data row "Row 0 in section 1", a light gray footer row "Footer 1", and a light gray header row "Header 2".Section 1: Contains a white data row "Row 1 in section 1" and a light gray footer row "Footer 2".Table footer: A solid orange footer row labeled "Table footer".	 <p>The screenshot shows a UITableView with the following structure:</p> <ul style="list-style-type: none">Table header: A solid orange header row labeled "Table header".Rows: Two sections of data.<ul style="list-style-type: none">Section 0: Contains a light gray header row "HEADER 0", a white data row "Row 0 in section 0", a light gray footer row "Footer 0", a white header row "HEADER 1", a white data row "Row 0 in section 1", a light gray footer row "Footer 1", and a light gray header row "Header 2".Section 1: Contains a white data row "Row 1 in section 1" and a light gray footer row "Footer 2".Table footer: A solid orange footer row labeled "Table footer".



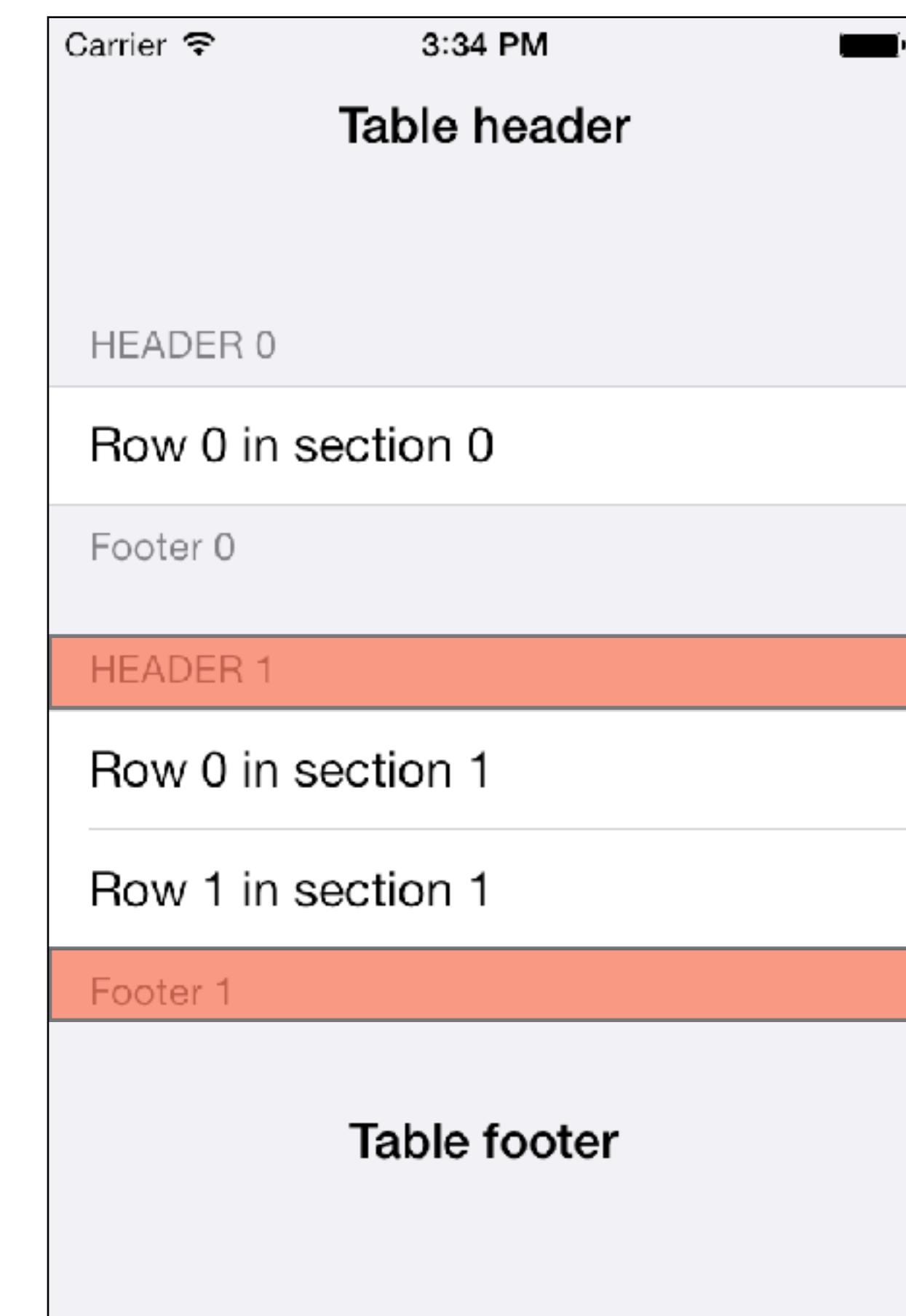
UITableView

UITableViewStylePlain

Section header →
Section footer →



UITableViewStyleGrouped



UITableView

UITableViewController

UITableViewController

Table cell
(UITableViewCell)

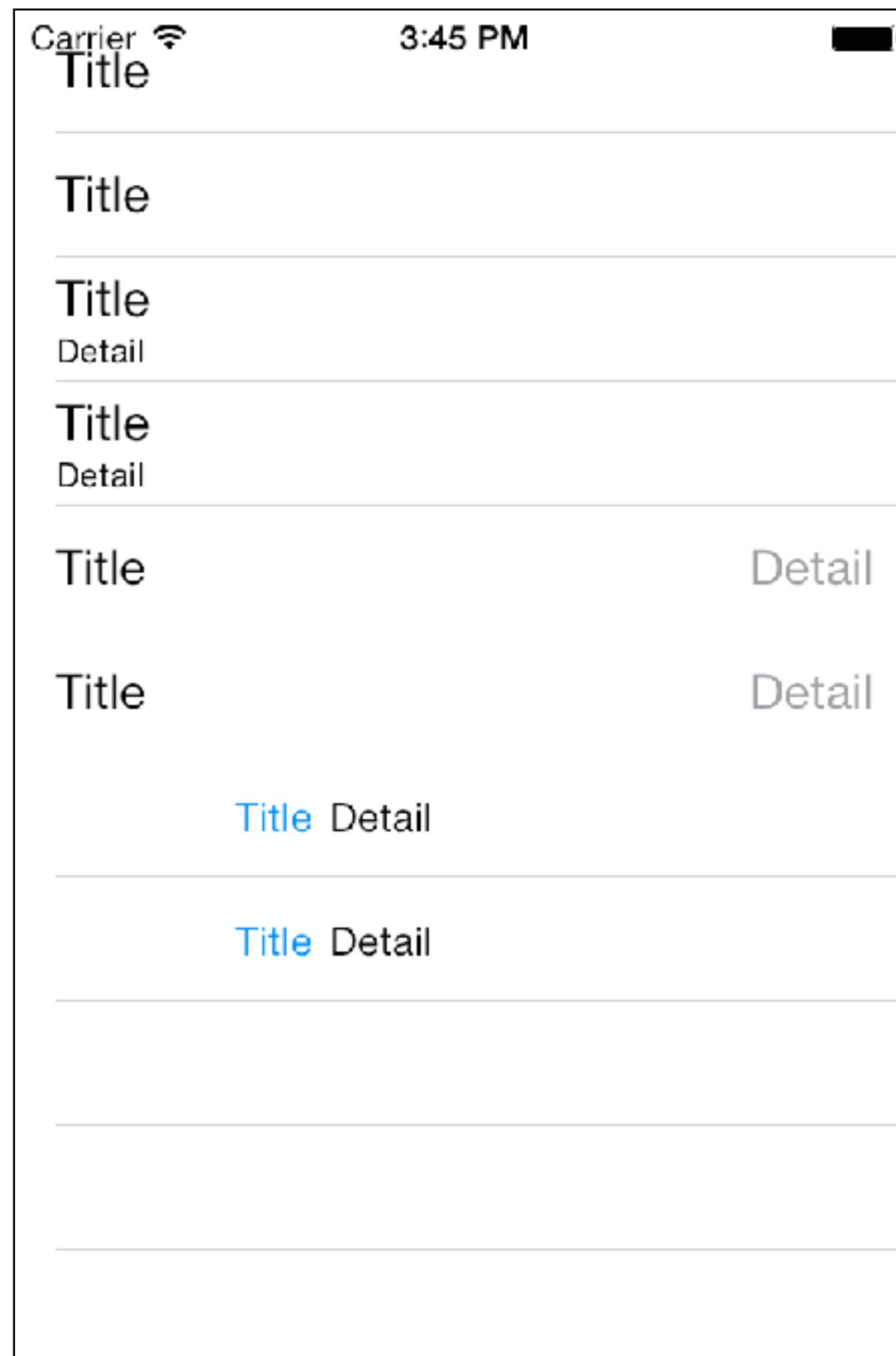
→

Section	Header	Footer
0	Header 0	Footer 0
0	Row 0 in section 0	
1	Header 1	
1	Row 0 in section 1	
1	Row 1 in section 1	
1	Footer 1	

UITableViewController

Section	Header	Footer
0	HEADER 0	
0	Row 0 in section 0	
1	Footer 0	
1	HEADER 1	
1	Row 0 in section 1	
1	Row 1 in section 1	
1	Footer 1	

Table cell styles

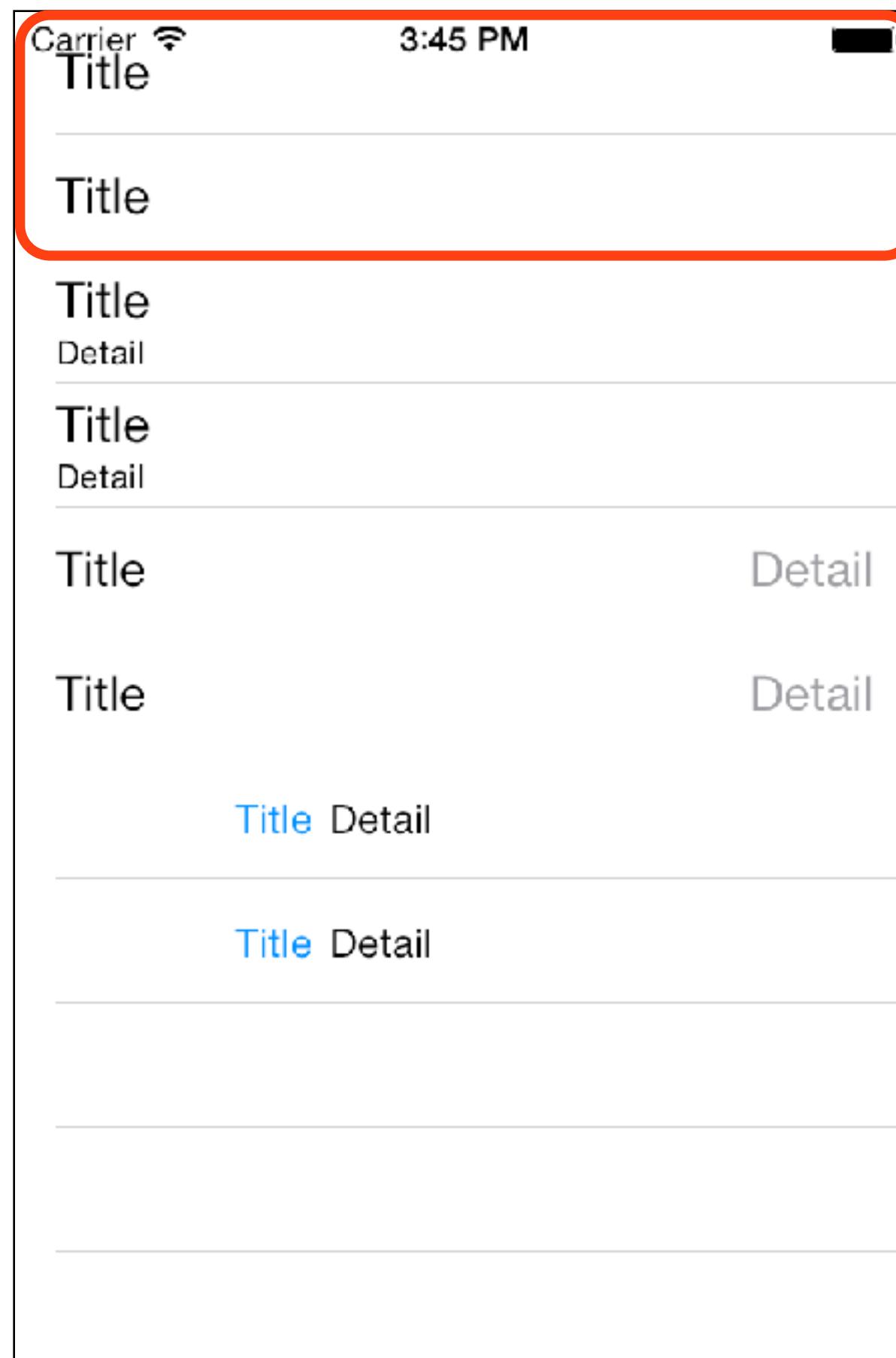


System-defined table cells can have different styles, depending on which content they are intended to display.

Table cells can be customized to have a different look in order to display custom content.



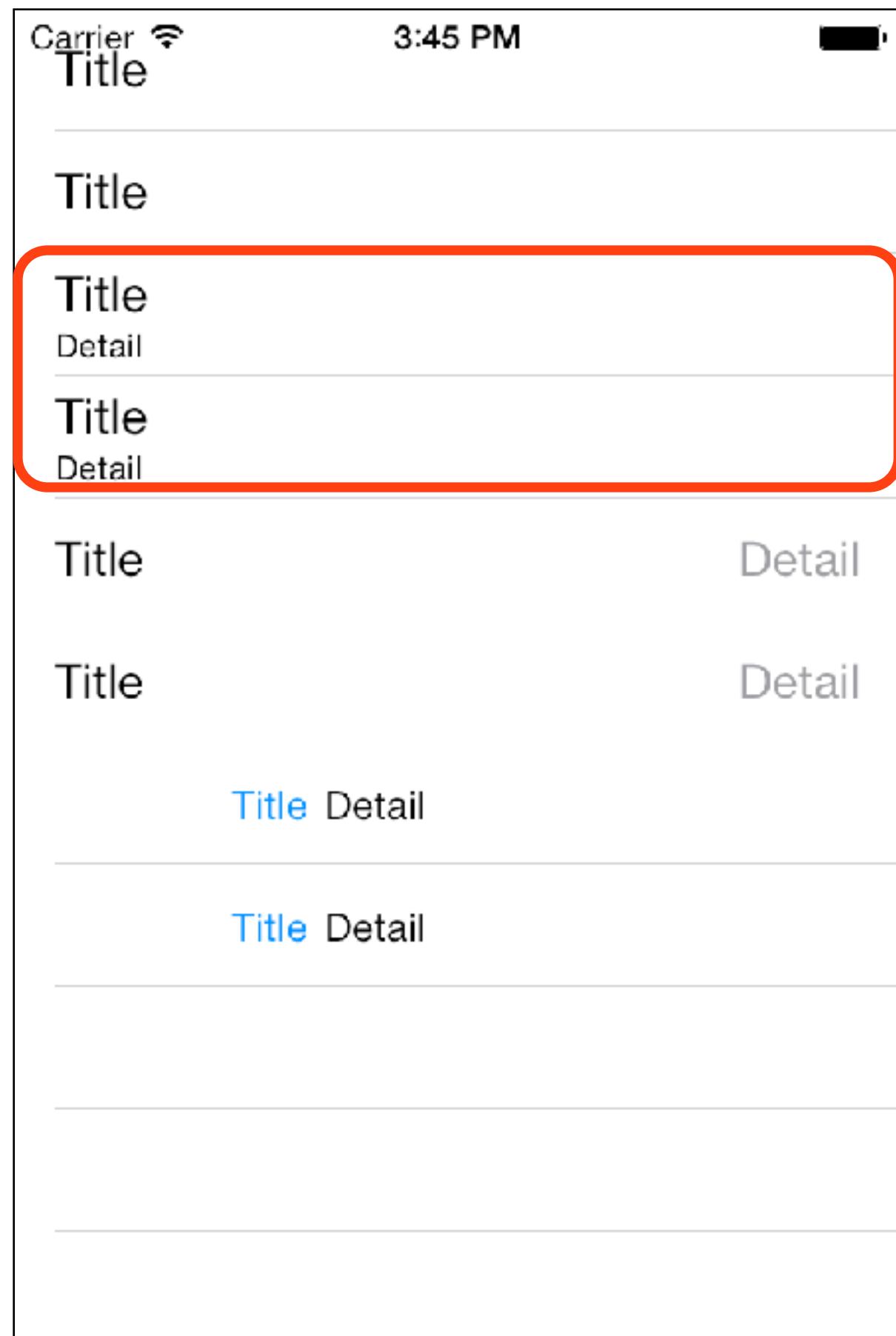
Table cell styles



UITableViewControllerStyleDefault



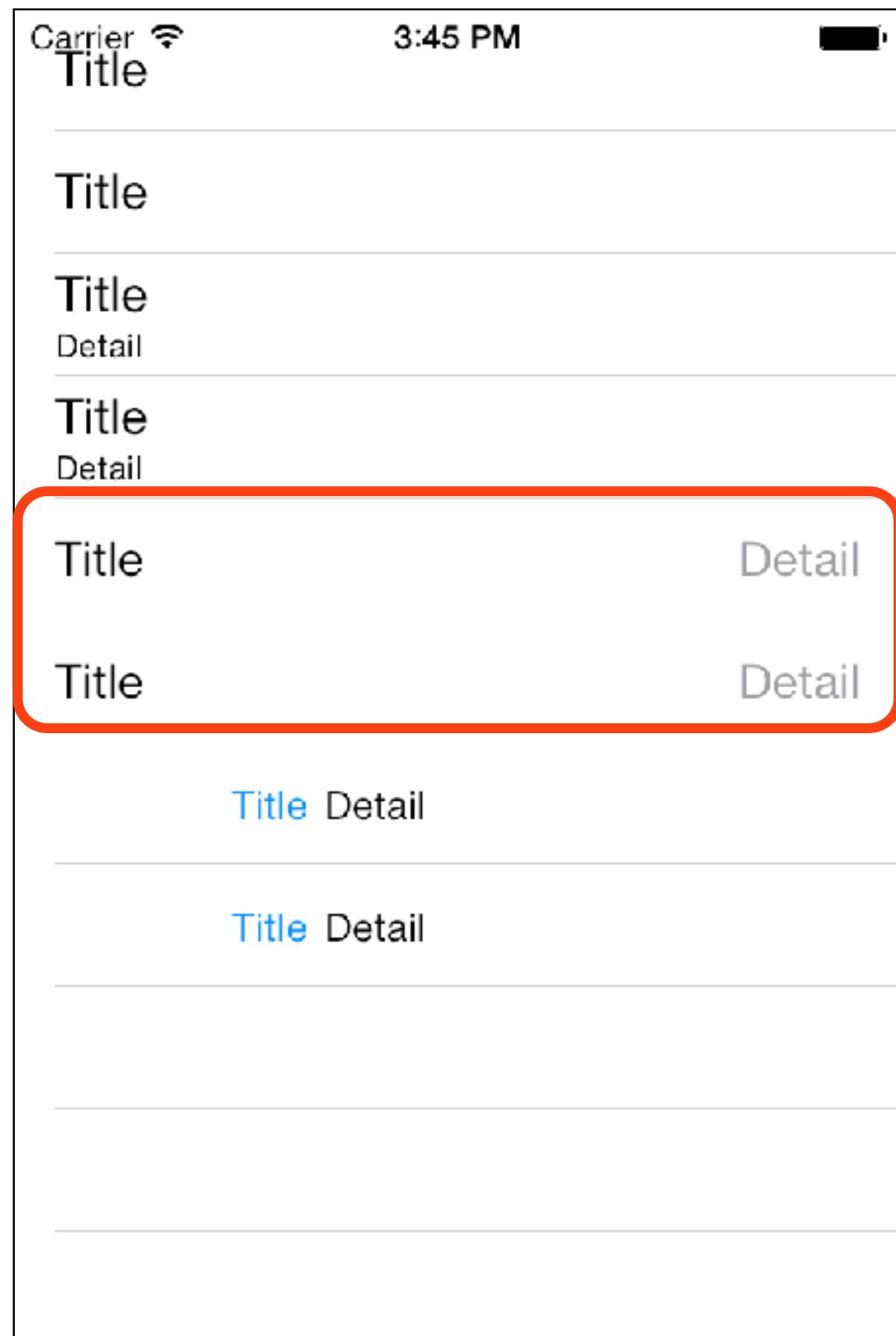
Table cell styles



UITableViewControllerStyleSubtitle

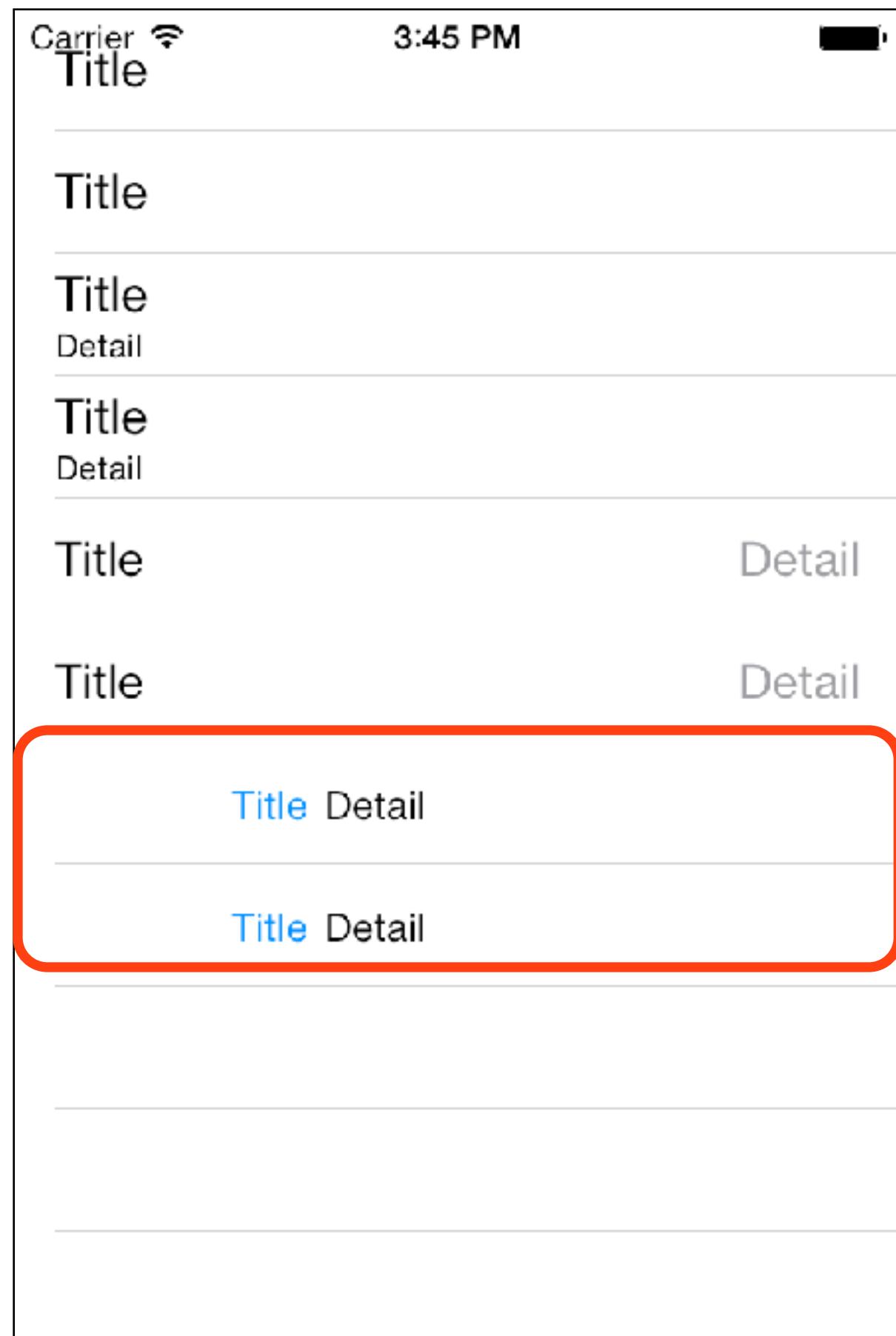


Table cell styles



UITableViewControllerStyleValue1

Table cell styles



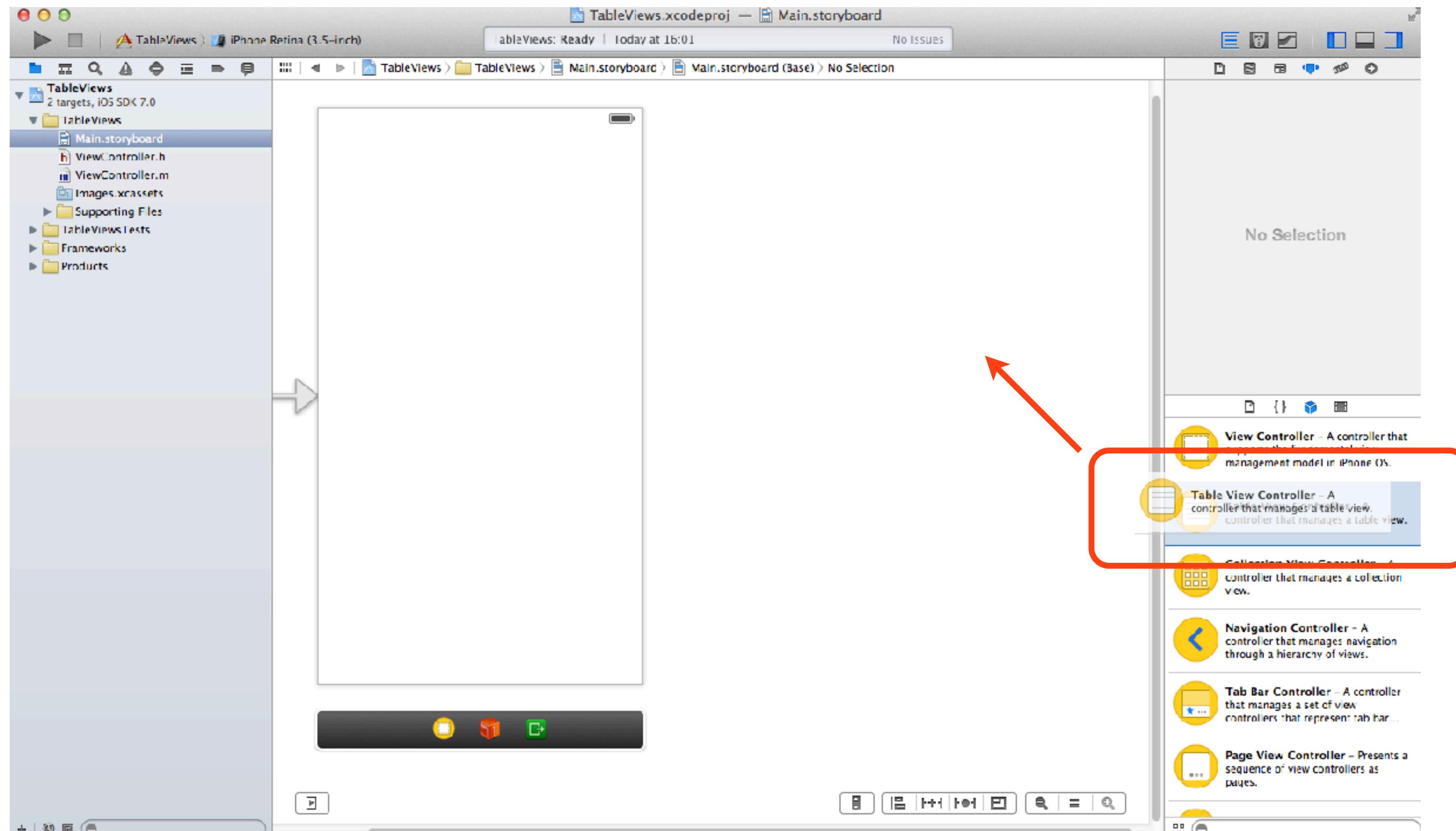
UITableViewControllerStyleValue2



Table View Controller

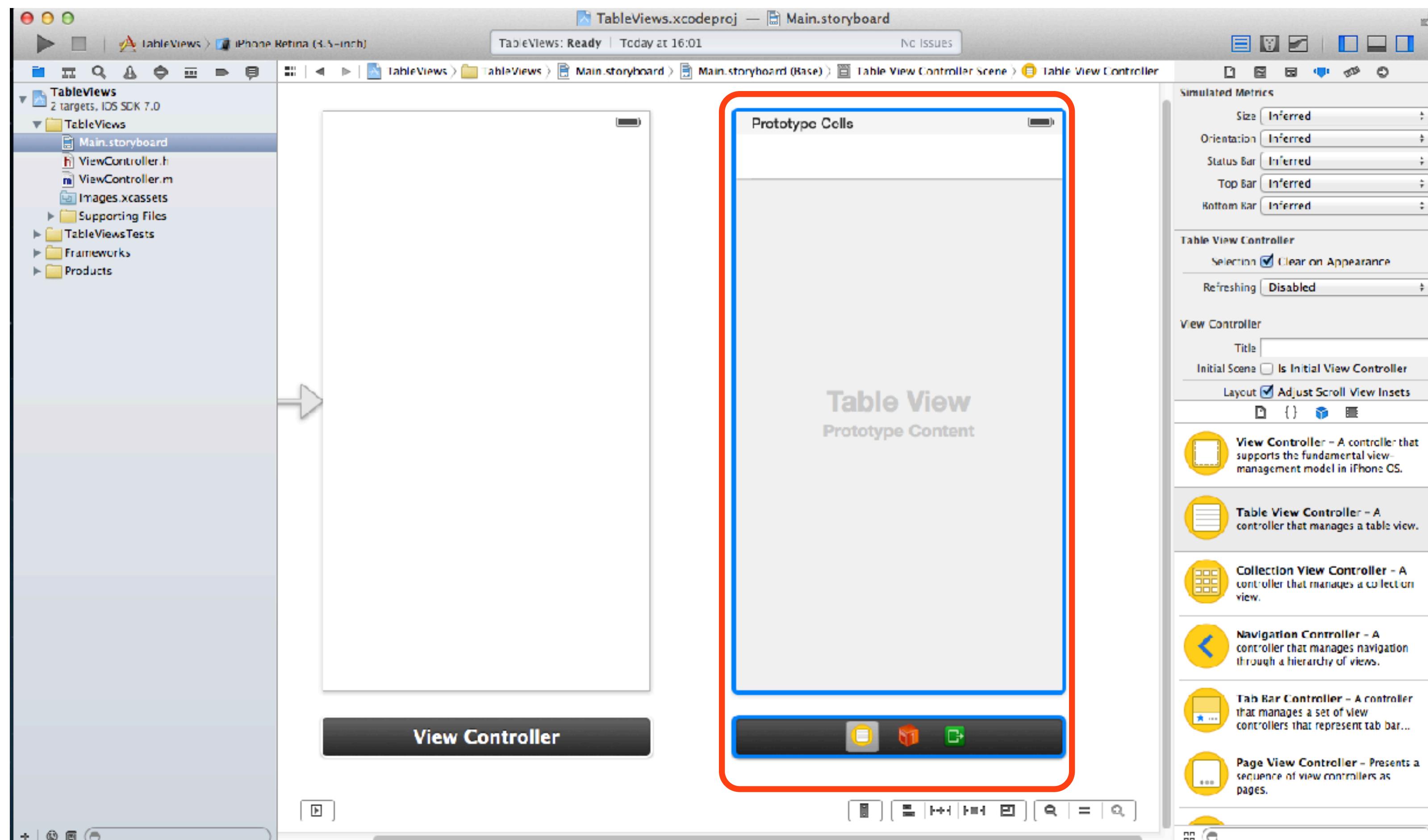
- A table view is usually controlled by a specialized view controller, called **UITableViewController**
- Table views normally take the whole view
- To add a table view controller to storyboard, just drag a Table View Controller item from the object palette
- When adding a Table View Controller, the controller is a subclass of **UITableViewController** and the view is a **UITableView**
- After a Table View Controller has been dragged, you need to create a new Objective-C class that is a subclass of **UITableViewController** and set it as the class of the view in the Identity inspector in Storyboard
- The table view can be configured to set its style (plain or grouped) and type of content (static or dynamic)

Table View Controller



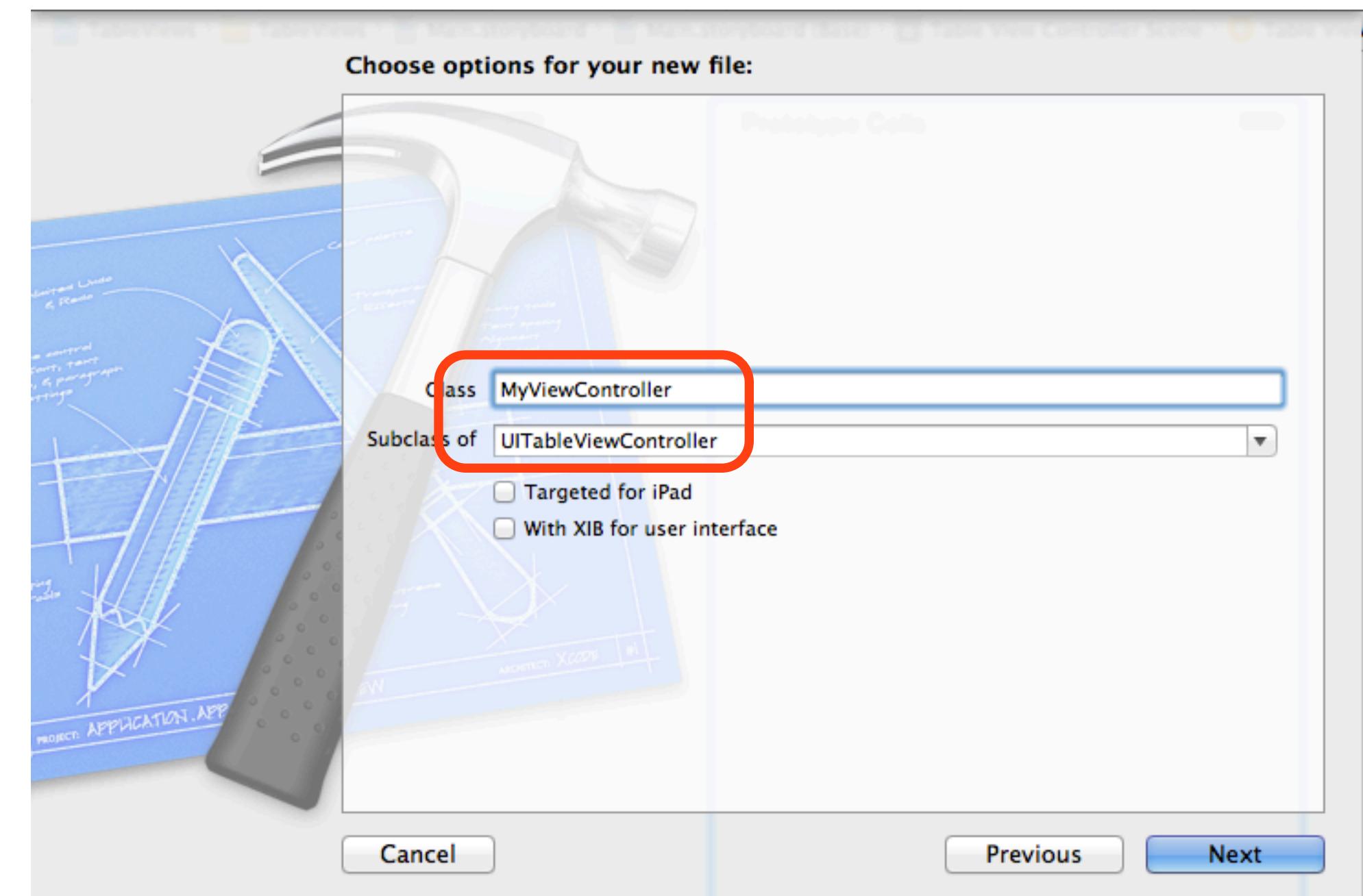
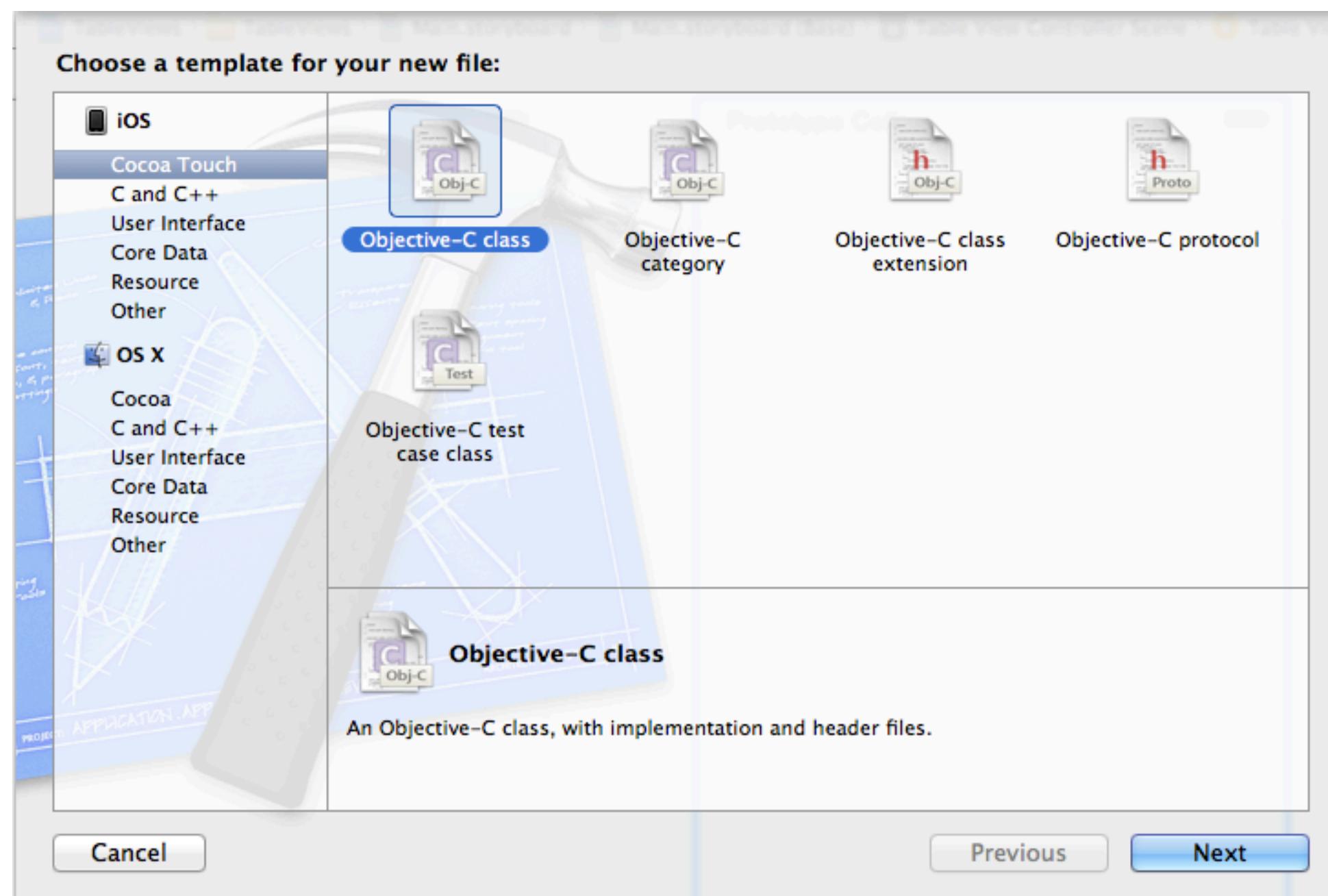
Drag a Table View Controller item from the object palette

Table View Controller



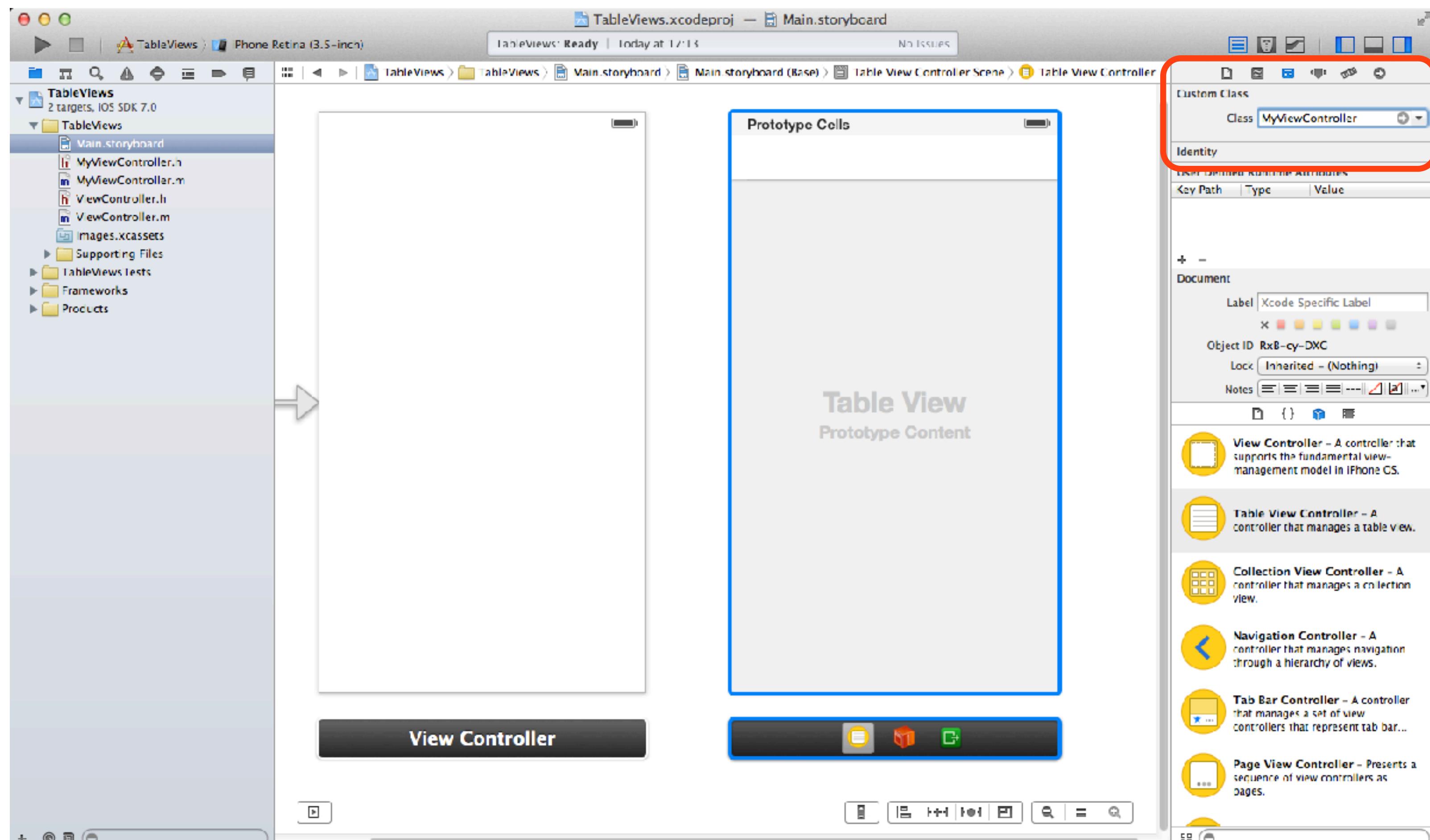
Drag a Table View Controller item from the object palette

Table View Controller



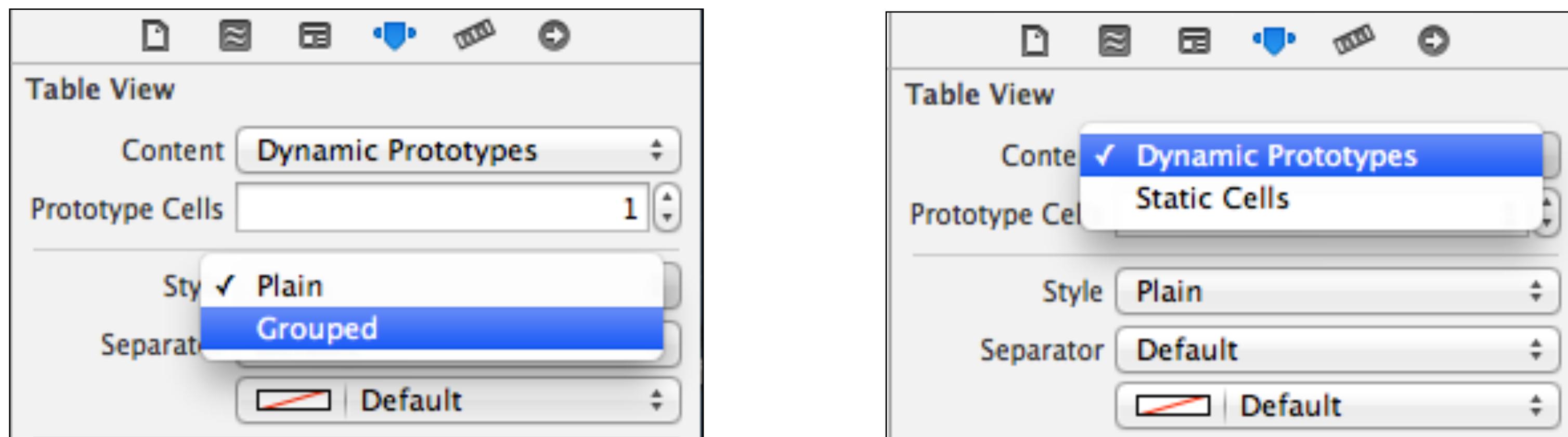
Create a new Objective-C class ($\text{⌘}-\text{N}$) that is a subclass of UITableViewController

Table View Controller



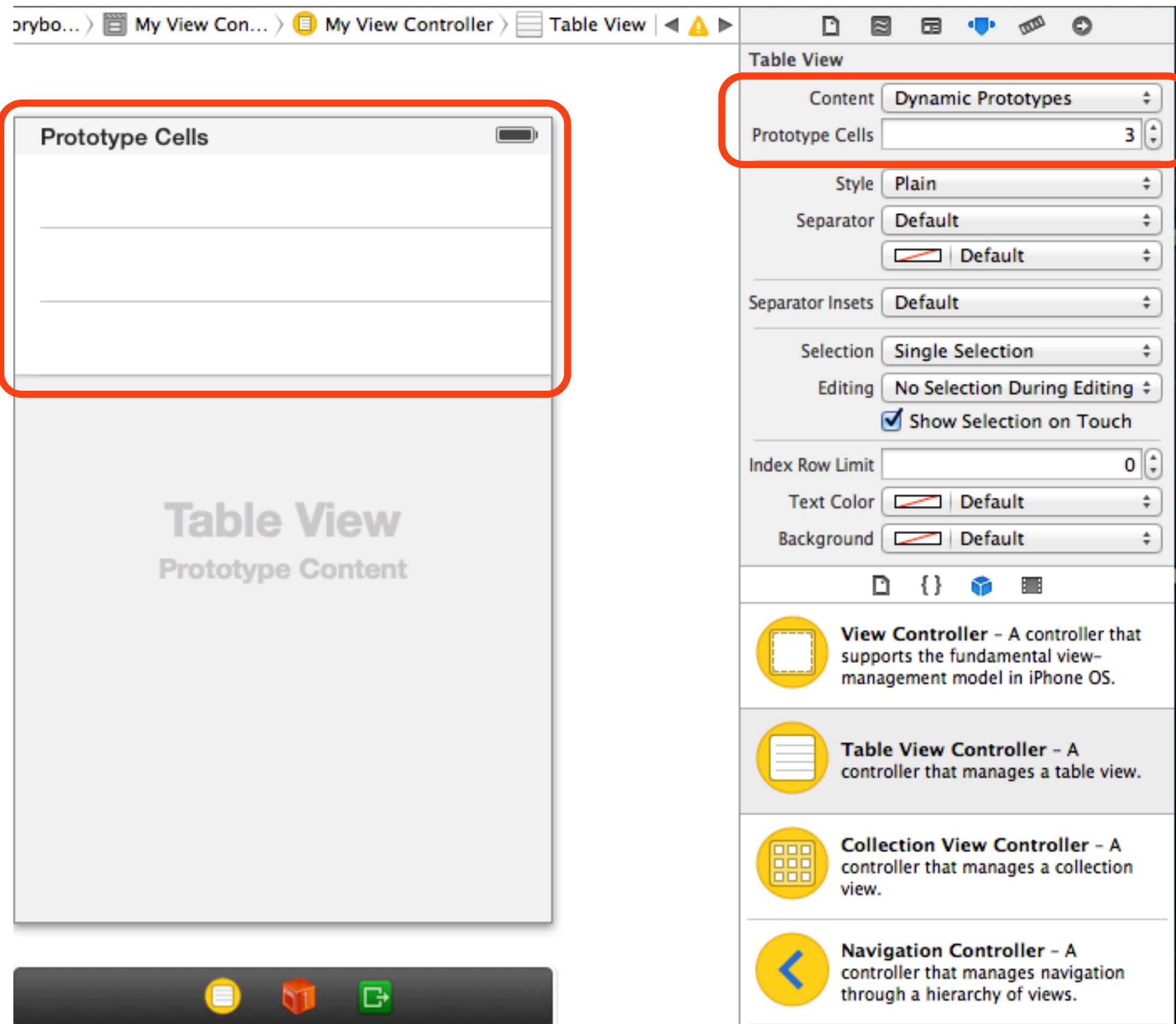
Set the new class as the class of the view in the Identity inspector

Table View Controller



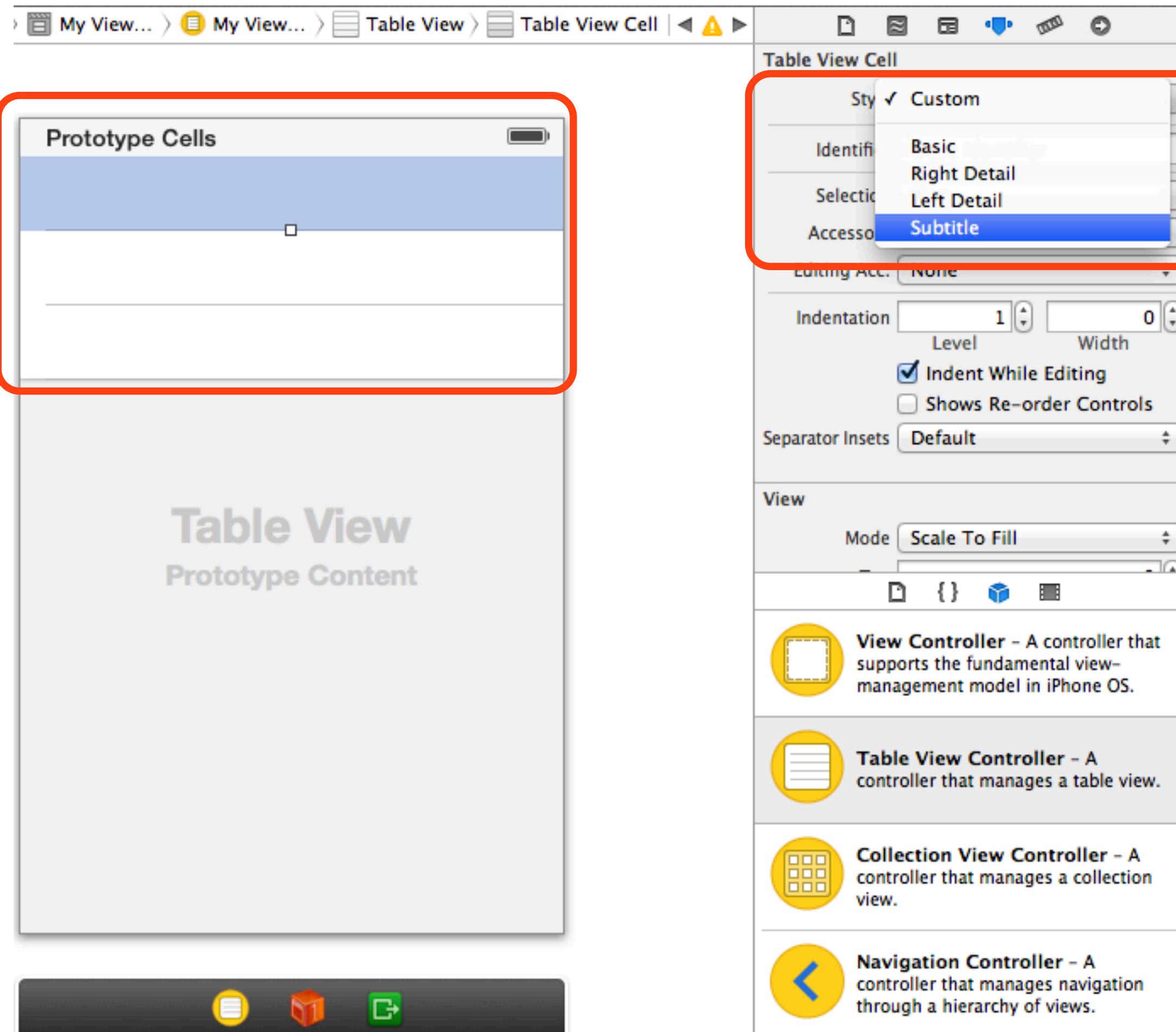
Configure style (plain or grouped) and type of content (dynamic or static) of the table view

Table View Controller



When the dynamic prototypes option is selected, it is possible to define the prototypes (templates) for the table cells that will be displayed can be configured

Table View Controller



For each prototype cell, it is possible to set its style...

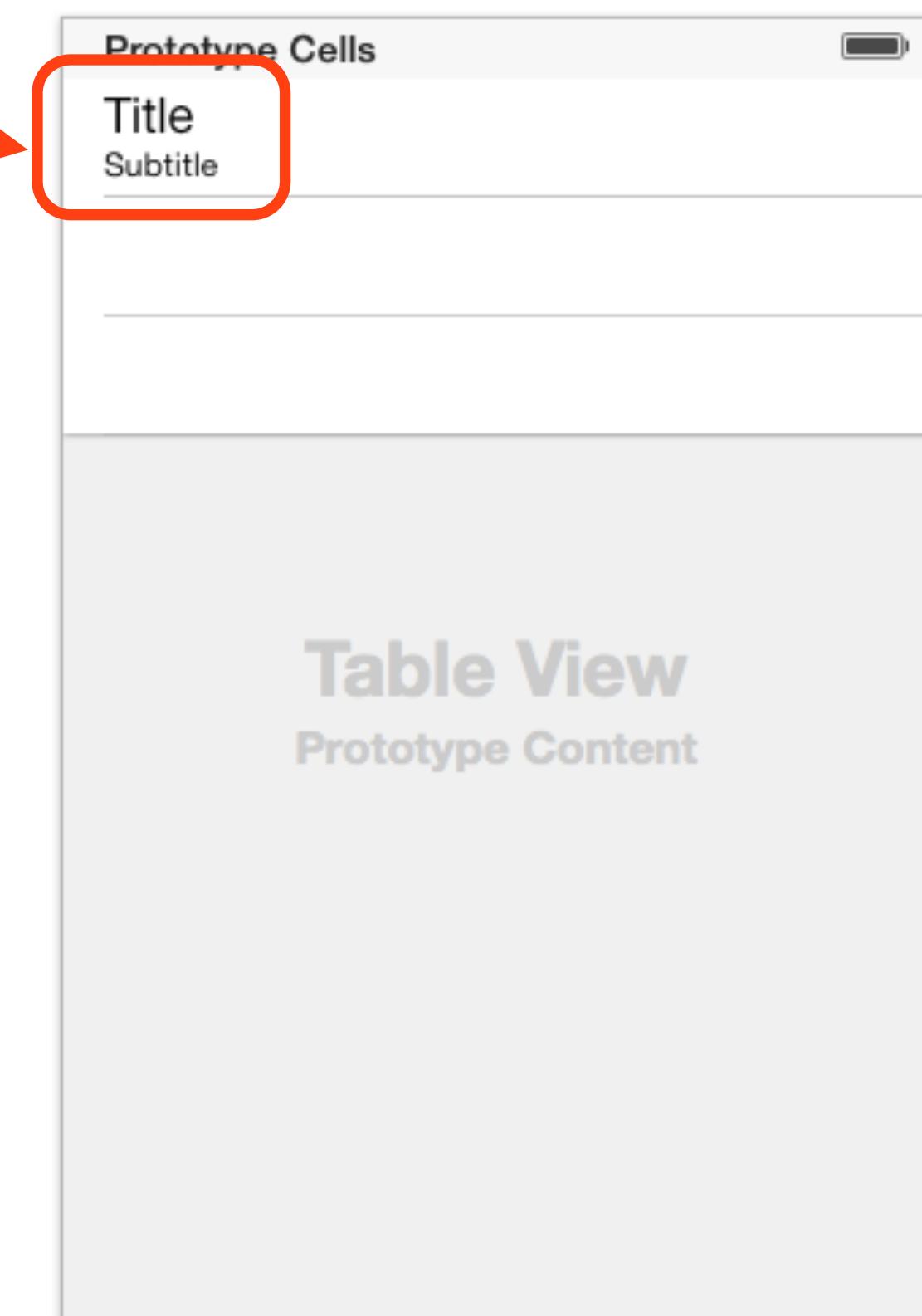
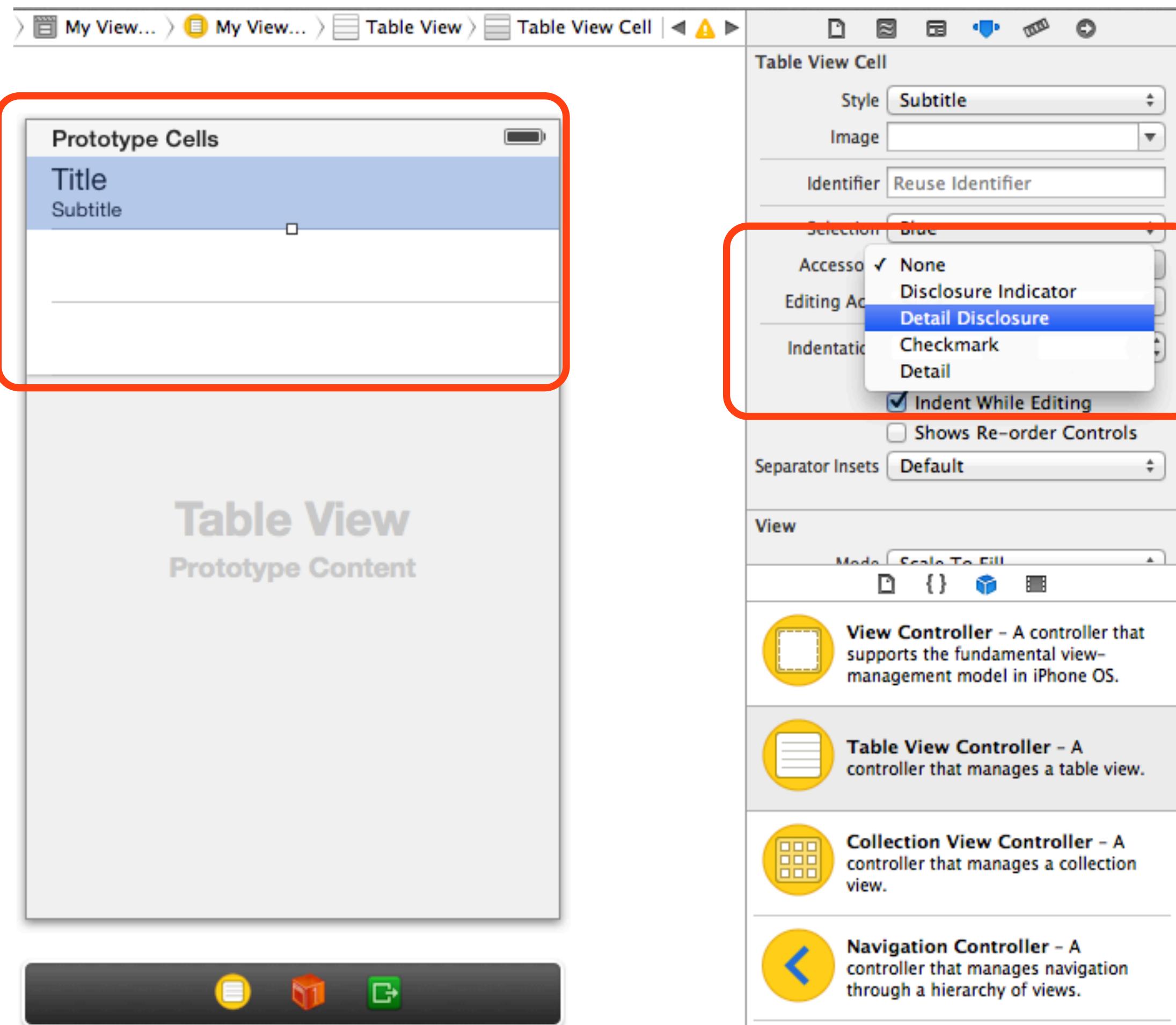


Table View Controller



... and accessories (which provide a hint of what will happen by selecting a row)

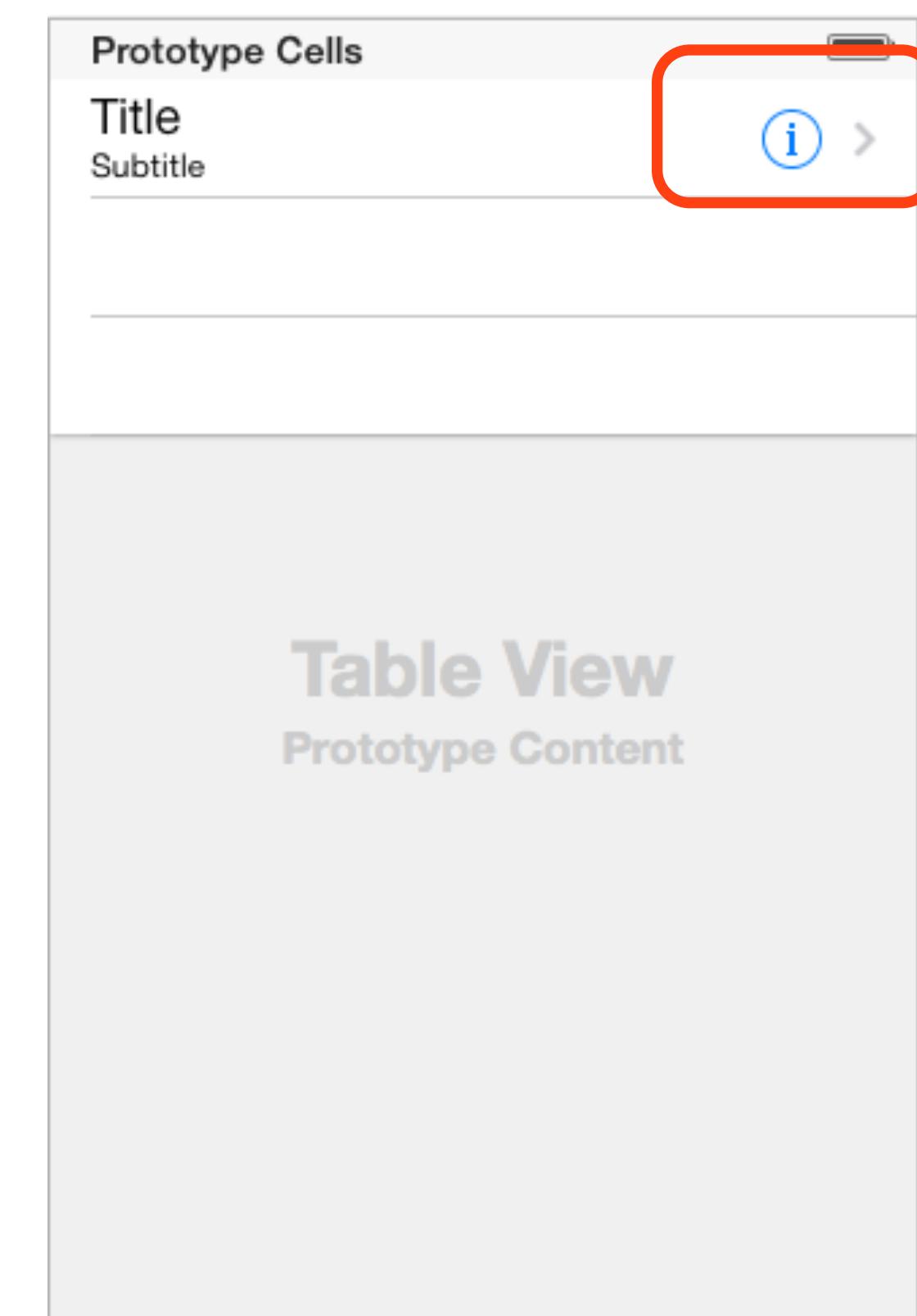
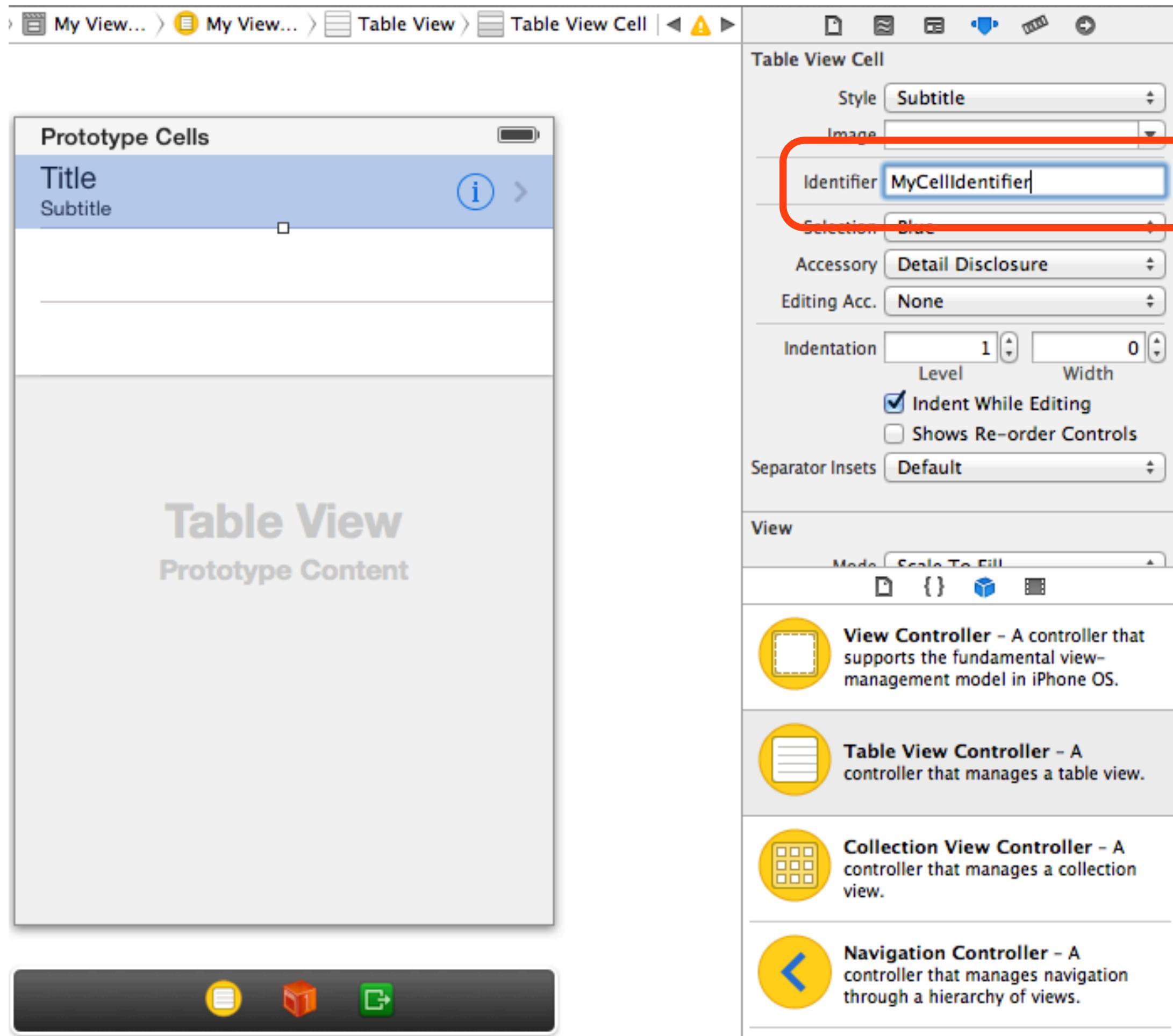




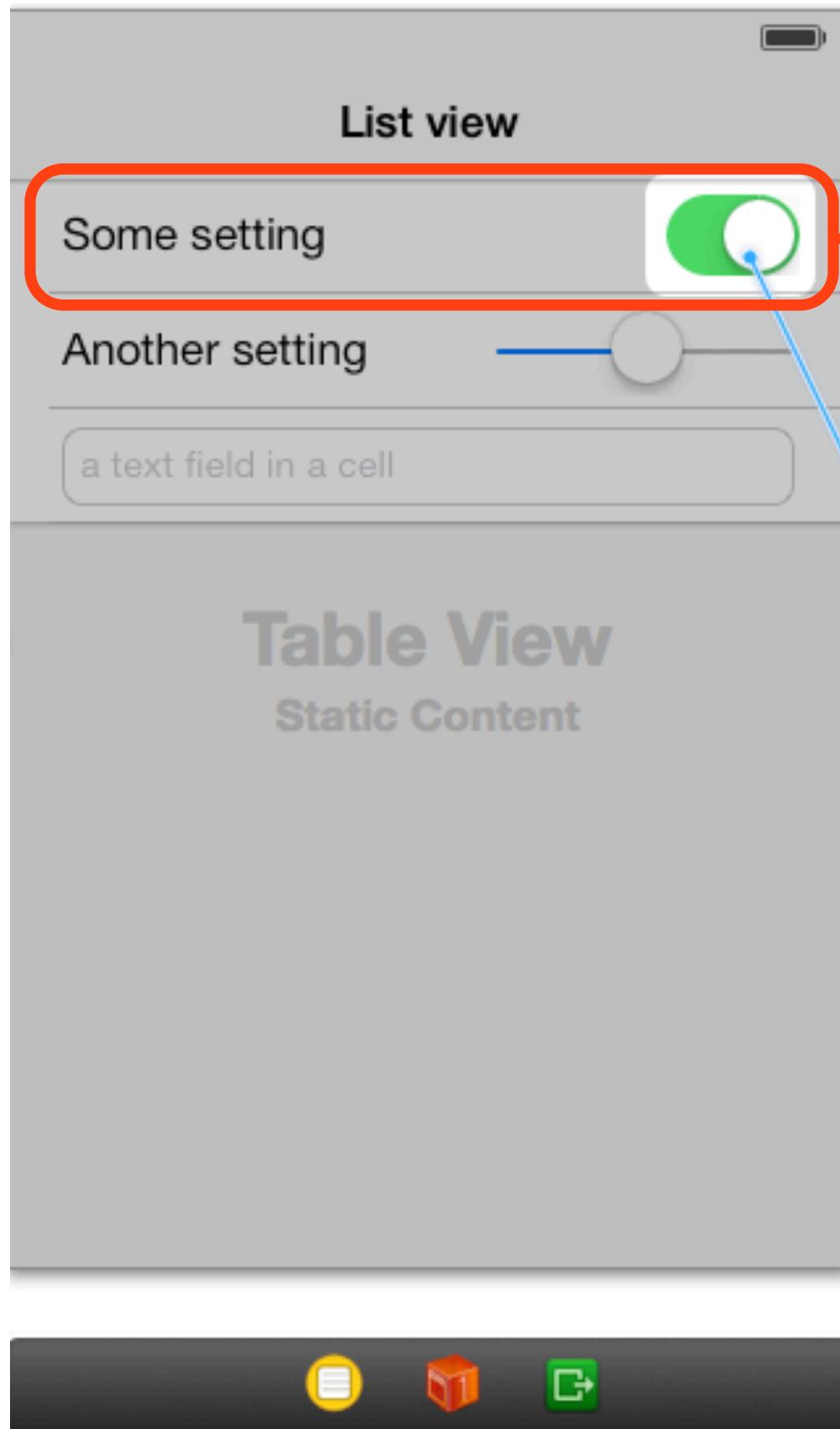
Table View Controller



An identifier for the cell must be specified so that it is possible to reference to the cell prototype in code (similar to the identifier of a segue)



Table View Controller

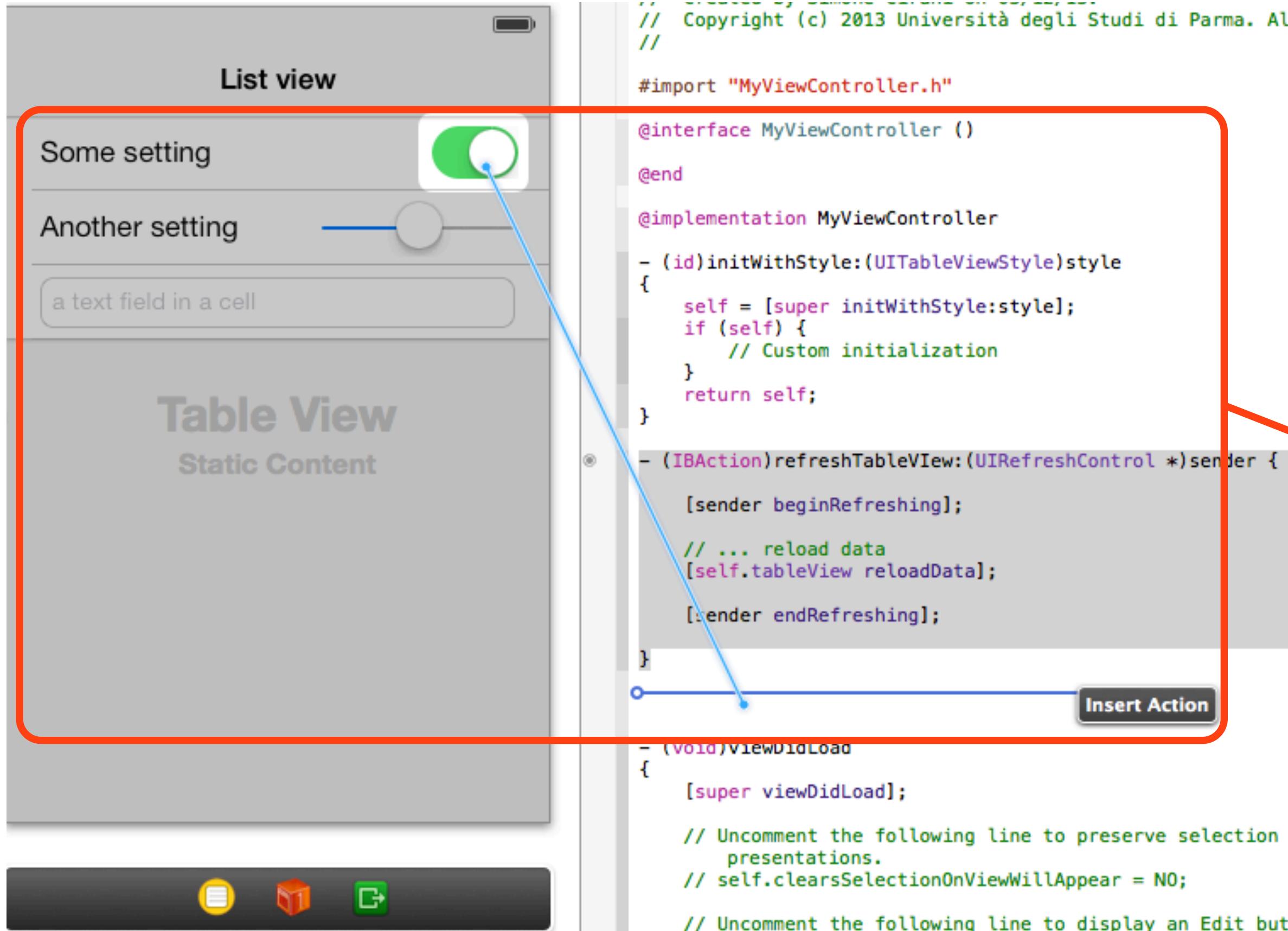


```
// ...  
// Copyright (c) 2013 Università degli Studi di Parma. All rights reserved.  
  
#import "MyViewController.h"  
  
@interface MyViewController ()  
@end  
  
@implementation MyViewController  
  
- (id)initWithStyle:(UITableViewStyle)style  
{  
    self = [super initWithStyle:style];  
    if (self) {  
        // Custom initialization  
    }  
    return self;  
}  
  
- (IBAction)refreshTableVIew:(UIRefreshControl *)sender {  
    [sender beginRefreshing];  
    // ... reload data  
    [self.tableView reloadData];  
    [sender endRefreshing];  
}  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    // Uncomment the following line to preserve selection between presentations.  
    // self.clearsSelectionOnViewWillAppear = NO;  
  
    // Uncomment the following line to display an Edit button in the navigation bar for this view controller.  
    // self.navigationItem.rightBarButtonItem = self.editButtonItem;  
}
```

Static cells can be configured directly in storyboard

Views and controls (such as labels, switches, and sliders) can be added to the cells

Table View Controller



Target-actions can be added to the view controller by ctrl-dragging from the control to the implementation file

NSIndexPath

- The **NSIndexPath** class is used to represent the path to a specific node in a tree of nested array collections
- In iOS, UIKit adds some functionalities to **NSIndexPath** instances so that they can be used to refer to the position of a cell in a table view
- There are two properties that are used for this purpose:
 - **section**: an index number identifying a section in a table view (or collection view)
 - **row**: an index number identifying a row in a section of a table view
- An instance of **NSIndexPath** is passed in as argument to the methods that refer to a specific cell in the table view in order to identify the cell univocally

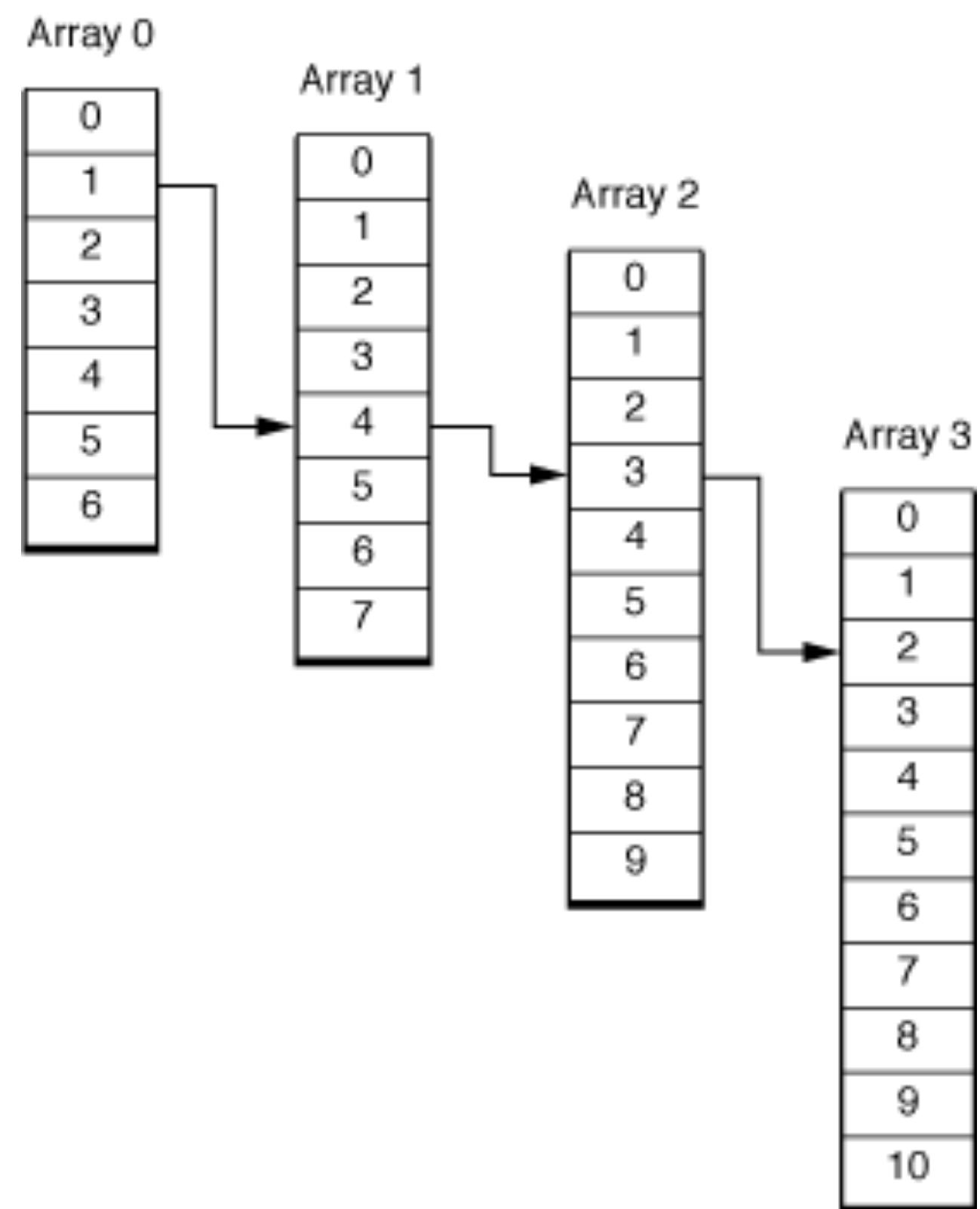


Table view content: UITableViewDataSource

- A table view relies on an external object to inject content into rows, called **data source**
- The data source stands between the model and the table view, so it is common (but not necessary) that the controller is the data source for a table view that it manages
- Typically a data source must provide content to the table by answering to the following questions:
 - ▶ How many sections will the table view have?
 - ▶ How many rows are there in section *i*?
 - ▶ What content (**UITableViewCell**) should be displayed in the cell at section *i* and row *j*?
 - ▶ Which title should be displayed for the header and footer of section *i*?
 - ▶ Is a table cell at section *i* and row *j* editable?
- A table view's data source must conform to the **UITableViewDataSource** protocol
- A data source must be provided for table views that display dynamic contents: it is the **WHAT** that you are displaying
- Static tables do not need to implement **UITableViewDataSource** protocol methods (all automatic)



UITableViewDataSource

- The **UITableViewDataSource** protocol defines the following methods, which must be implemented by a data source to give enough information to the UITableView to draw its content:
 - **(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView**
 - **(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section**
 - **(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath**
 - **(NSString *)tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)section**
 - **(NSString *)tableView:(UITableView *)tableView titleForFooterInSection:(NSInteger)section**
 - **(BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:(NSIndexPath *)indexPath**
 - **(void)tableView:(UITableView *)tableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)indexPath**



UITableViewDataSource

- `(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView`

How many sections will the given table view have? (defaults to 1)

- `(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{
 // Return the number of sections.
 return <NUMBER OF SECTIONS>;
}`



UITableViewDataSource

- `(NSInteger)tableView: (UITableView *)tableView numberOfRowsInSection: (NSInteger)section`

How many rows will there be in the given section?
- `(NSInteger)tableView: (UITableView *)tableView numberOfRowsInSection: (NSInteger)section{
 // Return the number of rows in the section.
 return <NUMBER OF ROWS FOR SECTION section>;
}`



UITableViewDataSource

- `(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath`

What content (`UITableViewCell`) should be displayed for the cell at the given index path?

```
- (UITableViewCell *)tableView:(UITableView *)tableView  
cellForRowAtIndexPath:(NSIndexPath *)indexPath{  
  
    static NSString *CellIdentifier = @"MyCellIdentifier";  
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier  
forIndexPath:indexPath];  
  
    // Configure the cell...  
  
    return cell;  
}
```

UITableViewDataSource

- `(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath`

What content (`UITableViewCell`) should be displayed for the cell at the given index path?

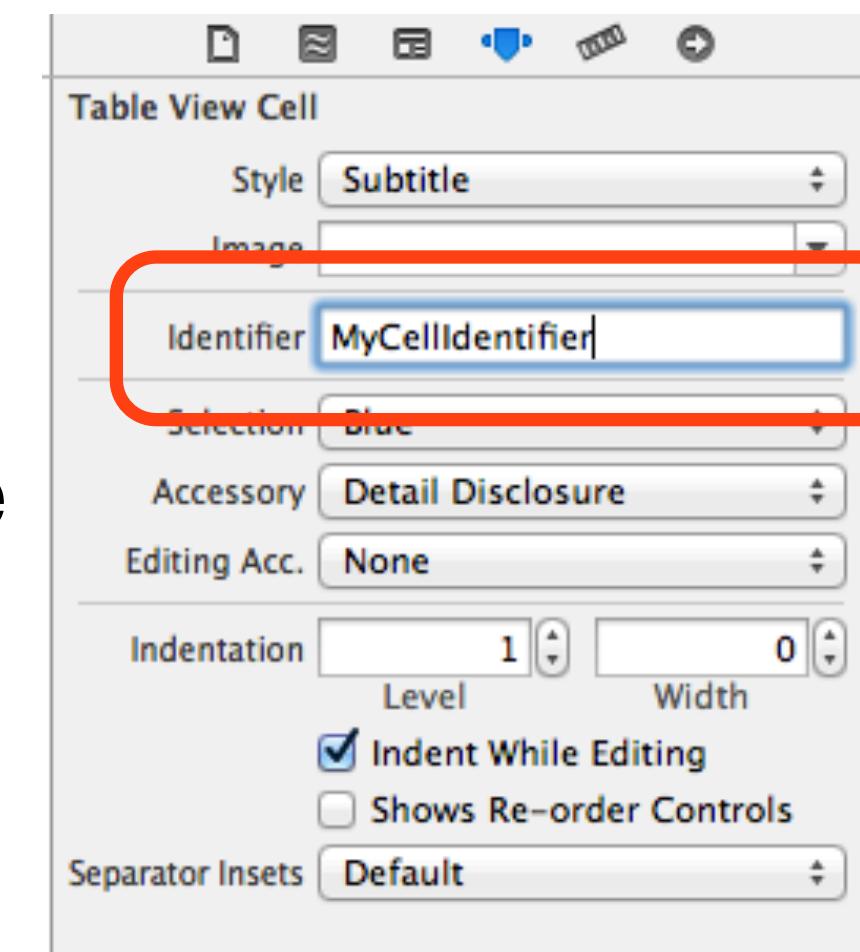
- `(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath{

static NSString *CellIdentifier = @"MyCellIdentifier";
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];

// Configure the cell...

return cell;
}`

Dynamic prototype identifier specified in the Attribute inspector in storyboard; very important to determine which prototype should be used if there are many





UITableViewDataSource

- `(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath`

What content (`UITableViewCell`) should be displayed for the cell at the given index path?

- `(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath{

static NSString *CellIdentifier = @”MyCellIdentifier”;
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];

// Configure the cell...

return cell;
}`

A table view's data source should reuse cell objects for performance reasons (it would be extremely inefficient to always create new cells for each row in the table view: only those on screen should be kept in memory)

UITableViewDataSource

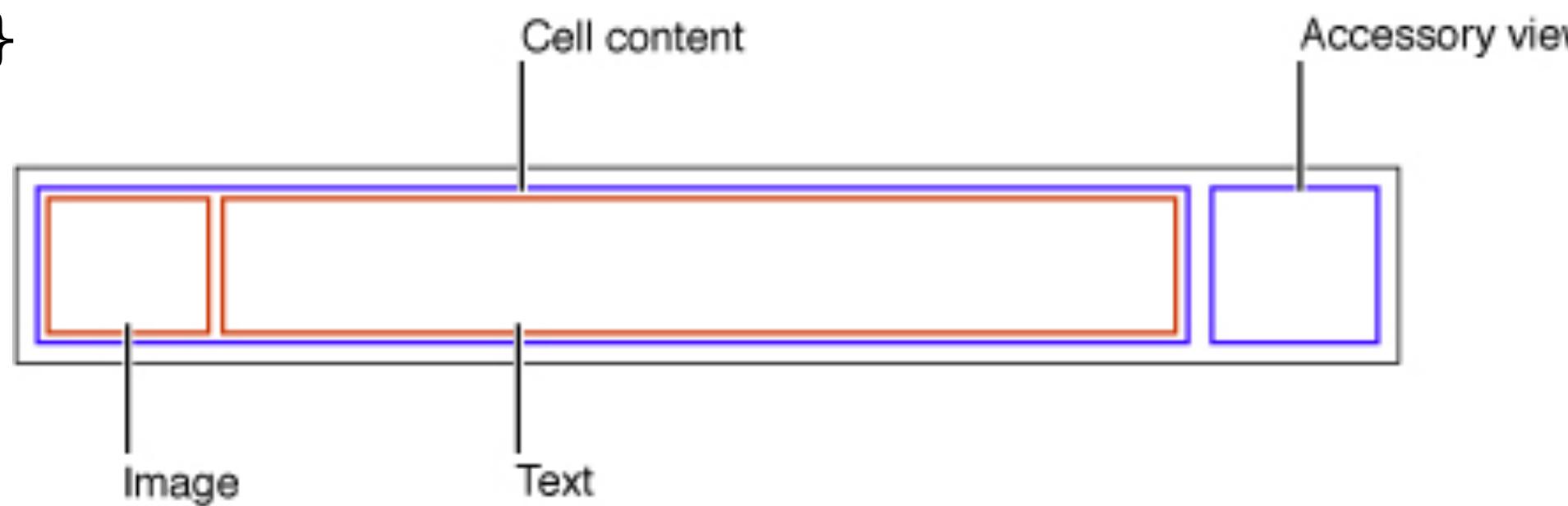
- `(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath`

What content (`UITableViewCell`) should be displayed for the cell at the given index path?

- `(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath{

static NSString *CellIdentifier = @"MyCellIdentifier";
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];

// Configure the cell...
cell.textLabel.text = [NSString stringWithFormat:@"cell at row [%d]", indexPath.row];
return cell;
}`



Here is where you can set the cell's `textLabel`, `detailTextLabel`, and `imageView` properties of the cell



UITableViewDataSource

- `(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath`

What content (`UITableViewCell`) should be displayed for the cell at the given index path?

- `(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath{

static NSString *CellIdentifier = @"MyCellIdentifier";
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];

// Configure the cell...
cell.textLabel.text = [NSString stringWithFormat:@"cell at row [%d]", indexPath.row];
return cell;
}`



Custom table view cells can also be used to display special content that does not fit into the system-provided cell types

UITableViewDataSource

- `(NSString *)tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)section`
- `(NSString *)tableView:(UITableView *)tableView titleForFooterInSection:(NSInteger)section`

What is the title for the given section's header/footer?

- `(NSString *)tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)section{
 return [NSString stringWithFormat:@"Header %d",section];
}`
- `(NSString *)tableView:(UITableView *)tableView titleForFooterInSection:(NSInteger)section{
 return [NSString stringWithFormat:@"Footer %d",section];
}`



Table view behavior: UITableViewDelegate

- A table view relies on an external object to specify its appearance and behavior, called **delegate**
- The delegate is needed to manage selections, configure section headings and footers, help to delete and reorder cells
- It is common (but not necessary) that the controller is the delegate for a table view that it manages
- Typically a delegate must implement methods that answer to the following questions:
 - ▶ What should be done if a cell is about to be/was selected/deselected? (instead of just following a segue)
 - ▶ What should be done if a cell's accessory button was tapped? (a special behavior can be supplied)
 - ▶ Which height should a cell have?
 - ▶ Which view should be displayed for the header/footer of a given section?
- A table view's delegate must conform to the **UITableViewDelegate** protocol
- A delegate must be provided for table views that must manage selections and can have particular behavior and/or appearance: it is the HOW the table view is being displayed

UITableViewDelegate

- `(NSIndexPath *)tableView:(UITableView *)tableView willSelectRowAtIndexPath:(NSIndexPath *)indexPath`
- `(NSIndexPath *)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath`
- `(NSIndexPath *)tableView:(UITableView *)tableView willDeselectRowAtIndexPath:(NSIndexPath *)indexPath`
- `(NSIndexPath *)tableView:(UITableView *)tableView didDeselectRowAtIndexPath:(NSIndexPath *)indexPath`
 - `(void)tableView:(UITableView *)tableView accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath`
 - `(CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath`
 - `(UIView *)tableView:(UITableView *)tableView viewForHeaderInSection:(NSInteger)section`
 - `(UIView *)tableView:(UITableView *)tableView viewForFooterInSection:(NSInteger)section`

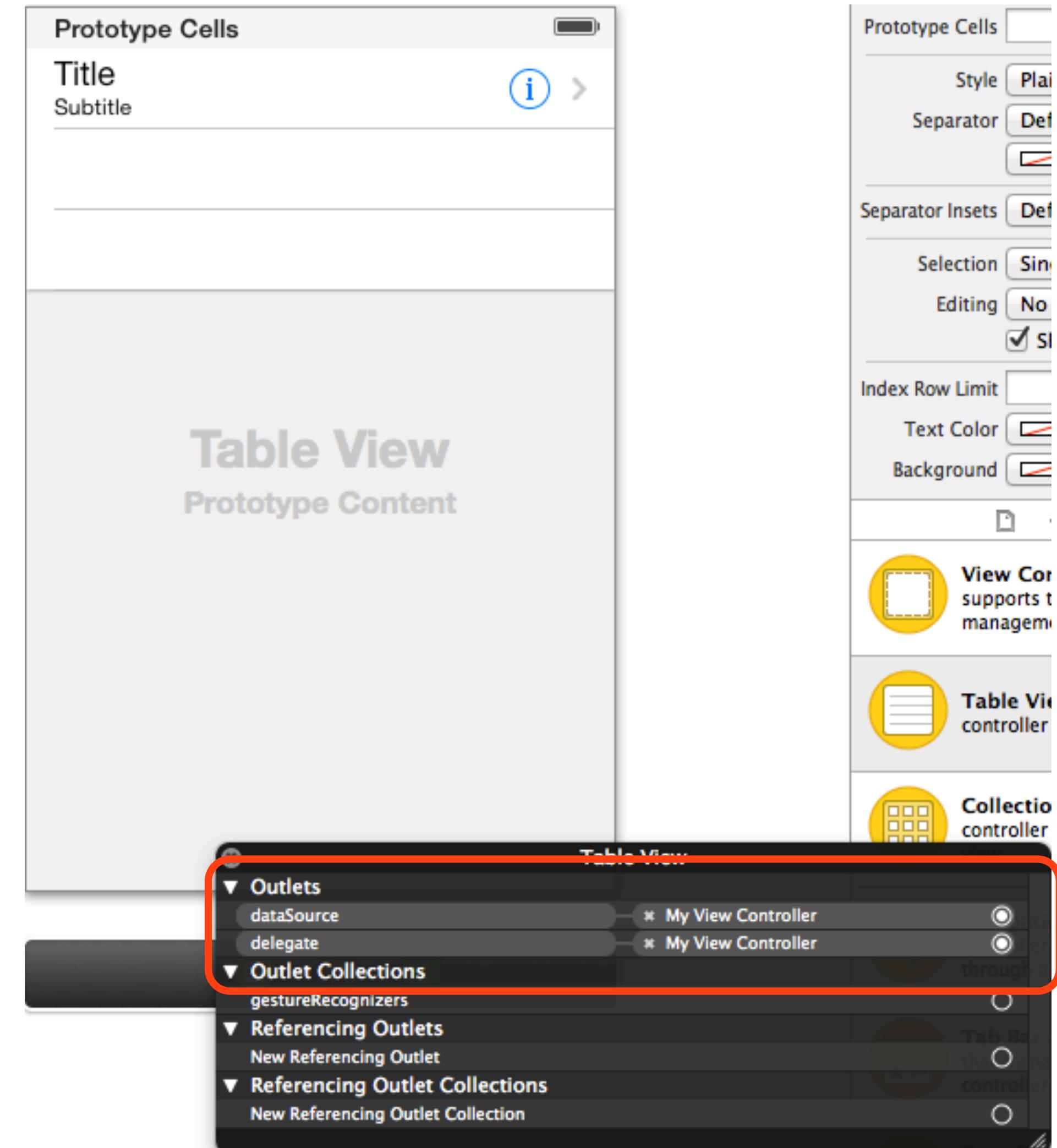
UITableViewController

- A UITableViewController sets itself as the data source and delegate for the table view it controls
- A UITableViewController has a `tableView` property which can be used to refer to the table view

```
@property UITableView *tableView;
```

- For a UITableViewController the following identity holds:

```
self.view == self.tableView;
```





UITableViewController

- When a subclass of **UITableViewController** is created, Xcode provides a skeleton implementation of the most important methods that must be implemented to conform to the **UITableViewDataSource** protocol

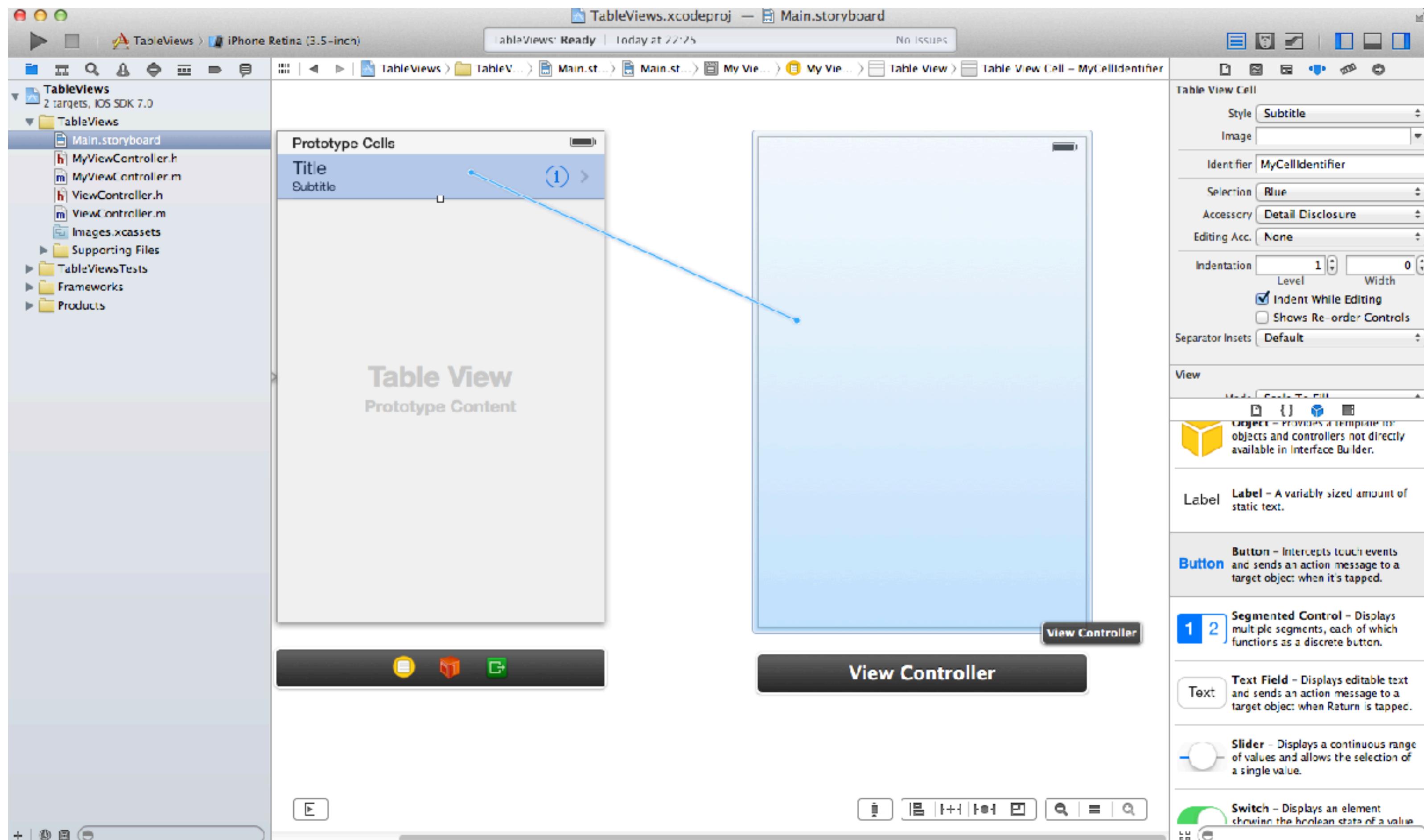
```
#pragma mark - Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section{
    // Return the number of rows in the section.
    return 0;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath{
    static NSString *CellIdentifier = @"MyCellIdentifier";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];
    // Configure the cell...
    return cell;
}
```

Segue from a UITableViewCellStyle

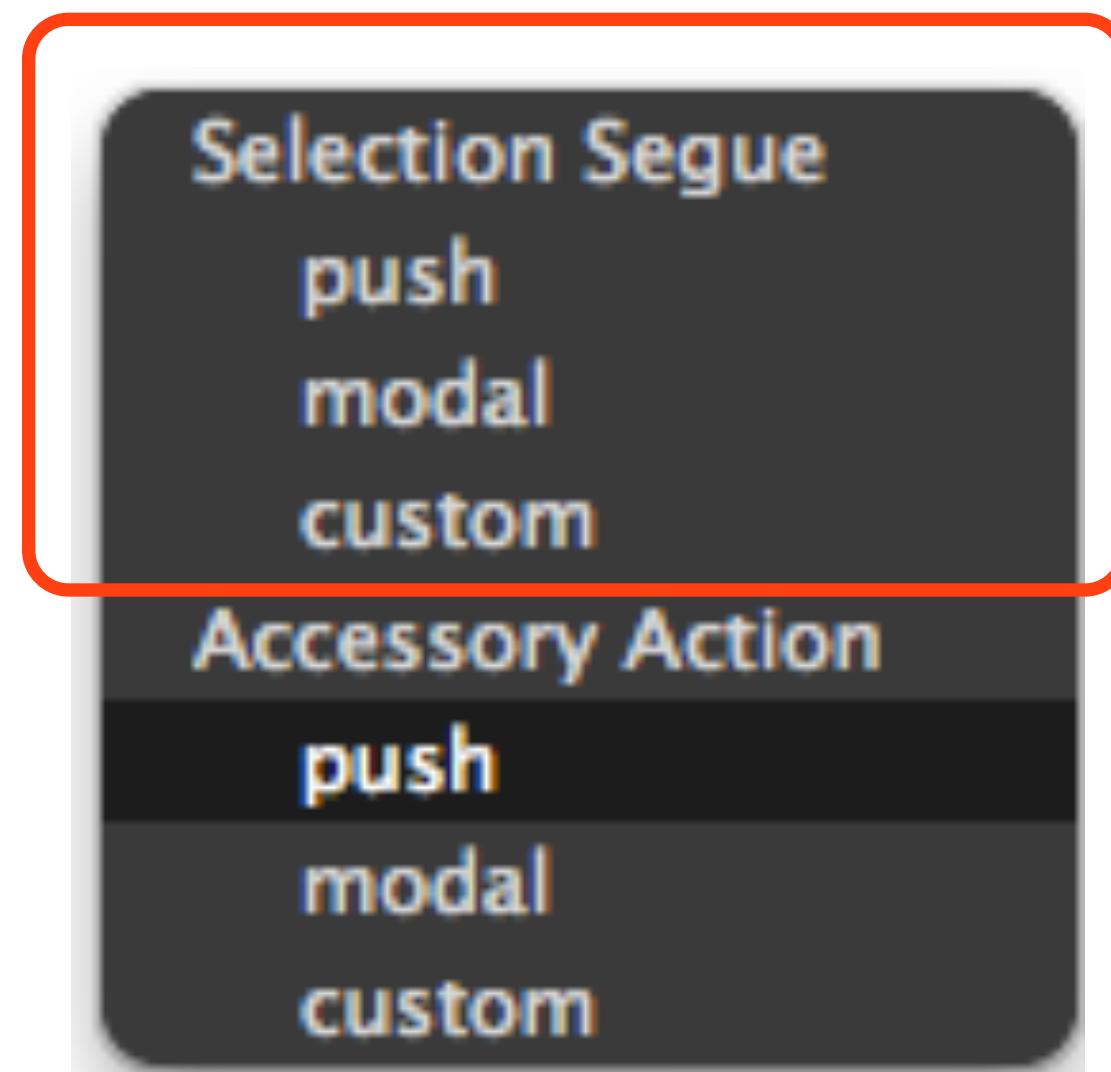


Ctrl-drag from the cell prototype to the destination view controller



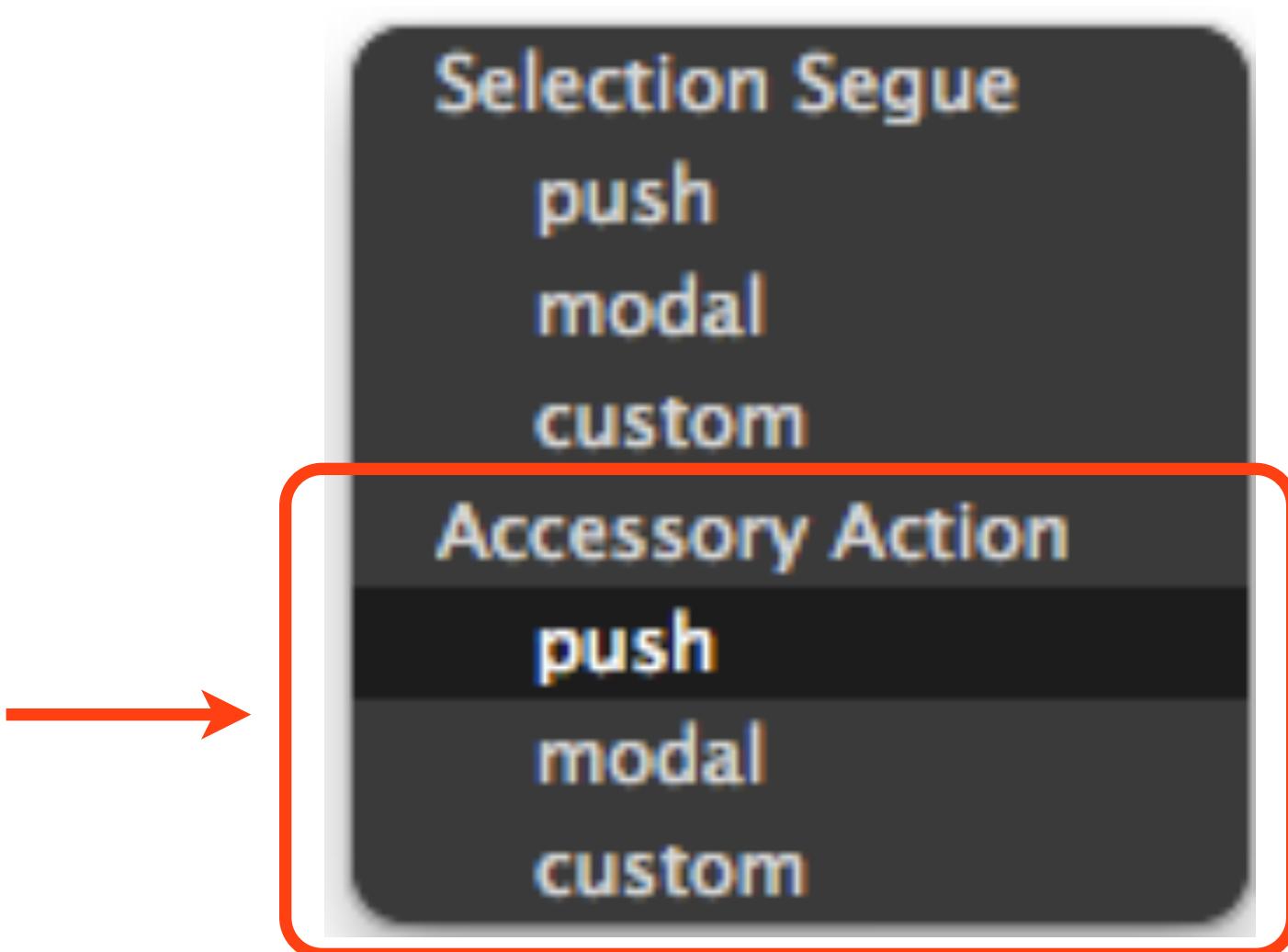
Segue from a UITableViewCells

A segue can be configured to start when the cell is selected... →



Segue from a UITableViewCellStyle

...or when the cell's accessory is tapped





Preparing for a segue from a UITableViewCells

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender{
    // Get the new view controller using [segue destinationViewController].
    // Pass the selected object to the new view controller.

    NSIndexPath *indexPath = [self.tableView indexPathForCell:sender];
    if([segue.destinationViewController isKindOfClass:[SomeViewController class]]){
        SomeViewController *vc = (SomeViewController *)segue.destinationViewController;
        vc.title = [NSString stringWithFormat:@"Detail %d",indexPath.row];
    }
}
```

Preparing for a segue from a UITableViewCells

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender{  
    // Get the new view controller using [segue destinationViewController].  
    // Pass the selected object to the new view controller.  
  
    NSIndexPath *indexPath = [self.tableView indexPathForCell:sender];  
  
    if([segue.destinationViewController isKindOfClass:[SomeViewController class]]){  
  
        SomeViewController *vc = (SomeViewController *)segue.destinationViewController;  
        vc.title = [NSString stringWithFormat:@"Detail %d",indexPath.row];  
    }  
}
```



The sender is the table view cell that was selected

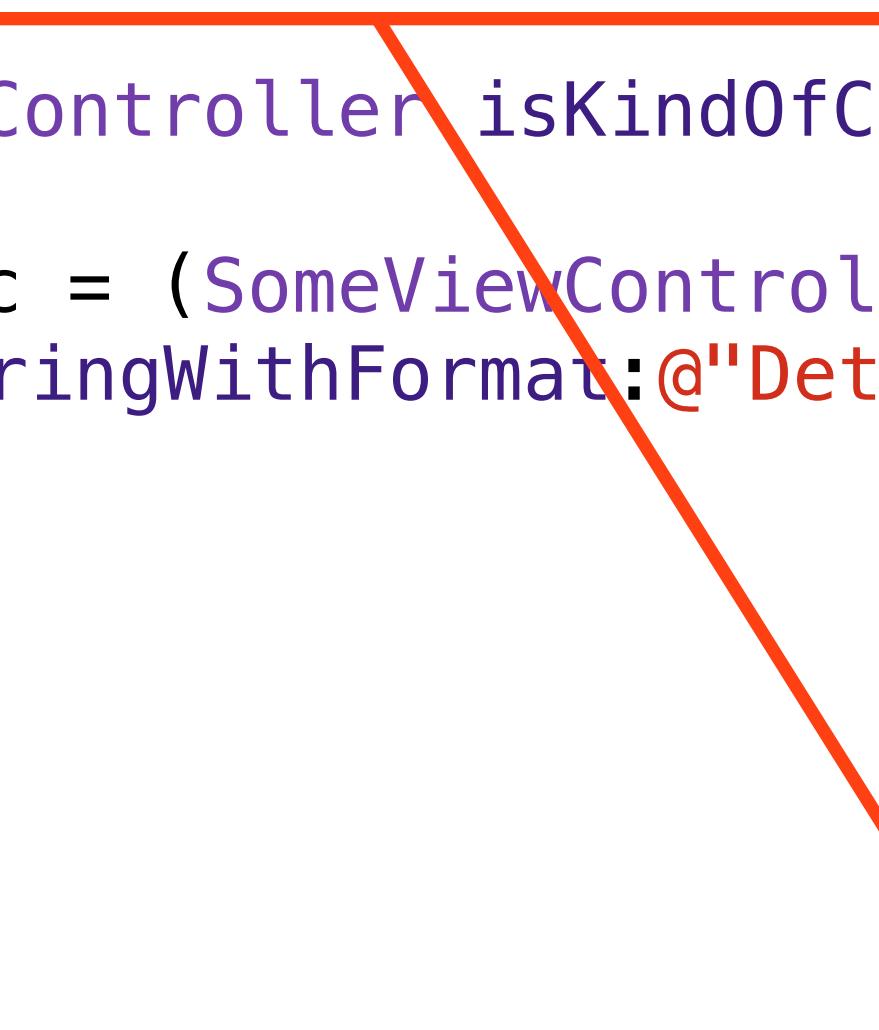
Sometimes, it can be a good idea also to use
introspection to check first whether the sender is actually
a UITableViewCells using `isKindOfClass:`

Preparing for a segue from a UITableViewCells

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender{
    // Get the new view controller using [segue destinationViewController].
    // Pass the selected object to the new view controller.

    NSIndexPath *indexPath = [self.tableView indexPathForCell:sender];
}

if([segue.destinationViewController isKindOfClass:[SomeViewController class]]){
    SomeViewController *vc = (SomeViewController *)segue.destinationViewController;
    vc.title = [NSString stringWithFormat:@"Detail %d",indexPath.row];
}
```



The index path of the cell can be retrieved with the **UITableView** method **indexPathForCell**:

Preparing for a segue from a UITableViewCells

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender{
    // Get the new view controller using [segue destinationViewController].
    // Pass the selected object to the new view controller.

    NSIndexPath *indexPath = [self.tableView indexPathForCell:sender];
    if([segue.destinationViewController isKindOfClass:[SomeViewController class]]){
        SomeViewController *vc = (SomeViewController *)segue.destinationViewController;
        vc.title = [NSString stringWithFormat:@"Detail %d", indexPath.row];
    }
}
```

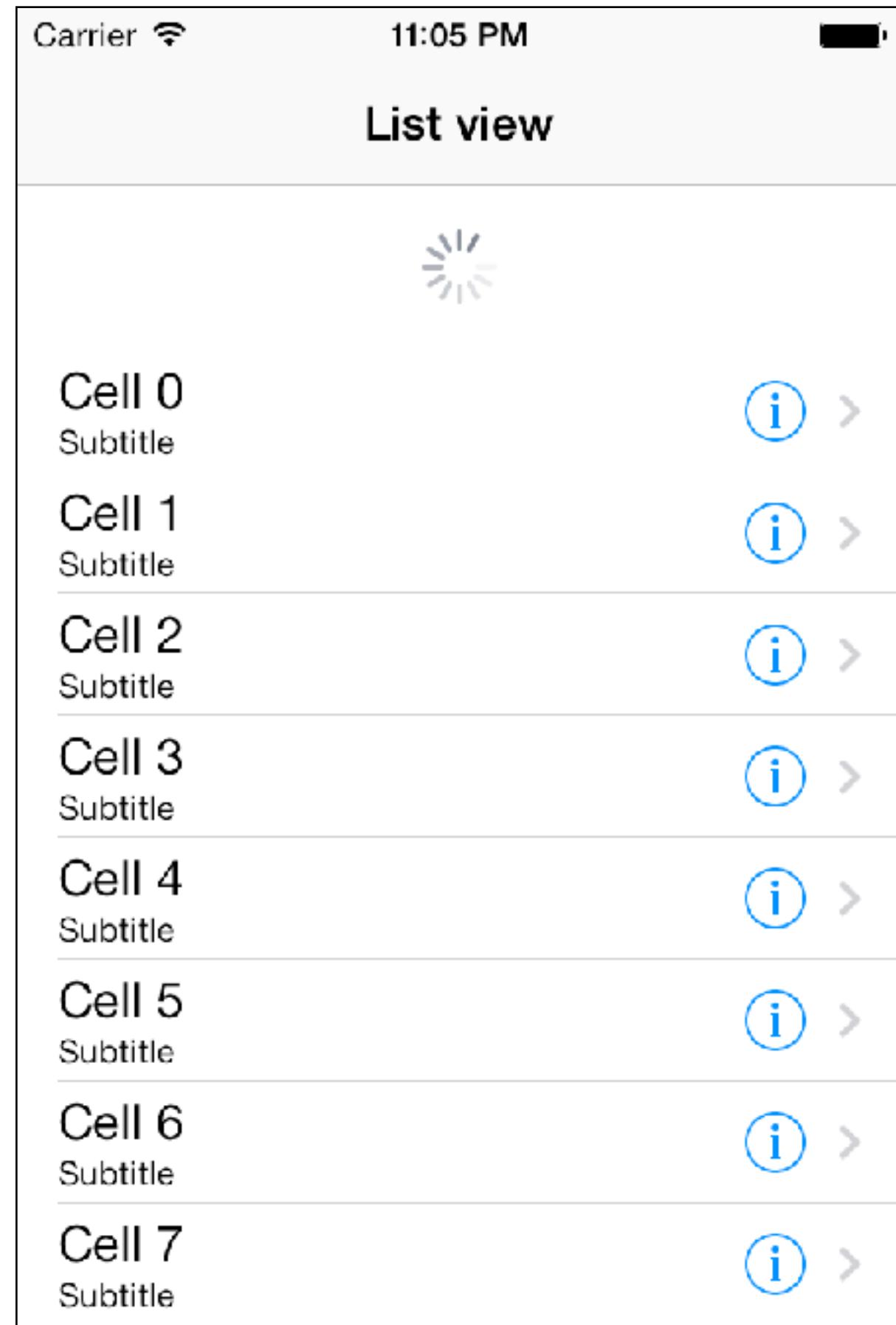


Pass data to the destination view controller

Pull to refresh

- All **UITableViewController** instances come with a built-in activity indicator that can be used to indicate that table view data are being reloaded
- If an action has been set for the refresh control, it will be triggered when the user pulls the table view down (usually referred to as “pull-to-refresh”)

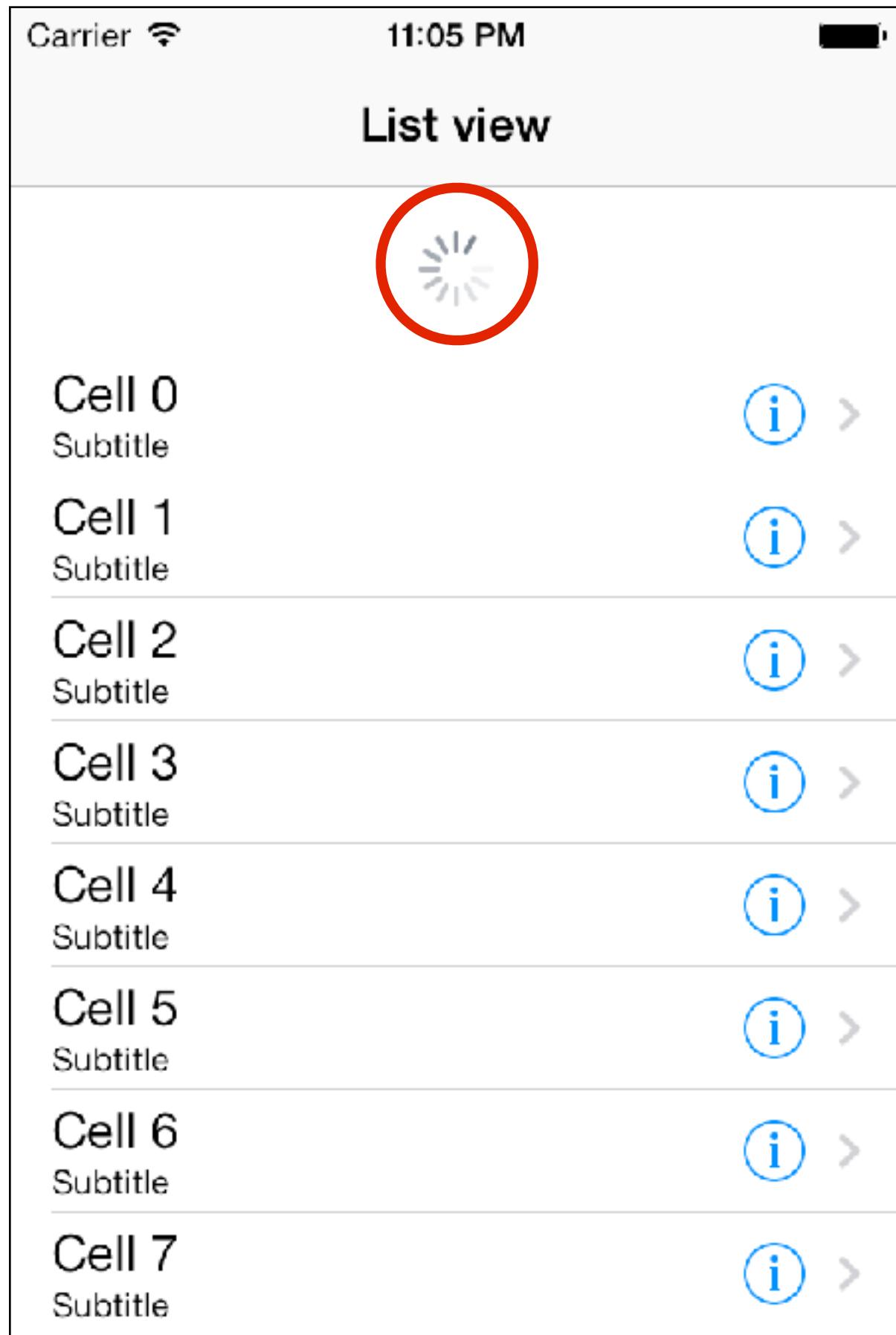
```
@property UIRefreshControl *refreshControl;
```



Pull to refresh

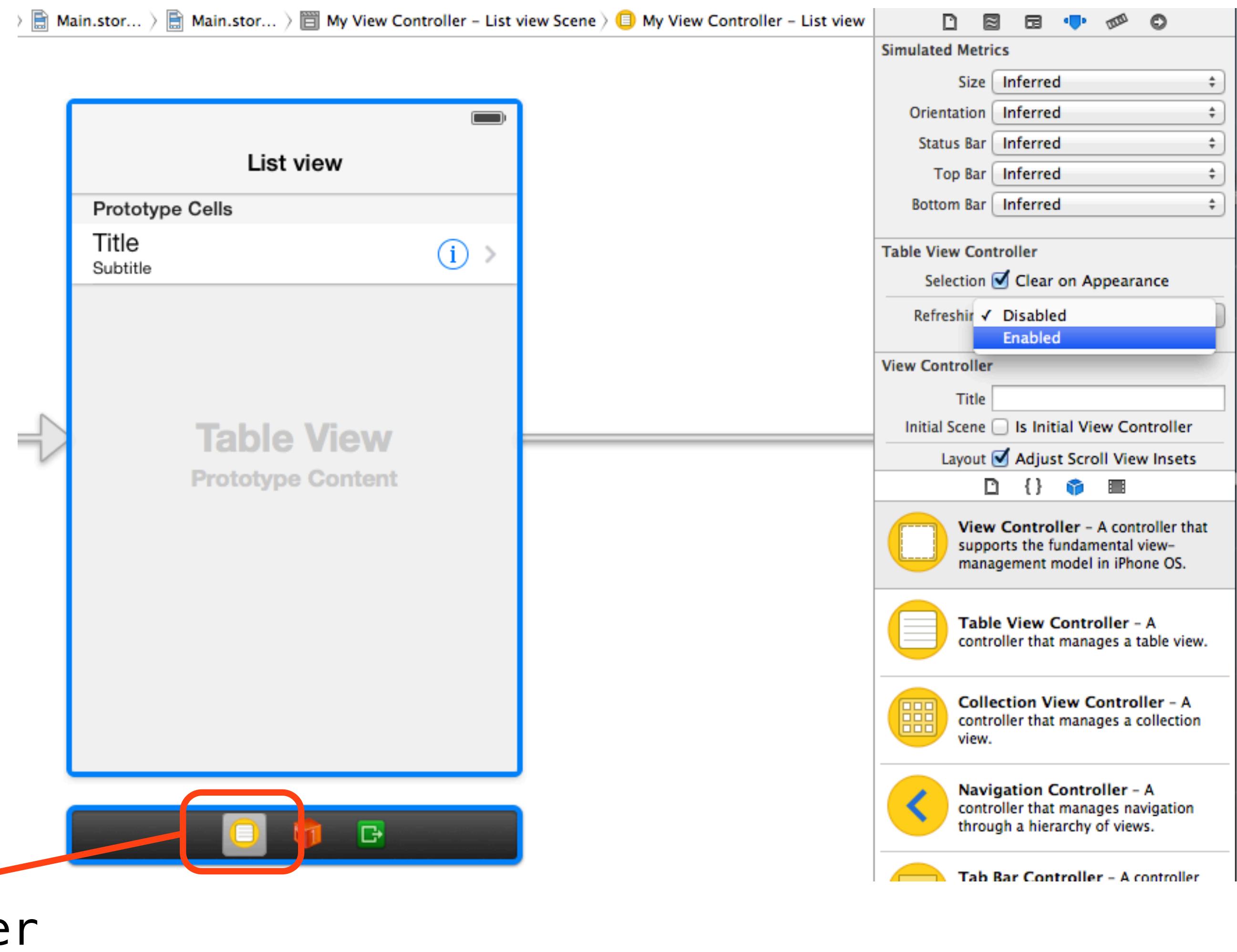
- All **UITableViewController** instances come with a built-in activity indicator that can be used to indicate that table view data are being reloaded
- If an action has been set for the refresh control, it will be triggered when the user pulls the table view down (usually referred to as “pull-to-refresh”)

```
@property UIRefreshControl *refreshControl;
```



Pull to refresh

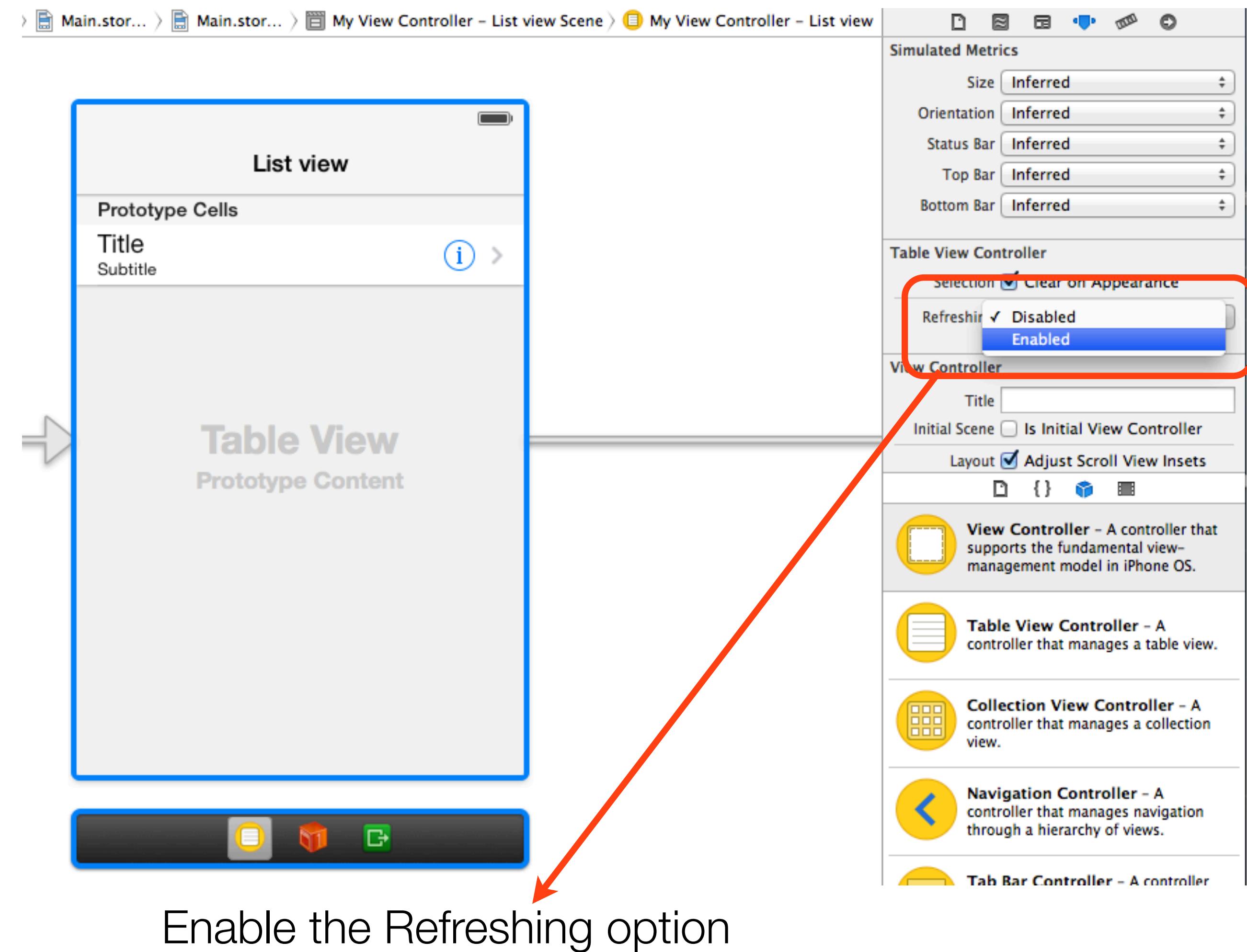
- The activity view is called refresh control and can be enabled in the Attribute inspector of the **UITableViewController**



Select the **UIViewController**

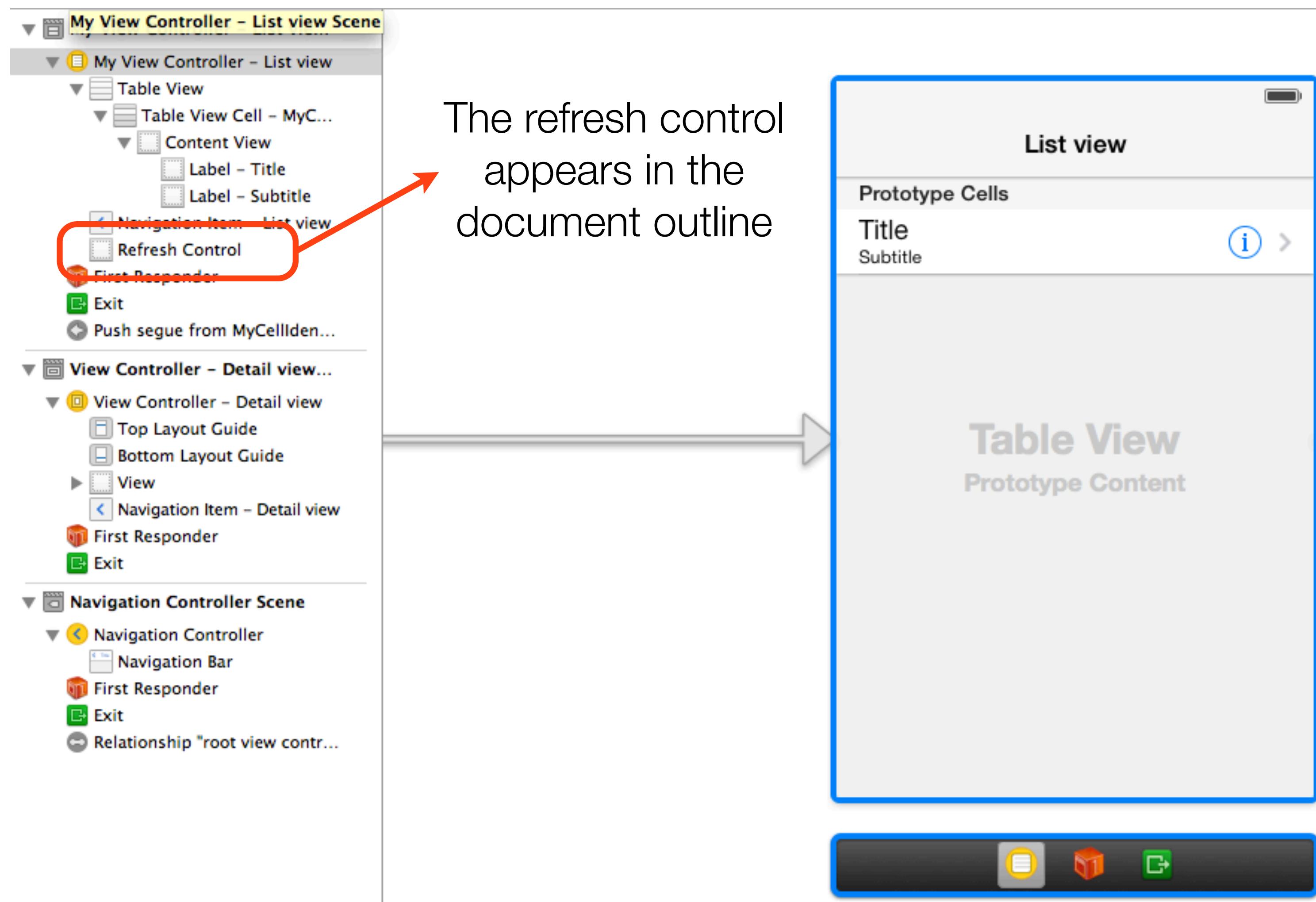
Pull to refresh

- The activity view is called refresh control and can be enabled in the Attribute inspector of the **UITableViewController**

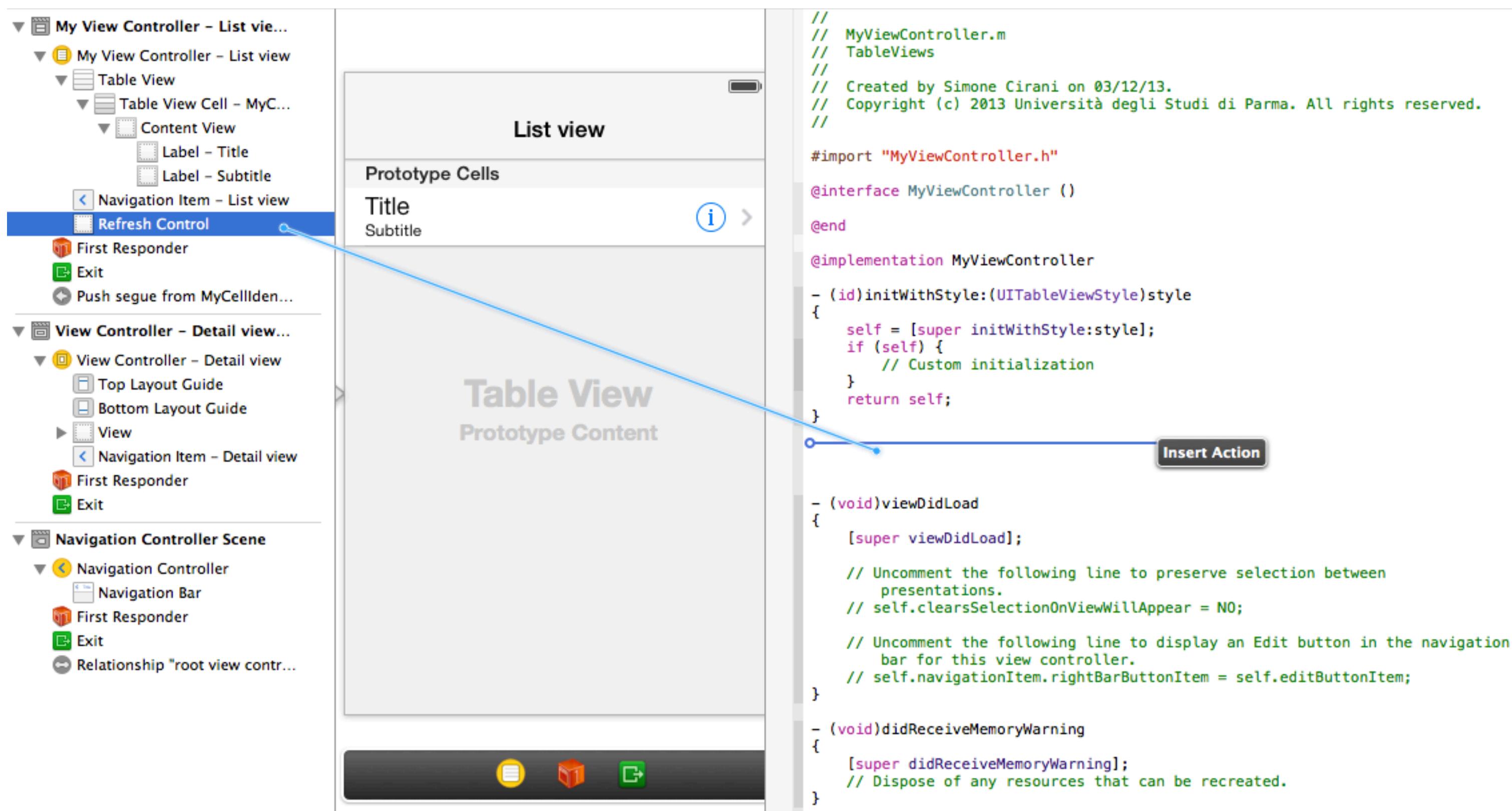


Enable the Refreshing option

Pull to refresh



Pull to refresh



Ctrl-drag from the refresh control to insert a target-action



Pull to refresh

```
- (IBAction)refreshTableView:(UIRefreshControl *)sender {  
    [sender beginRefreshing];  
  
    // ... reload data  
  
    [self.tableView reloadData];  
  
    [sender endRefreshing];  
}
```

Pull to refresh

```
- (IBAction)refreshTableView:(UIRefreshControl *)sender {  
    [sender beginRefreshing]; → Start the spinner  
    // ... reload data  
    [self.tableView reloadData];  
    [sender endRefreshing];  
}
```

Pull to refresh

```
- (IBAction)refreshTableView:(UIRefreshControl *)sender {  
    [sender beginRefreshing];  
    // ... reload data  
    [self.tableView reloadData];  
    [sender endRefreshing];  
}
```



Fetch fresh data in background queue

Pull to refresh

```
- (IBAction)refreshTableView:(UIRefreshControl *)sender {  
    [sender beginRefreshing];  
    // ... reload data  
    [self.tableView reloadData];  
    [sender endRefreshing];  
}
```



Redraw the cells

Pull to refresh

```
- (IBAction)refreshTableView:(UIRefreshControl *)sender {  
    [sender beginRefreshing];  
    // ... reload data  
    [self.tableView reloadData];  
    [sender endRefreshing];  
}
```



Stop the spinner



UIWebView

- A web view is used to embed web content, loaded from a URL, into an application
- Web views support browsing and moving back and forward in the history
- The **UIWebView** class implements the web view
- A **UIWebView** object can be dragged into the view controller in storyboard or created programmatically and added to the view
- A **UIWebView** can then be linked as an outlet by ctrl-dragging from the UIWebView to the interface declaration of the view controller
- It is possible to configure the **UIWebView** to automatically scale web content to fit on the screen by setting the **scalesPagesToFit** property



UIWebView

- To load content into the web view, two methods can be used:
 - **loadRequest:**

```
NSURL *url = [NSURL URLWithString:@"http://mobdev.ce.unipr.it"];
NSURLRequest *request = [NSURLRequest requestWithURL:url];
[self.webView loadRequest:request];
```

- **loadHTMLString:baseURL:**

```
NSString *html = @"<html><body>Hello, MobDev!</body></html>";
[self.webView loadHTMLString:html baseURL:url];
```



UIWebViewDelegate

- A web view can have a delegate that can be used to track the loading of web content
- The delegate object must conform to the **UIWebViewDelegate** protocol
- Set as web view delegate:

```
self.webView.delegate = self;
```

- A delegate's class must declare to conform to the **UIWebViewDelegate** protocol and implement the optional methods:

```
@interface WebViewViewController : UIViewController <UIWebViewDelegate>
```

- If a delegate has been set for a web view, it must be set to nil before disposing the web view instance (e.g. in the dealloc method where the web view is released)

```
- (void)dealloc{  
    webView.delegate = nil;  
}
```



UIWebViewDelegate methods

```
- (BOOL)webView:(UIWebView *)webView  
shouldStartLoadWithRequest:(NSURLRequest *)request  
navigationType:(UIWebViewNavigationType)navigationType
```

- This method is called before a web view begins loading a frame and can be used to “trap” the loading
- If the method returns YES the content is loaded, otherwise the loading will not occur
- Arguments passed in:
 - webView: the web view that is about to load content
 - request: the URL loading request, which can be used to get the URL that is about to load
 - navigationType: the type of action (a link was clicked, a form submitted, a page was reloaded, ...)



UIWebViewDelegate methods

- `(void)webViewDidStartLoad:(UIWebView *)webView`

- This method is called after a web view has started loading a frame
- Arguments passed in:
 - `webView`: the web view that is loading content



UIWebViewDelegate methods

- `(void)webViewDidFinishLoad:(UIWebView *)webView`

- This method is called after a web view has finished loading a frame
- Arguments passed in:
 - `webView`: the web view that is loaded content



UIWebViewDelegate methods

- `(void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error`

- This method is called if a web view fails to load
- Arguments passed in:
 - **webView**: the web view that is about to load content
 - **error**: the error that occurred during the loading



Mobile Application Development



Lecture 7
ScrollView, TableView, WebView



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).



Mobile Application Development



Lecture 8
Core Location and Map Kit



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).

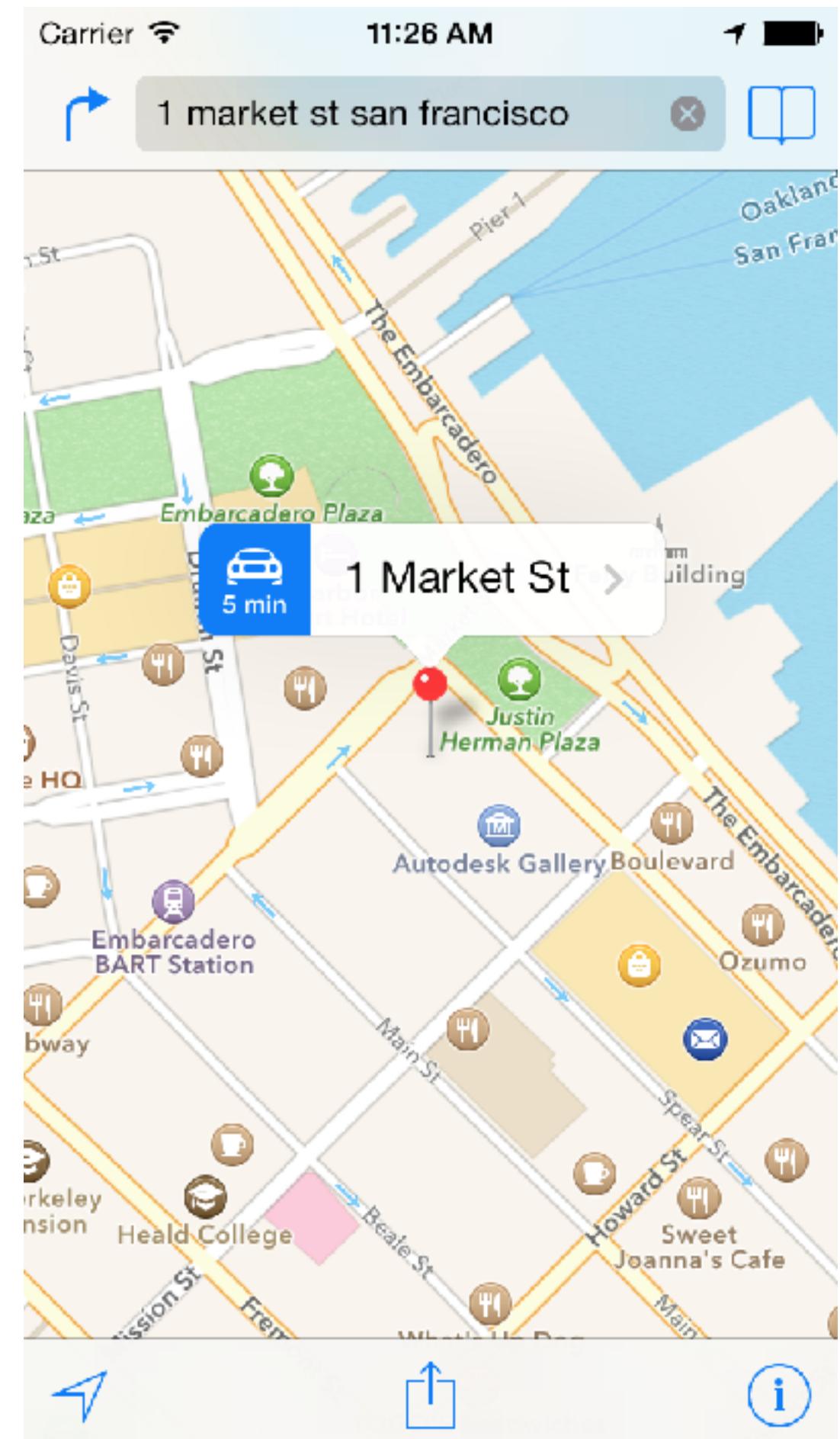
Lecture Summary

- Core Location
- Getting the user's location
- Geocoding
- Map Kit
- Annotating maps
- Segueing programmatically
- Working with JSON



Location-based applications

- Location-based information in iOS involve:
- Location services
- Maps
- Location services are provided by the **Core Location** framework
- Objective-C APIs
- Provides information related to the user's location and heading
- Maps are provided by the **Map Kit** framework
- Support for displaying and annotating maps



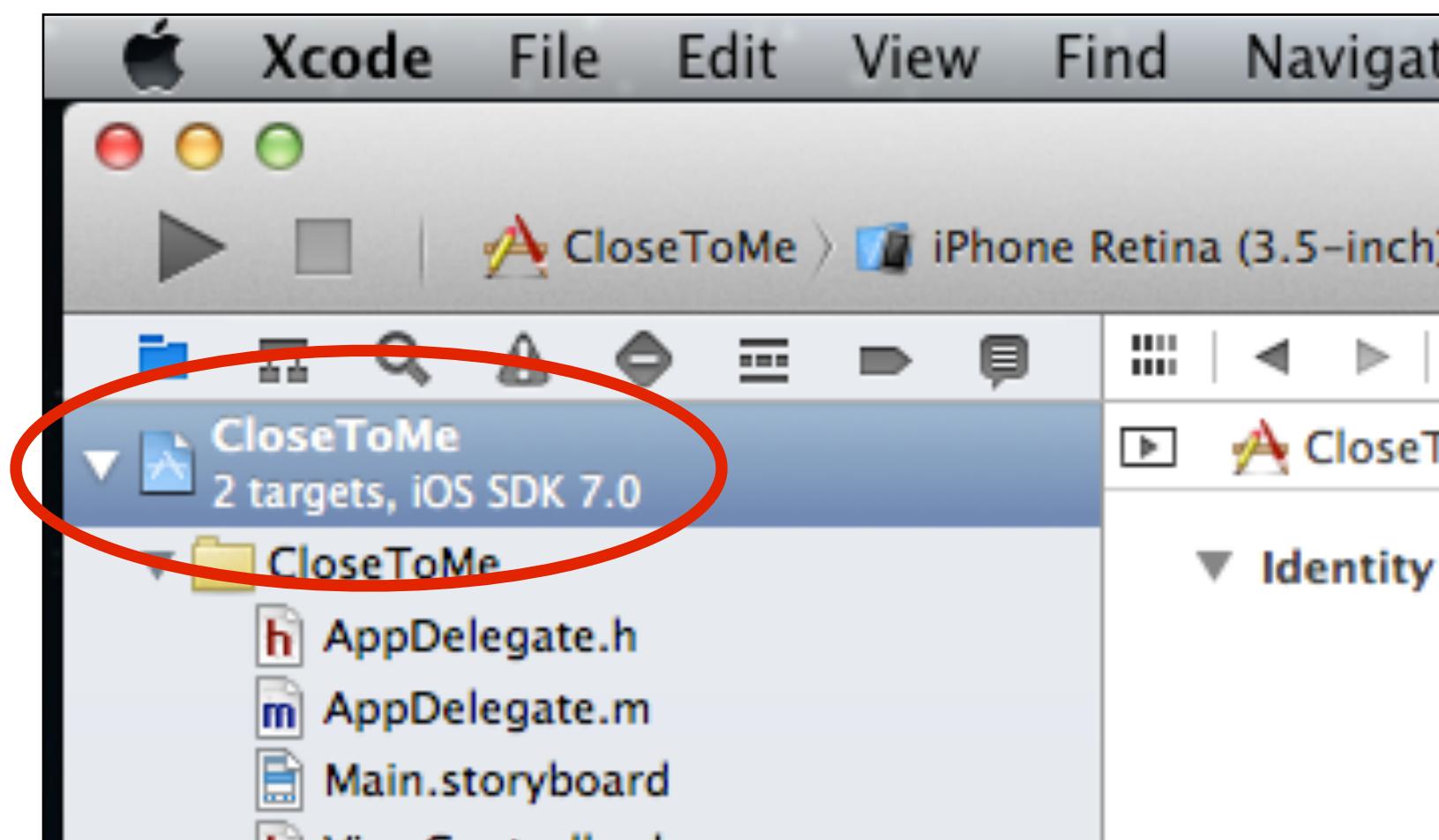


Location services

- iOS applications are intended to be executed on mobile devices
- Location services provide support for mobility
- Location services monitor the user's position and generate updates
- The Core Location framework provides support for location services and must be linked to the project in order to gain access to all the interfaces it provides
- Anytime a class or protocol defined in the Core Location framework is used, the framework must be imported into the file with the following import directive

```
#import <CoreLocation/CoreLocation.h>
```

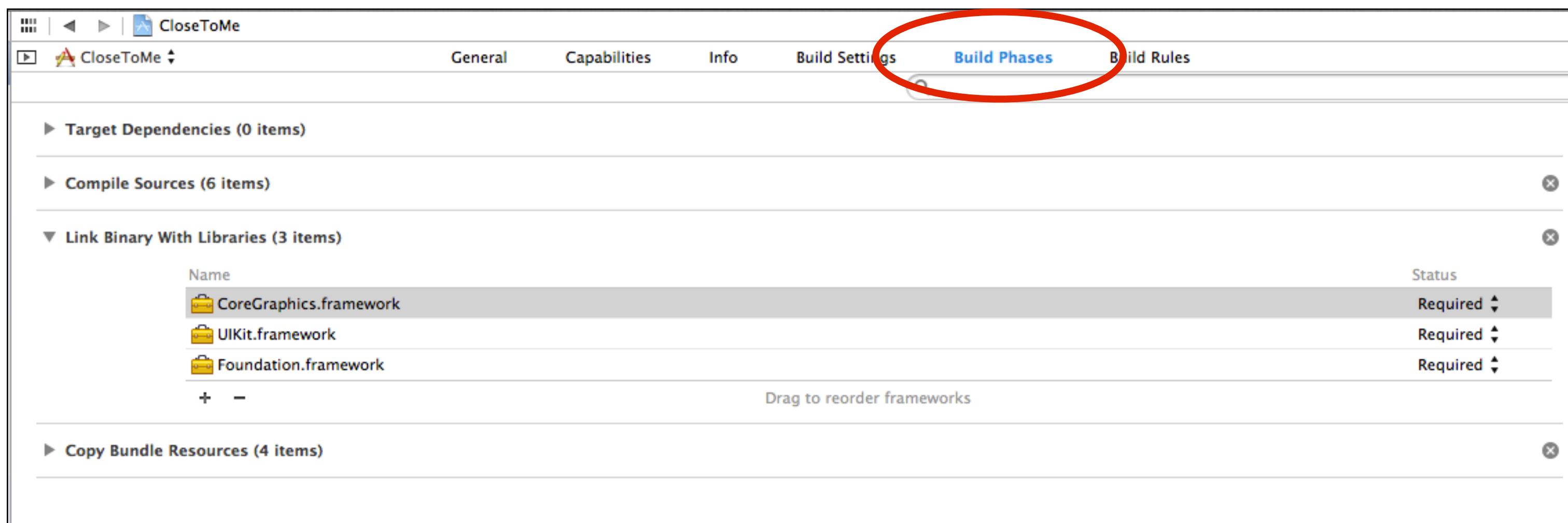
Linking the Core Location framework to your project



Select the project file in the navigator

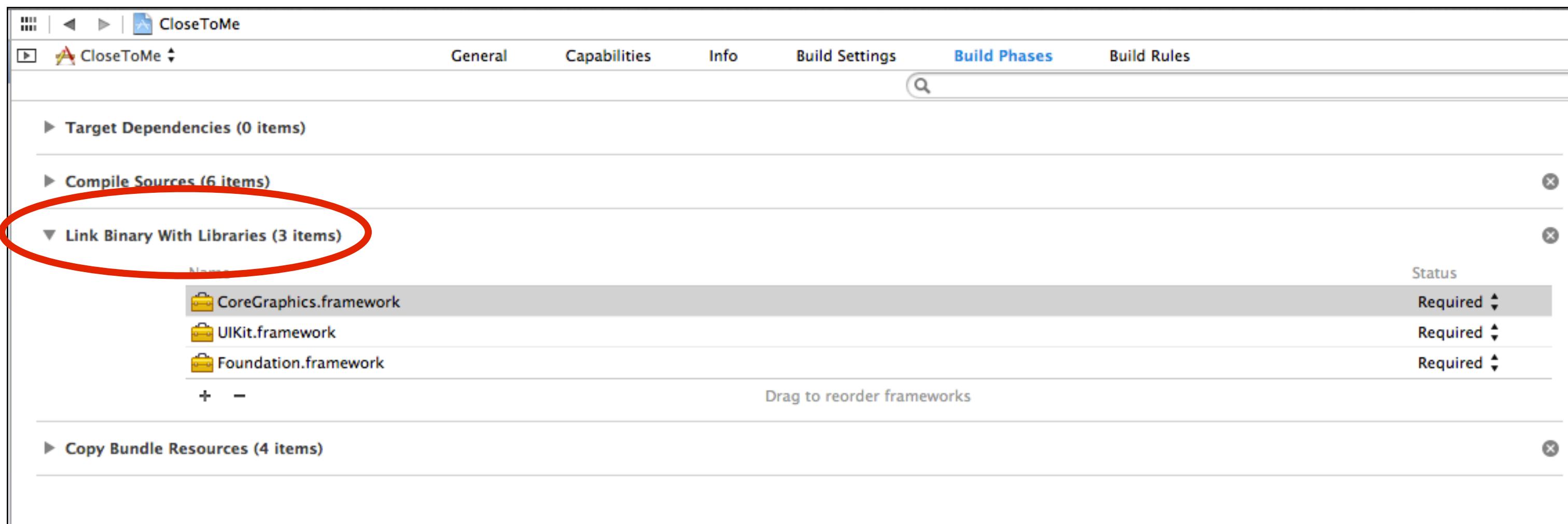


Linking the Core Location framework to your project



Select the **Build Phases** tab

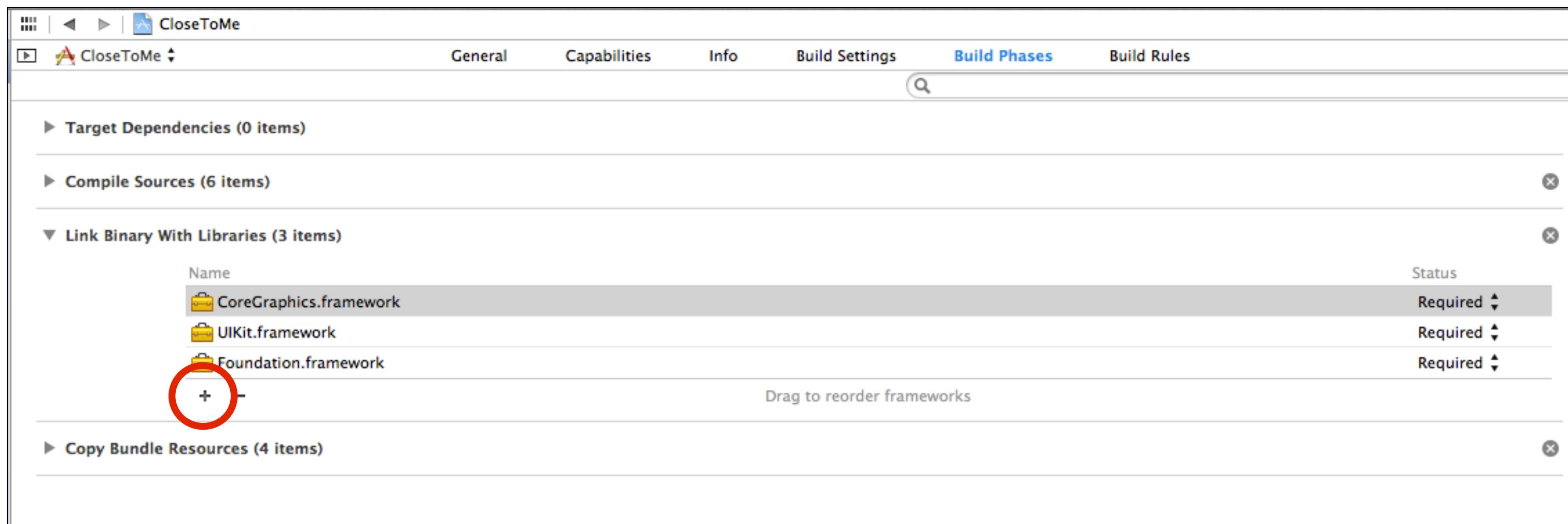
Linking the Core Location framework to your project



Expand the **Link Binary With Libraries** section



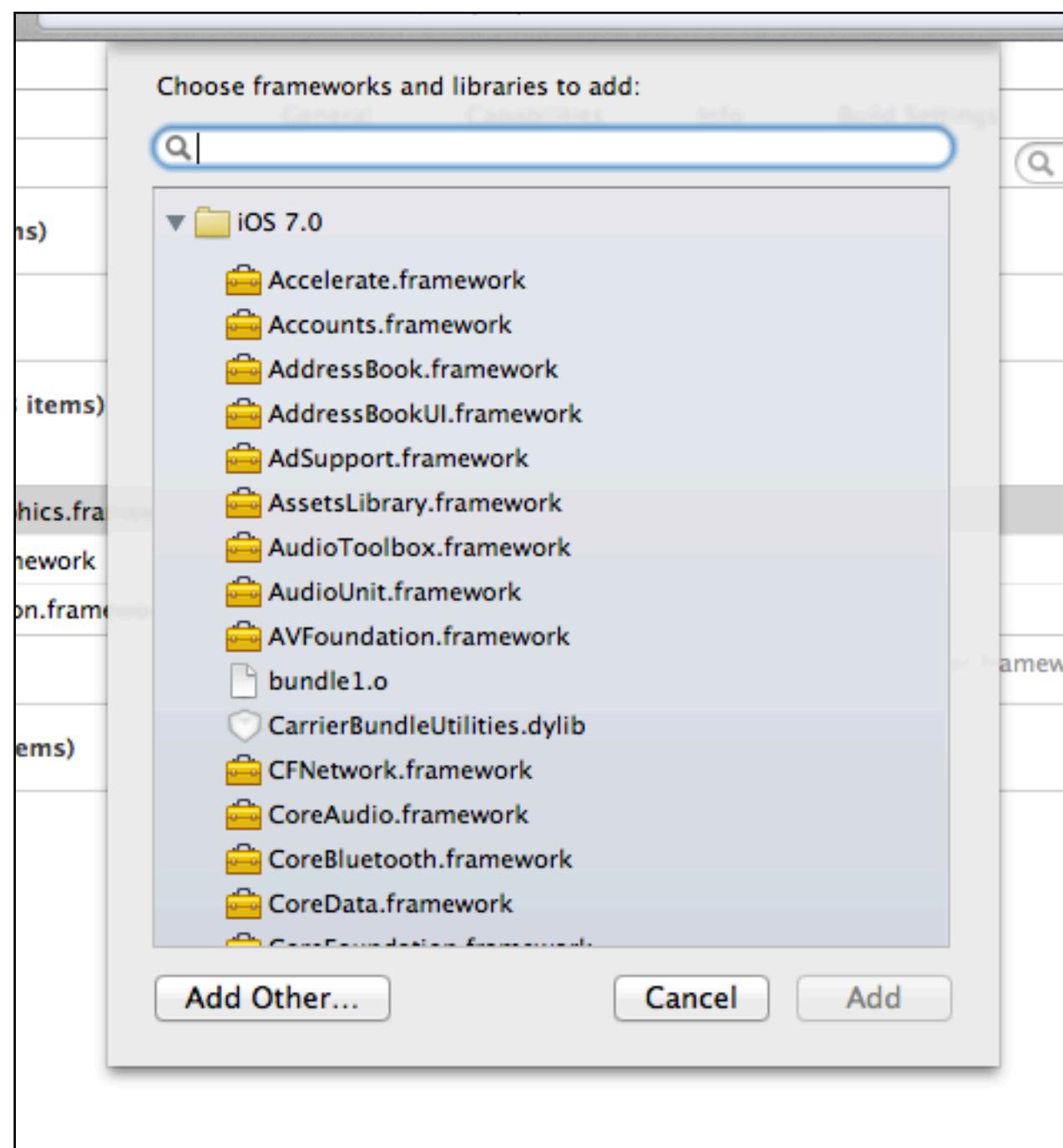
Linking the Core Location framework to your project



Click on the '+' button to add a framework

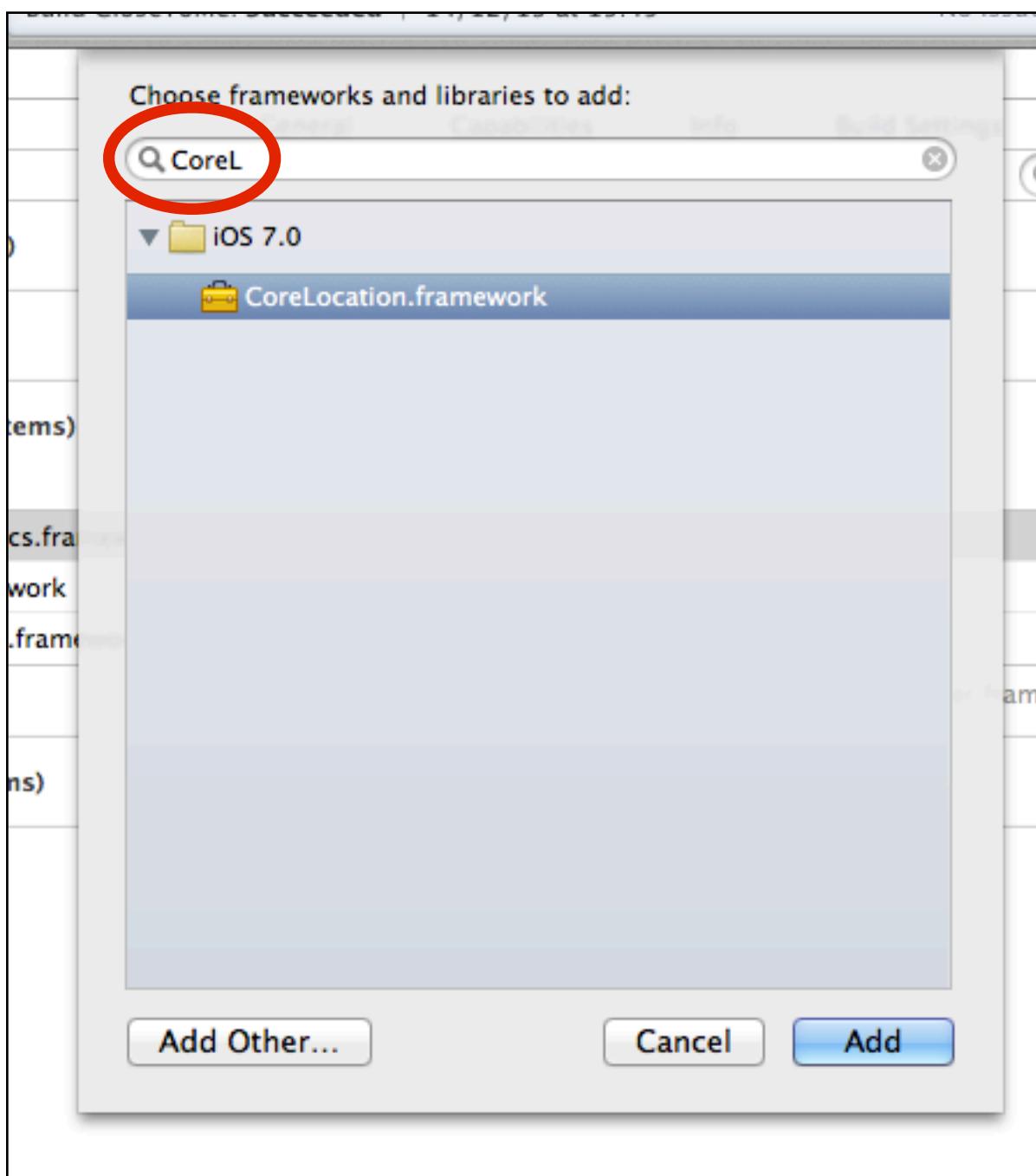


Linking the Core Location framework to your project



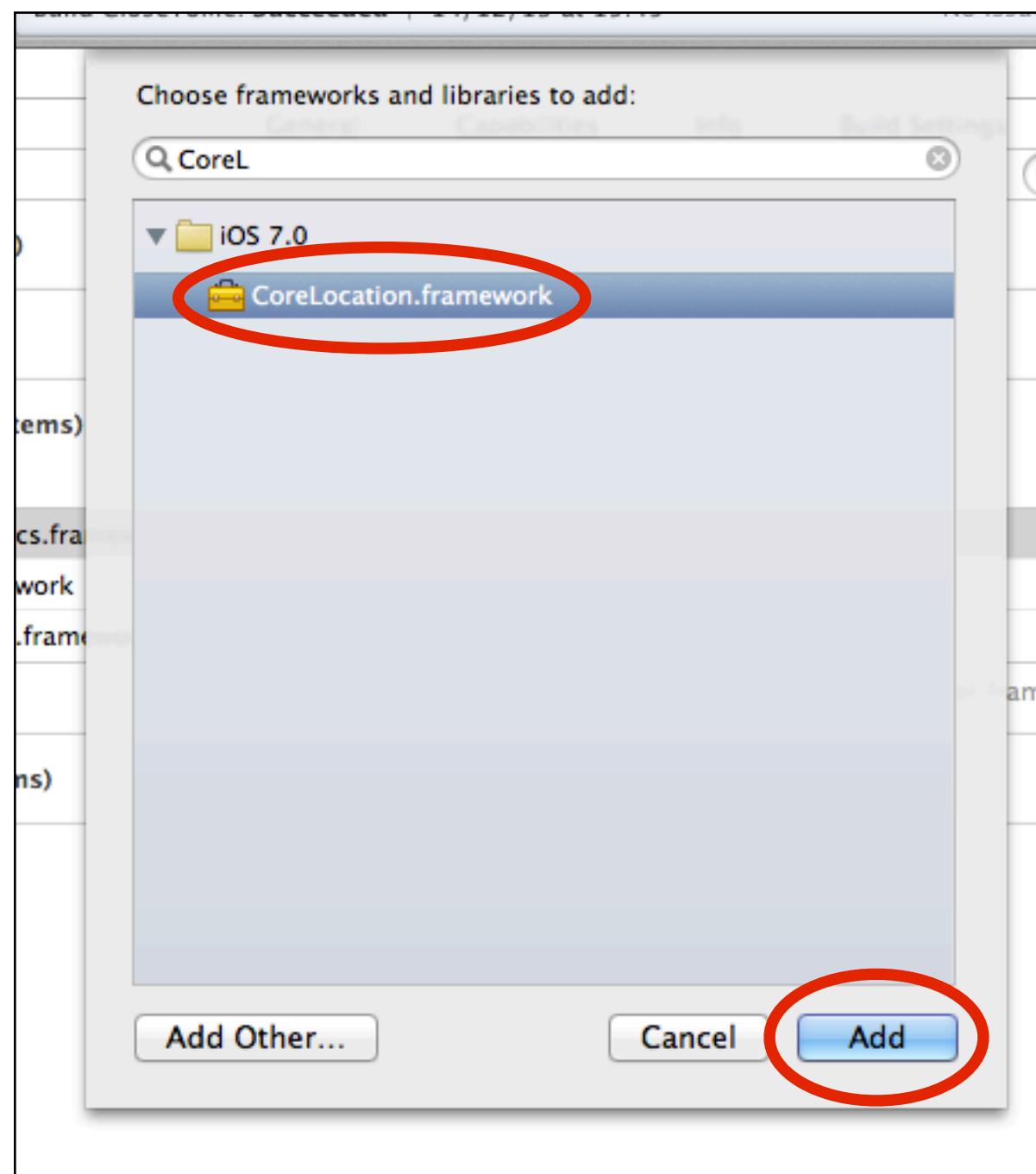
A selection menu appears

Linking the Core Location framework to your project



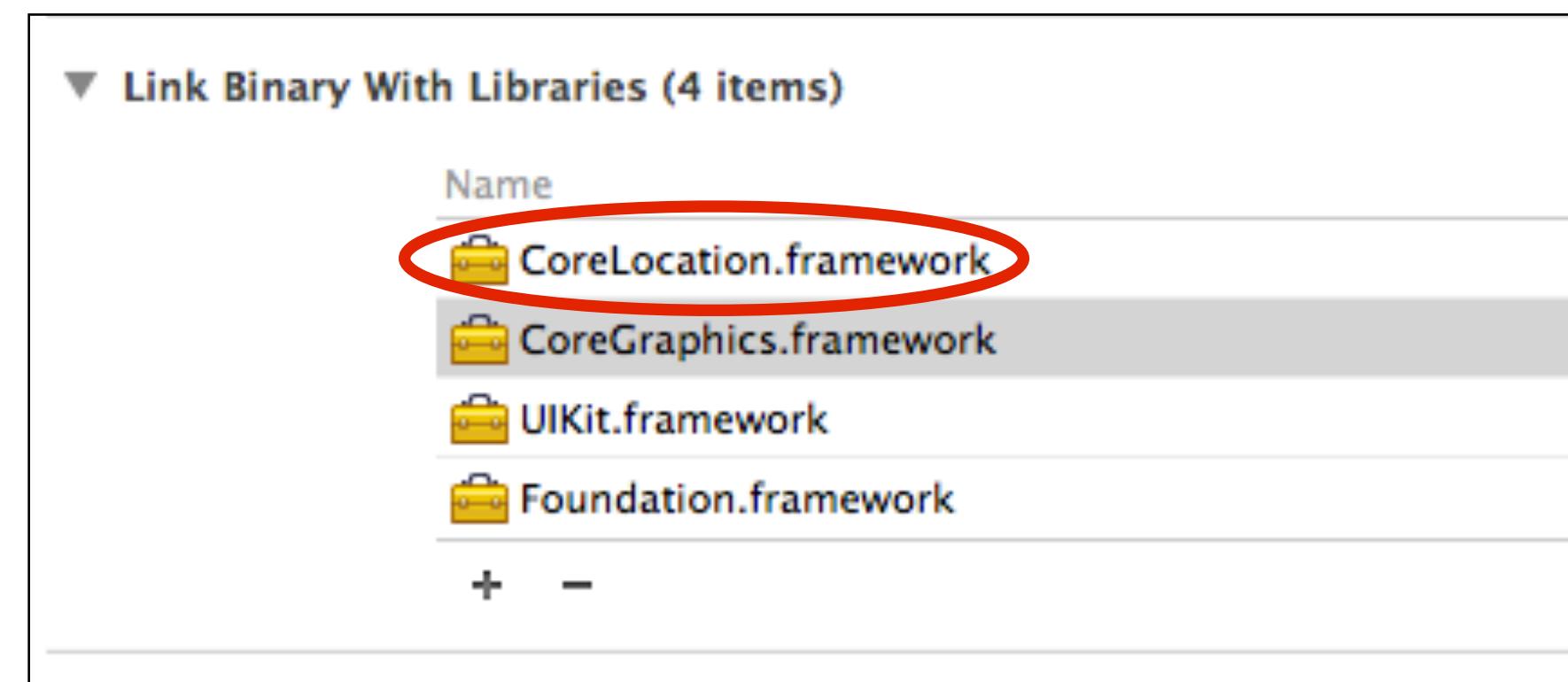
Start typing in the search box for ‘Core Location’

Linking the Core Location framework to your project



Select 'CoreLocation.framework' and then click 'Add'

Linking the Core Location framework to your project



CoreLocation.framework is now linked with your project

Requiring location services in your app

- If the app requires location services to run properly, the **Info.plist** file of the Xcode project should contain an entry for the **Required Device Capabilities** key with the **location-services** value

Key	Type	Value
▼ Information Property List	Dictionary	(14 items)
Localization native development r...	String	Italy
Bundle display name	String	\${PRODUCT_NAME}
Executable file	String	\${EXECUTABLE_NAME}
Bundle identifier	String	it.unipr.ce.mobdev.\${PRODUCT_NAME}:rfc1034identifier
InfoDictionary version	String	6.0
Bundle name	String	\${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone envir...	Boolean	YES
Main storyboard file base name	String	Main
▼ Required device capabilities	Array	(2 items)
Item 0	String	armv7
Item 1	String	location-services
► Supported interface orientations	Array	(3 items)



Getting the user's location

- The Core Location framework provides mechanisms to locate the current position of the device and use that information in the application
- Location information are obtained from the built-in cellular, Wi-Fi, or GPS hardware to triangulate a location fix for the device
- The Core Location service reports the device location to the app and can be configured to generate periodic updates as it receives new or improved data
- There are two different services that can be used to get the device's current location:
 - **standard location service**: configurable, general-purpose mechanism (most common)
 - **significant-change location service**: low-power location service which can wake up an app that is suspended or not running (available only in iOS 4+)
- Be aware that location services are power-intensive and can drain the device battery significantly, so they must be managed properly according to the app's requirements

Configuring the Core Location service

- The Core Location service is managed by an instance of **CLLocationManager**, which is the class that handles location data and notifies the application of location and heading updates
- The **CLLocationManager** can be configured in order to tune in the desired behavior
- An instance of **CLLocationManager** (created with `alloc/init`) can be used:
 1. to configure the parameters that determine when location and heading events should be delivered
 2. to start and stop the actual delivery of those events
- The configuration is performed by setting a number of properties:

`CLLocationAccuracy desiredAccuracy;`
`CLLocationDistance distanceFilter;`
`CLActivityType activityType;`



Setting the desired accuracy

- The `desiredAccuracy` property can be used to set the accuracy of the location data
- This is a hint that is given to the location services: the receiver does its best to achieve the requested accuracy but the actual accuracy is not guaranteed
- It is important to assign a value to this property that is appropriate for your usage scenario: determining a location with greater accuracy requires more time and more power
- Accepted values are of type `CLLocationAccuracy`:

<code>kCLLocationAccuracyBestForNavigation</code>	Use the highest possible accuracy and combine it with additional sensor data
<code>kCLLocationAccuracyBest</code>	Use the highest-level of accuracy
<code>kCLLocationAccuracyNearestTenMeters</code>	Accurate to within ten meters of the desired target
<code>kCLLocationAccuracyHundredMeters</code>	Accurate to within one hundred meters
<code>kCLLocationAccuracyKilometer</code>	Accurate to the nearest kilometer
<code>kCLLocationAccuracyThreeKilometers</code>	Accurate to the nearest three kilometers



Distance filter

- The `distanceFilter` property can be used to configure the minimum distance in meters that a device must travel before an update event is generated
- This distance is measured relative to the previously delivered location
- The constant value `kCLLocationDistanceFilterNone` (default) can be used to be notified of all events with no filter being set
- This property is used only with standard location service and not with significant-change location service



Activity type

- The `activityType` property can be used to configure the type of activity that the user is performing
- This information can be used by the location service to determine when location updates may be automatically paused
- Pausing updates gives the system the opportunity to save power in situations where the user's location is not likely to be changing
- The default value is the constant `CLActivityTypeOther`
- Other constants that can be used:
 - `CLActivityTypeAutomotiveNavigation`
 - `CLActivityTypeOtherNavigation`
 - `CLActivityTypeFitness`



Monitoring location and heading changes

- A CLLocationManager instance can start/stop monitoring location and heading updates with the following methods:

```
[locationManager startUpdatingLocation];
```

```
[locationManager startUpdatingHeading];
```

```
[locationManager stopUpdatingLocation];
```

```
[locationManager stopUpdatingHeading];
```



Monitoring regions

- A `CLLocationManager` instance can also monitor events related to specific regions in space, such as entering or exiting a certain region
- The region-monitoring service can be used to define the boundaries for multiple geographical regions
- To start and stop monitoring for a specific region, the following methods can be used:

```
[locationManager startMonitoringForRegion:region];
```

```
[locationManager stopMonitoringForRegion:region];
```



Being notified: CLLocationManagerDelegate

- The **CLLocationManager** class defines a **delegate** property can be used to set a delegate object which will be notified when a location-related event has been generated
- The delegate must be an instance of a class conforming to the **CLLocationManagerDelegate** protocol
- The **CLLocationManagerDelegate** defines the methods used to receive location and heading updates from a **CLLocationManager**
- The **CLLocation** class represents the location data generated by a **CLLocationManager**
- **CLLocation** instances include the geographical coordinates and altitude of the device's location along with values indicating the accuracy of the measurements and when those measurements were made, as well as heading and speed information



Conforming to CLLocationManagerDelegate

- The `CLLocationManagerDelegate` defines the methods used to receive location and heading updates from a `CLLocationManager`
 - `(void)locationManager:(CLLocationManager *)manager didUpdateLocations:(NSArray *)locations`
 - `(void)locationManager:(CLLocationManager *)manager didUpdateHeading:(CLHeading *)newHeading`
 - `(void)locationManagerDidPauseLocationUpdates:(CLLocationManager *)manager`
 - `(void)locationManagerDidResumeLocationUpdates:(CLLocationManager *)manager`
 - `(void)locationManager:(CLLocationManager *)manager didStartMonitoringForRegion:(CLRegion *)region`
 - `(void)locationManager:(CLLocationManager *)manager didEnterRegion:(CLRegion *)region`
 - `(void)locationManager:(CLLocationManager *)manager didExitRegion:(CLRegion *)region`
 - `(void)locationManager:(CLLocationManager *)manager didFailWithError:(NSError *)error`



Example of location monitoring

```
@interface ViewController ()<CLLocationManagerDelegate>

@property (nonatomic, strong) CLLocationManager *locationManager;

@end

@implementation

- (CLLocationManager *)locationManager{
    if(!_locationManager) _locationManager = [[CLLocationManager alloc] init];
    return _locationManager;
}

- (void)viewDidLoad{
    [super viewDidLoad];
    self.locationManager.desiredAccuracy = kCLLocationAccuracyBest;
    self.locationManager.distanceFilter = 100;
    self.locationManager.delegate = self;
    [self.locationManager startUpdatingLocation];
}

- (void)locationManager:(CLLocationManager *)manager didUpdateLocations:(NSArray *)locations{
    CLLocation *currentLocation = [locations lastObject];
    ...
}

@end
```



Example of location monitoring

```
@interface ViewController ()<CLLocationManagerDelegate>

@property (nonatomic, strong) CLLocationManager *locationManager;

@end

@implementation

- (CLLocationManager *)locationManager{
    if(!_locationManager) _locationManager = [[CLLocationManager alloc] init];
    return _locationManager;
}

- (void)viewDidLoad{
    [super viewDidLoad];
    self.locationManager.desiredAccuracy = kCLLocationAccuracyBest;
    self.locationManager.distanceFilter = 100;
    self.locationManager.delegate = self;
    [self.locationManager startUpdatingLocation];
}

- (void)locationManager:(CLLocationManager *)manager didUpdateLocations:(NSArray *)locations{
    CLLocation *currentLocation = [locations lastObject];
    ...
}

@end
```

Lazy loading
(postpone memory allocation to when it is actually needed)

It is a very good practice for strong properties



Example of location monitoring

```
@interface ViewController ()<CLLocationManagerDelegate>

@property (nonatomic, strong) CLLocationManager *locationManager;

@end

@implementation

- (CLLocationManager *)locationManager{
    if(!_locationManager) _locationManager = [[CLLocationManager alloc] init];
    return _locationManager;
}

- (void)viewDidLoad{
    [super viewDidLoad];
    self.locationManager.desiredAccuracy = kCLLocationAccuracyBest;
    self.locationManager.distanceFilter = 100;
    self.locationManager.delegate = self;
    [self.locationManager startUpdatingLocation];
}

- (void)locationManager:(CLLocationManager *)manager didUpdateLocations:(NSArray *)locations{
    CLLocation *currentLocation = [locations lastObject];
    ...
}

@end
```

Configures the location manager, sets the delegate, and starts receiving location updates



Example of location monitoring

```
@interface ViewController ()<CLLocationManagerDelegate>

@property (nonatomic, strong) CLLocationManager *locationManager;

@end

@implementation

- (CLLocationManager *)locationManager{
    if(!_locationManager) _locationManager = [[CLLocationManager alloc] init];
    return _locationManager;
}

- (void)viewDidLoad{
    [super viewDidLoad];
    self.locationManager.desiredAccuracy = kCLLocationAccuracyBest;
    self.locationManager.distanceFilter = 100;
    self.locationManager.delegate = self;
    [self.locationManager startUpdatingLocation];
}

- (void)locationManager:(CLLocationManager *)manager didUpdateLocations:(NSArray *)locations{
    CLLocation *currentLocation = [locations lastObject];
    ...
}

@end
```

This callback method is called by the location manager each time the location has changed



Simulating locations in the simulator

- It is possible to simulate locations in the simulator, thus avoiding to test your code on the device
- One or more GPX files must added to the project
- GPX files are XML files that have a specific document format

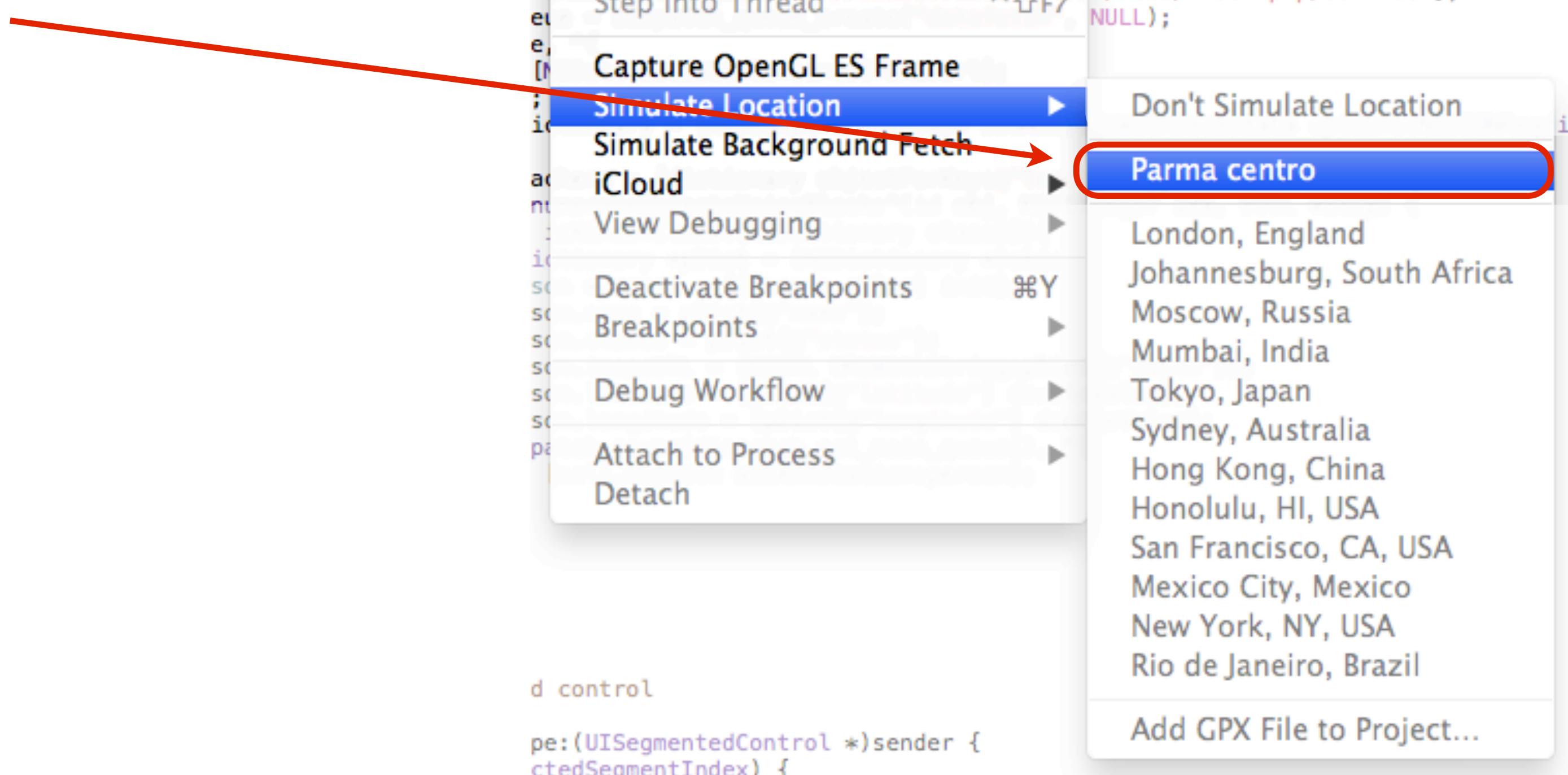
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<gpx
  xmlns="http://www.topografix.com/GPX/1/1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/GPX/1/1/gpx.xsd"
  version="1.1"
  creator="gpx-poi.com">

  <wpt lat="44.801700" lon="10.328012">
    <time>2013-12-10T16:52:28Z</time>
    <name>Parma centro</name>
  </wpt>

</gpx>
```

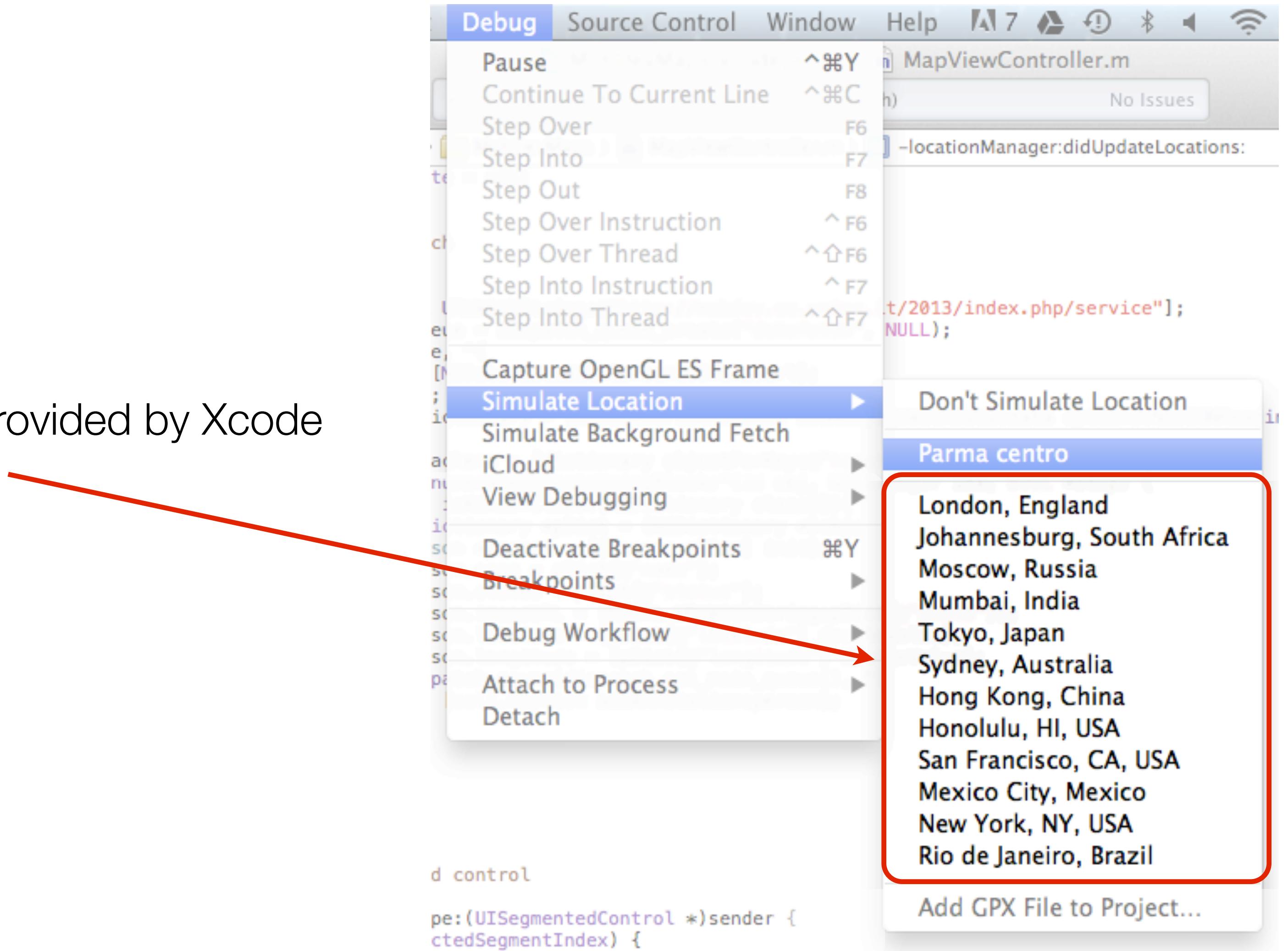
Simulating locations in the simulator

- Once a GPX file has been added to the project it is possible to simulate the user's location in that specific location by using the Debug → Simulate Location menu and then selecting the desired location



Simulating locations in the simulator

- Some locations are already provided by Xcode





Geocoding

- The **CLGeocoder** class provides services for converting between a coordinate (specified as a latitude and longitude) and the user-friendly representation of that coordinate, consisting of:
 - street, city, state, and country
 - a relevant point of interest, landmarks, or other identifying information
- A geocoder object works with a network-based service to look up placemark information for its specified coordinate value
- **Reverse-geocoding** requests take a latitude and longitude value and find a user-readable address
- **Forward-geocoding** requests take a user-readable address and find the corresponding latitude and longitude value
- The results are returned using a **CLPlacemark** object
- Be aware that geocoding requests are rate-limited for each app

Reverse geocoding

- Reverse geocoding can be performed with the following method of an instance of **CLGeocoder**
 - `(void)reverseGeocodeLocation:(CLLocation *)location completionHandler:(CLGeocodeCompletionHandler)completionHandler`
- The **CLGeocodeCompletionHandler** type is defined as

```
typedef void (^CLGeocodeCompletionHandler)(NSArray *placemark, NSError *error)
```

- This method submits the specified location data to the geocoding server asynchronously and returns
- The completion handler block will be executed on the main thread
- For most geocoding requests, the **placemark** array passed in the completion handler should contain only one entry



Reverse geocoding example

```
CLGeocoder *geocoder = [[CLGeocoder alloc] init];  
  
[geocoder reverseGeocodeLocation:location  
    completionHandler:^(NSArray *placemarks, NSError *error) {  
  
    [placemarks enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {  
  
        if([obj isKindOfClass:[CLPlacemark class]]){  
  
            CLPlacemark *pm = (CLPlacemark *)obj;  
            ...  
        }  
    }];  
}];
```

Reverse geocoding example

```
CLGeocoder *geocoder = [[CLGeocoder alloc] init];  
  
[geocoder reverseGeocodeLocation:location  
                      completionHandler:^(NSArray *placemarks, NSError *error) {  
  
    [placemarks enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {  
  
        if([obj isKindOfClass:[CLPlacemark class]]){  
  
            CLPlacemark *pm = (CLPlacemark *)obj;  
            ...  
        }  
    }];  
}];
```

A callout box points from the `enumerateObjectsUsingBlock:` method to the following text:

Convenient method to iterate through a collection



Reverse geocoding example

```
CLGeocoder *geocoder = [[CLGeocoder alloc] init];  
  
[geocoder reverseGeocodeLocation:location  
    completionHandler:^(NSArray *placemarks, NSError *error) {  
  
    [placemarks enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {  
  
        if([obj isKindOfClass:[CLPlacemark class]]){  
  
            CLPlacemark *pm = (CLPlacemark *)obj;  
            ...  
        }  
    }];  
}];
```

The object that has been
extracted from the array
(introspection is needed)

Reverse geocoding example

```
CLGeocoder *geocoder = [[CLGeocoder alloc] init];  
  
[geocoder reverseGeocodeLocation:location  
    completionHandler:^(NSArray *placemarks, NSError *error) {  
  
    [placemarks enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {  
  
        if([obj isKindOfClass:[CLPlacemark class]]){  
            CLPlacemark *pm = (CLPlacemark *)obj;  
            ...  
        }  
    }];  
}];
```

The index of object in the array

Reverse geocoding example

```
CLGeocoder *geocoder = [[CLGeocoder alloc] init];  
  
[geocoder reverseGeocodeLocation:location  
    completionHandler:^(NSArray *placemarks, NSError *error) {  
  
    [placemarks enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {  
  
        if([obj isKindOfClass:[CLPlacemark class]]){  
            CLPlacemark *pm = (CLPlacemark *)obj;  
            ...  
        }  
    }];  
}];
```

A pointer to a BOOL variable
Set it to YES if you want to
stop iterating through the
array:
***stop = YES;**



Map Kit

- The Map Kit framework allows to embed a fully functional map interface into your app
- Map Kit can be used to display street-level and satellite maps
- Maps can be zoomed and panned programmatically
- The framework provides automatic support for the touch events that let users zoom and pan the map
- Maps can be annotated with custom information
- Anytime a class or protocol defined in the Map Kit framework is used, the framework must be imported into the file with the following import directive

```
#import <MapKit/MapKit.h>
```



Add a map to the user interface

- The `MKMapView` class (subclass of `UIView`) is a self-contained interface for presenting map data
- A `MKMapView` can be added:
 - programmatically with `alloc initWithFrame:`
 - in storyboard, by dragging it from the object palette
- A map view's visible area can be configured by setting the `region` property with a `MKCoordinateRegion` struct:

```
MKCoordinateRegion mapRegion;
mapRegion.center = location.coordinate;
mapRegion.span.latitudeDelta = 0.1;
mapRegion.span.longitudeDelta = 0.1;
[self.mapView setRegion:mapRegion animated:YES];
```



Zooming and panning

- To pan (without zooming), it is sufficient to set the `centerCoordinate` property of the map view

```
CLLocationCoordinate2D newCenter = CLLocationCoordinate2DMake(loc.latitude, loc.longitude);  
self.mapView.centerCoordinate = newCenter;
```

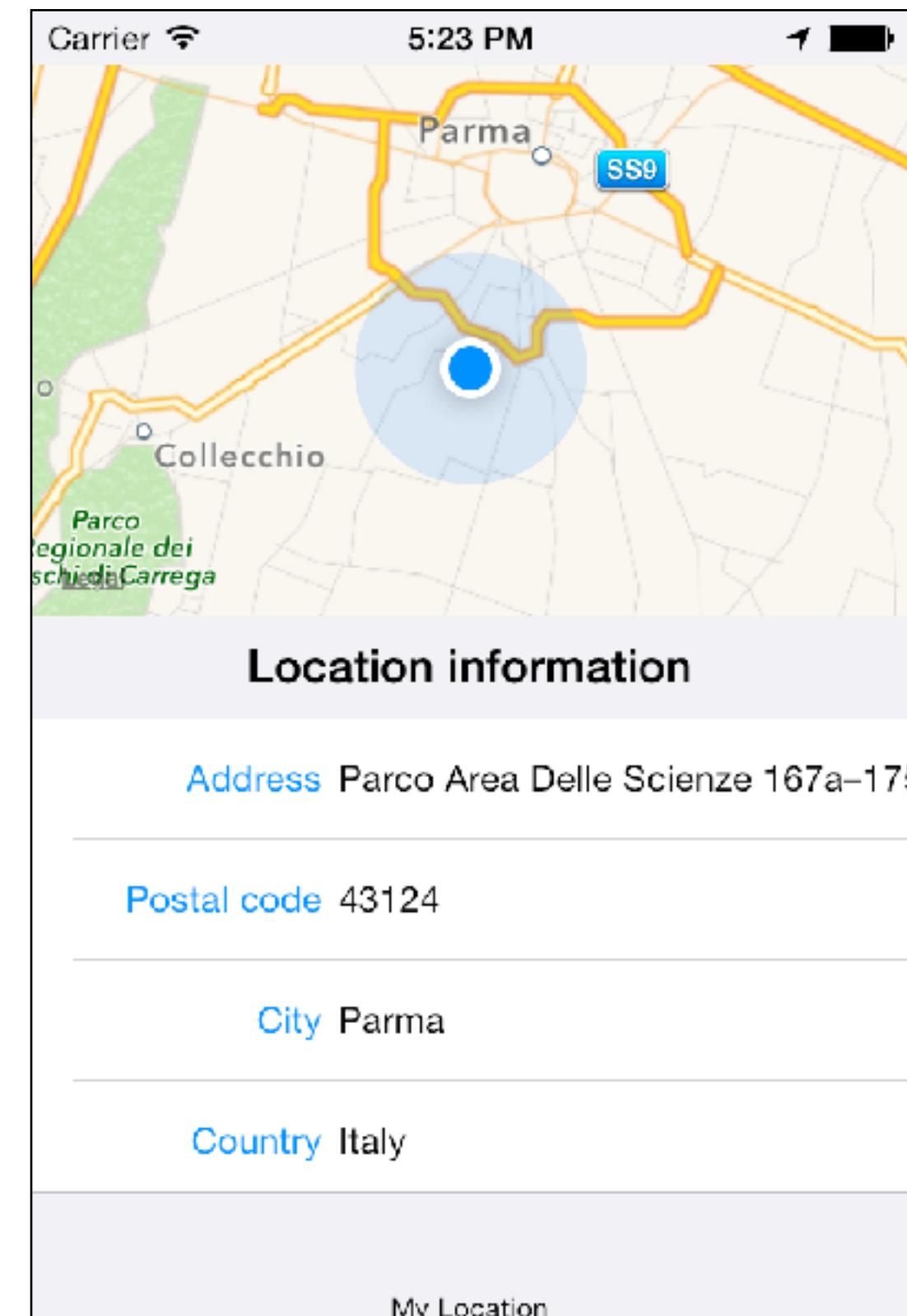
- To zoom, the region property must be reset

```
MKCoordinateRegion mapRegion;  
mapRegion.center = location.coordinate;  
mapRegion.span.latitudeDelta = 0.1;  
mapRegion.span.longitudeDelta = 0.1;  
[self.mapView setRegion:mapRegion animated:YES];
```

User location in map view

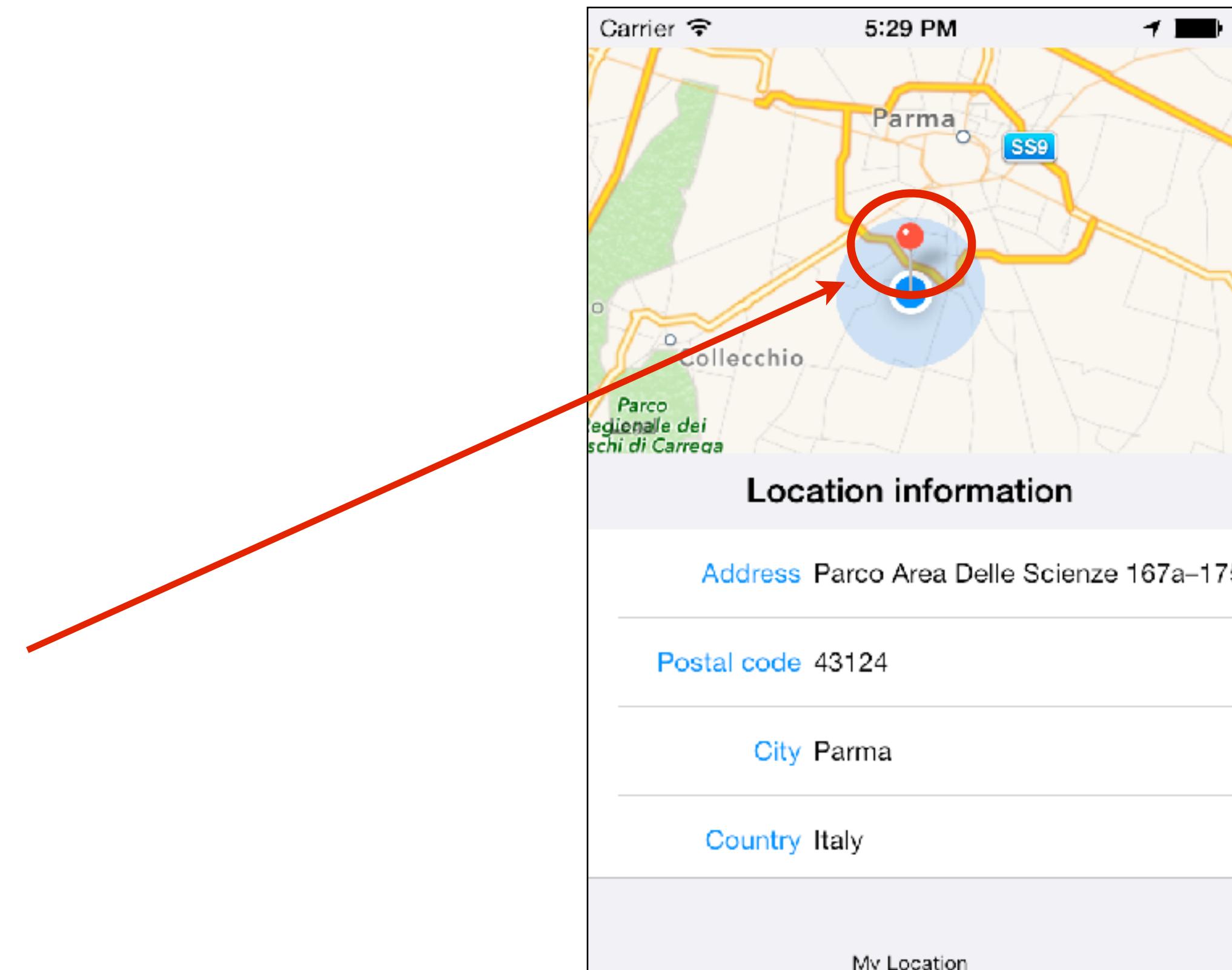
- To display the user location in the map view, the `showUserLocation` property of the map view must be set to **YES**

```
self.mapView.showsUserLocation = YES;
```



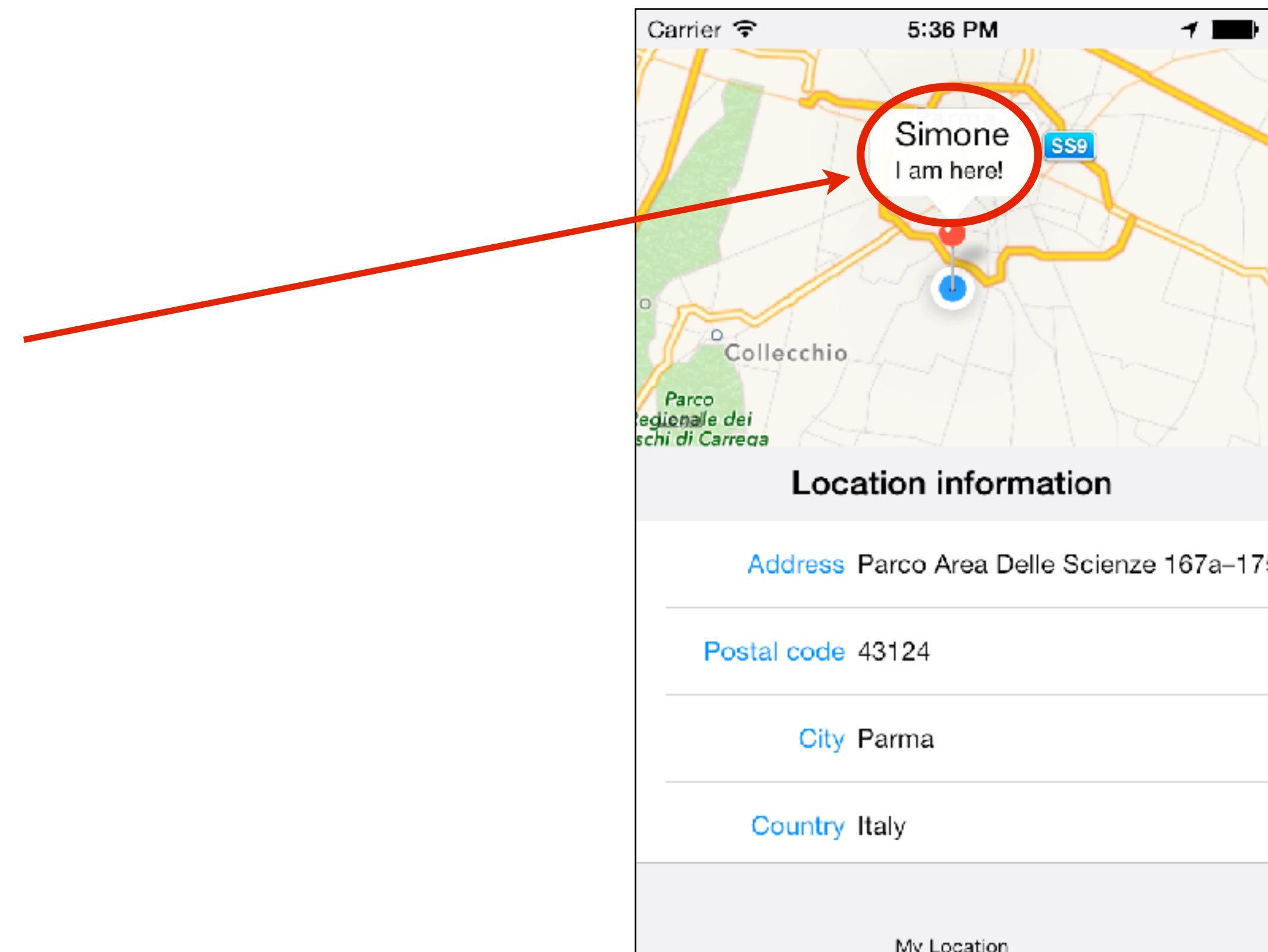
Annotating maps

- A map view can have annotations
- An annotation include a coordinate (latitude and longitude), a title, and a subtitle
- Annotations are displayed using an instance of MKAnnotationView



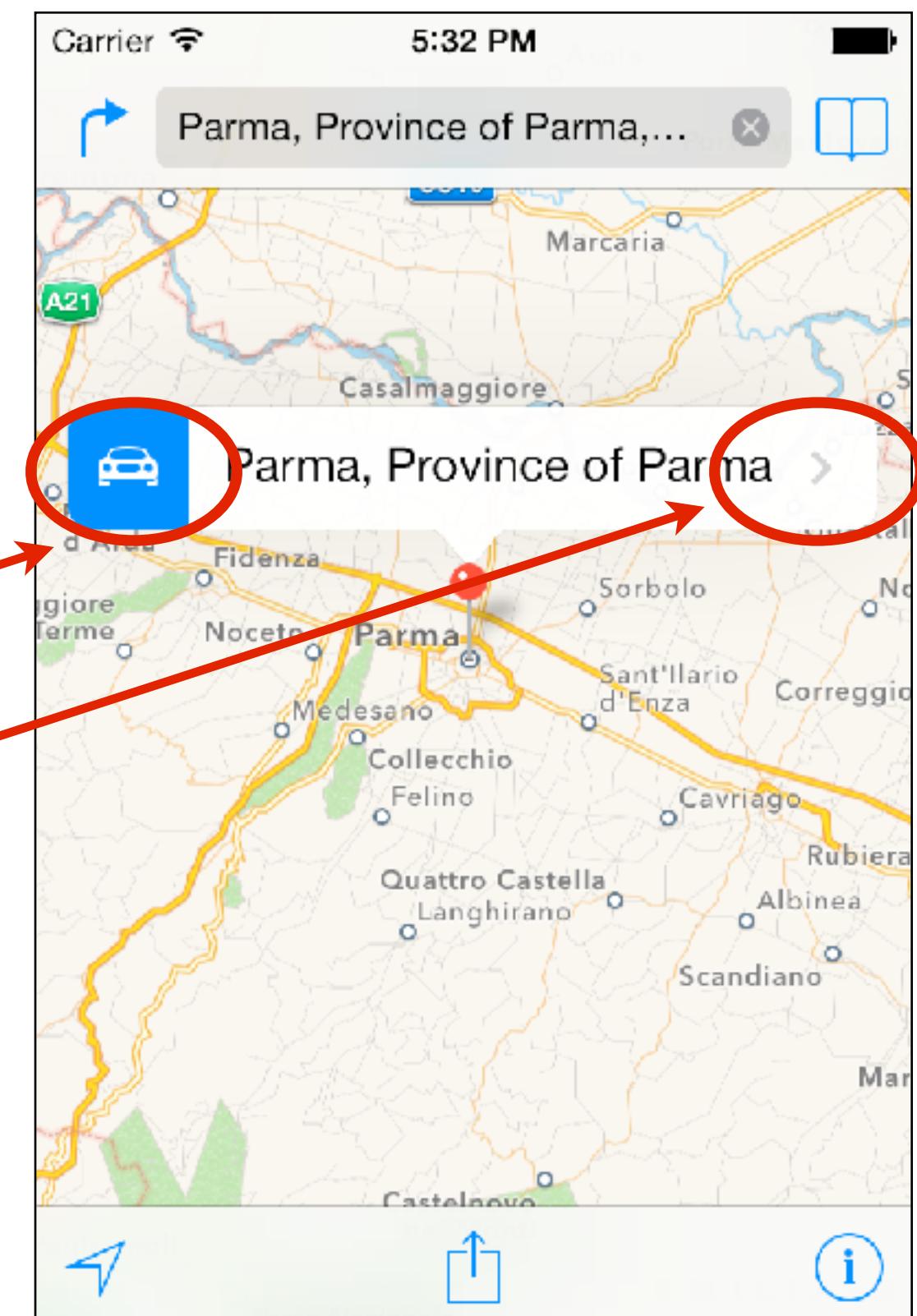
Annotating maps

- Annotations can also display a callout
- A callout displays the title and subtitle by default



Annotating maps

- Annotations can also display a callout
- A callout displays the title and subtitle by default
- A callout has also accessory view to further customize its contents:
 - `leftCalloutAccessoryView`
 - `rightCalloutAccessoryView`



Annotating maps

- The annotations that can be added to a map view are objects that conform to the **MKAnnotation** protocol
- Map view has a readonly property called **annotation** which holds all the annotations currently in the map
- An **MKAnnotation** has an associated **MKAnnotationView** which is actually displayed on the screen
- The **MKAnnotation** protocol defines:
 - a required property **coordinate** of type **CLLocationCoordinate2D**
 - two optional properties **title** and **subtitle** of type **NSString***





Using categories to annotate maps

- A great way to provide annotation capabilities to a class is to use categories
- By doing so, we add to the class the methods required by the MKAnnotation protocol, without touching the original class

```
@interface MDCheckin : NSObject  
  
@property (nonatomic, strong, readonly) MDPOI *poi;  
  
@end
```

MDCheckin.h

```
#import "MDCheckin.h"  
#import <MapKit/MapKit.h>  
  
@interface MDCheckin (Annotation)<MKAnnotation>  
  
@end
```

MDCheckin+Annotation.h

Using categories to annotate maps

- A great way to provide annotation capabilities to a class is to use categories
- By doing so, we add to the class the methods required by the `MKAnnotation` protocol, without touching the original class

```
@interface MDCheckin : NSObject  
  
@property (nonatomic, strong, readonly) MDPOI *poi;  
  
@end
```

MDCheckin.h

```
#import "MDCheckin.h"  
#import <MapKit/MapKit.h>  
  
@interface MDCheckin (Annotation) <MKAnnotation>  
  
@end
```

MDCheckin+Annotation.h

Simply declare the category to conform to the **MKAnnotation** protocol



Using categories to annotate maps

```
#import "MDCheckin+Annotation.h"

@implementation MDCheckin (Annotation)

- (CLLocationCoordinate2D)coordinate{
    CLLocationCoordinate2D coordinate;
    coordinate.latitude = self.poi.latitude;
    coordinate.longitude = self.poi.longitude;
    return coordinate;
}

- (NSString *)title{
    return self.poi.name;
}

@end
```

Implement the
`@required`
coordinate
method

MDCheckin+Annotation.m



Using categories to annotate maps

```
#import "MDCheckin+Annotation.h"

@implementation MDCheckin (Annotation)

- (CLLocationCoordinate2D)coordinate{
    CLLocationCoordinate2D coordinate;
    coordinate.latitude = self.poi.latitude;
    coordinate.longitude = self.poi.longitude;
    return coordinate;
}

- (NSString *)title{
    return self.poi.name;
}

@end
```

MDCheckin+Annotation.m

Implement the
`@optional title`
method

In this case we do not
implement the
`@optional subtitle` method

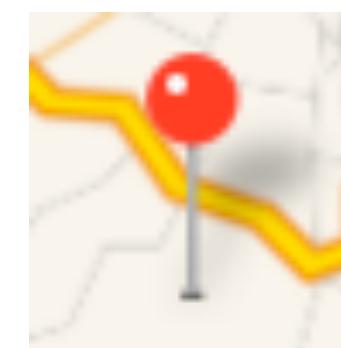


Annotating maps

- Since the map view's `annotation` property is readonly, proper methods must be used to add them to and remove them from a map view
 - `(void)addAnnotation:(id<MKAnnotation>)annotation`
 - `(void)addAnnotations:(NSArray *)annotations`
 - `(void)removeAnnotation:(id<MKAnnotation>)annotation`
 - `(void)removeAnnotations:(NSArray *)annotations`
- For performance reasons, a good practice is to put all annotations on the map right away (if possible)
- `MKAnnotationView` instances are reused by `MKMapView`, similarly to what `UITableViewCells` do with `UITableViewCell`s

Annotating maps

- The standard way to put annotation on a map is by using the `MKPinAnnotationView`
- It is possible to use any subclass of `MKAnnotationView` to customize the look of the annotation view
- When an annotation is tapped, a callout might be displayed if its `canShowCallout` property has been set to **YES**





MKMapViewDelegate

- The **MKMapView** has a **delegate** property that can be used to set a delegate object that can optionally receive events related to the map view and be responsible to provide concrete **MKAnnotationView** instances for a given annotation
- The delegate must conform to the **MKMapViewDelegate** protocol
- The methods of the protocol allow to:
 - respond to map position changes
 - respond to loading map data events
 - respond to user location changes
 - manage annotation views (can be used to customize the annotation's callout)
 - respond to annotation views drag events
 - respond to selection of annotation views (can be used to lazily initialize the callout's subviews)



Customizing annotation views

- `(MKAnnotationView *)mapView:(MKMapView *)mapView
viewForAnnotation:(id<MKAnnotation>)annotation`

- This method gets called any time an annotation must be put on the map view
- It asks to provide a custom **MKAnnotationView** for the given annotation
- It is very similar to **cellForRowAtIndexPath:** in **UITableViewDataSource**
- This is where it is possible to customize the annotation view



Customizing annotation views

```
- (MKAnnotationView *)mapView:(MKMapView *)mapView
    viewForAnnotation:(id<MKAnnotation>)annotation{

    static NSString *AnnotationIdentifier = @"ViewController";

    MKAnnotationView *view = [mapView
        dequeueReusableCellWithIdentifier:AnnotationIdentifier];
    if(!view){
        view = [[MKPinAnnotationView alloc] initWithAnnotation:annotation
            reuseIdentifier:AnnotationIdentifier];
        view.canShowCallout = YES;
    }

    view.annotation = annotation;

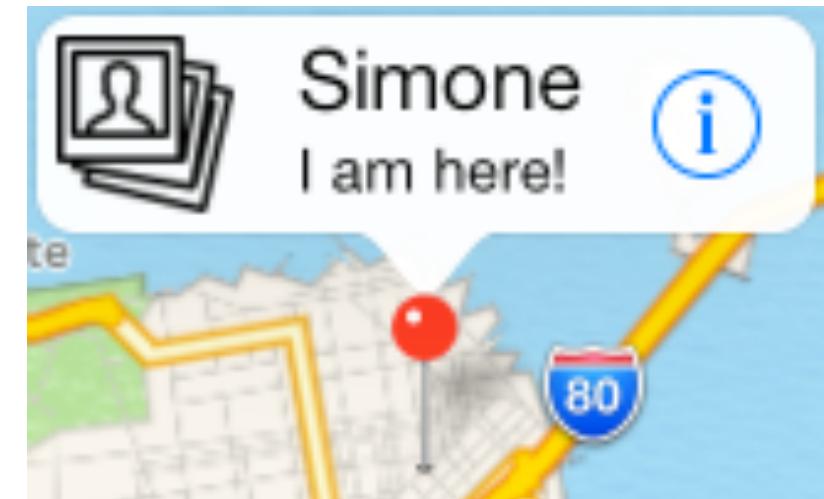
    UIImageView *imageView = [[UIImageView alloc] initWithFrame:CGRectMake(0, 0, 36, 36)];
    imageView.image = [UIImage imageNamed:@"photo"];
    view.leftCalloutAccessoryView = imageView;

    view.rightCalloutAccessoryView = [UIButton buttonWithType:UIButtonTypeInfoDark];

    return view;
}
```

Customizing annotation views

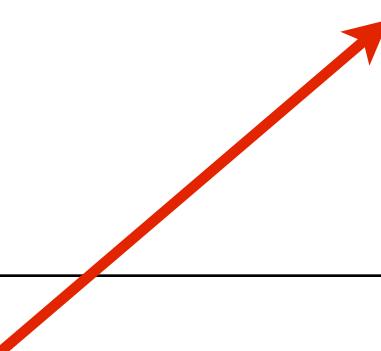
```
- (MKAnnotationView *)mapView:(MKMapView *)mapView
    viewForAnnotation:(id<MKAnnotation>)annotation{
    static NSString *AnnotationIdentifier = @"ViewController";
    MKAnnotationView *view = [mapView
        dequeueReusableCellWithIdentifier:AnnotationIdentifier];
    if(!view){
        view = [[MKPinAnnotationView alloc] initWithAnnotation:annotation
            reuseIdentifier:AnnotationIdentifier];
        view.canShowCallout = YES;
    }
    view.annotation = annotation;
    UIImageView *imageView = [[UIImageView alloc] initWithFrame:CGRectMake(0, 0, 36, 36)];
    imageView.image = [UIImage imageNamed:@"photo"];
    view.leftCalloutAccessoryView = imageView;
    view.rightCalloutAccessoryView = [UIButton buttonWithType:UIButtonTypeInfoDark];
    return view;
}
```



Responding to selection of an annotation view

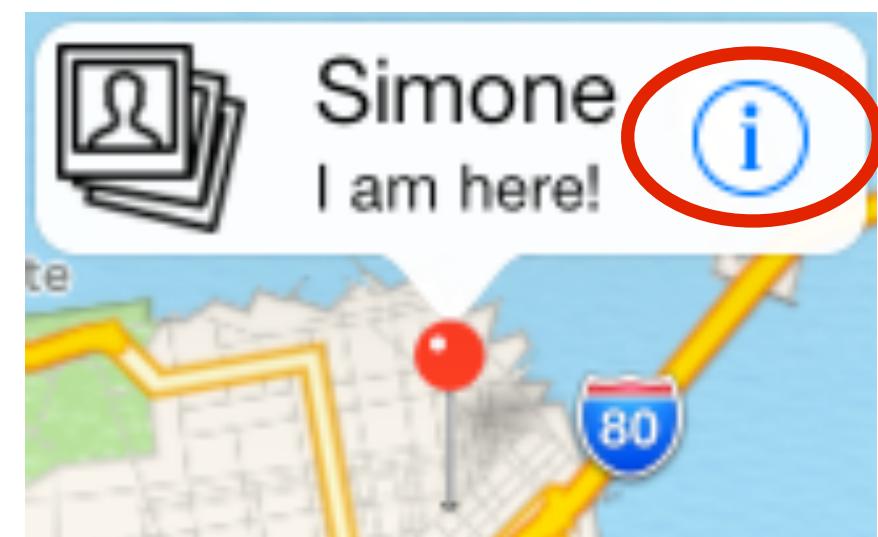
- The delegate gets notified when an annotation (the pin) is selected
- Typically, if the `canShowCallout` property has been set to YES, a callout will then be shown
- This is a good place to perform lazy initialization of the content of the callout so that it is loaded only when requested

```
- (void)mapView:(MKMapView *)mapView didSelectAnnotationView:(MKAnnotationView *)view{  
  
    if([view.leftCalloutAccessoryView isKindOfClass:[UIImageView class]]){  
        UIImageView *imageView = (UIImageView *)view.leftCalloutAccessoryView;  
        imageView.image = ...; //load from network using GCD (not in main queue!)  
    }  
}
```



be aware of annotation reuse!

Responding to tap in the callout



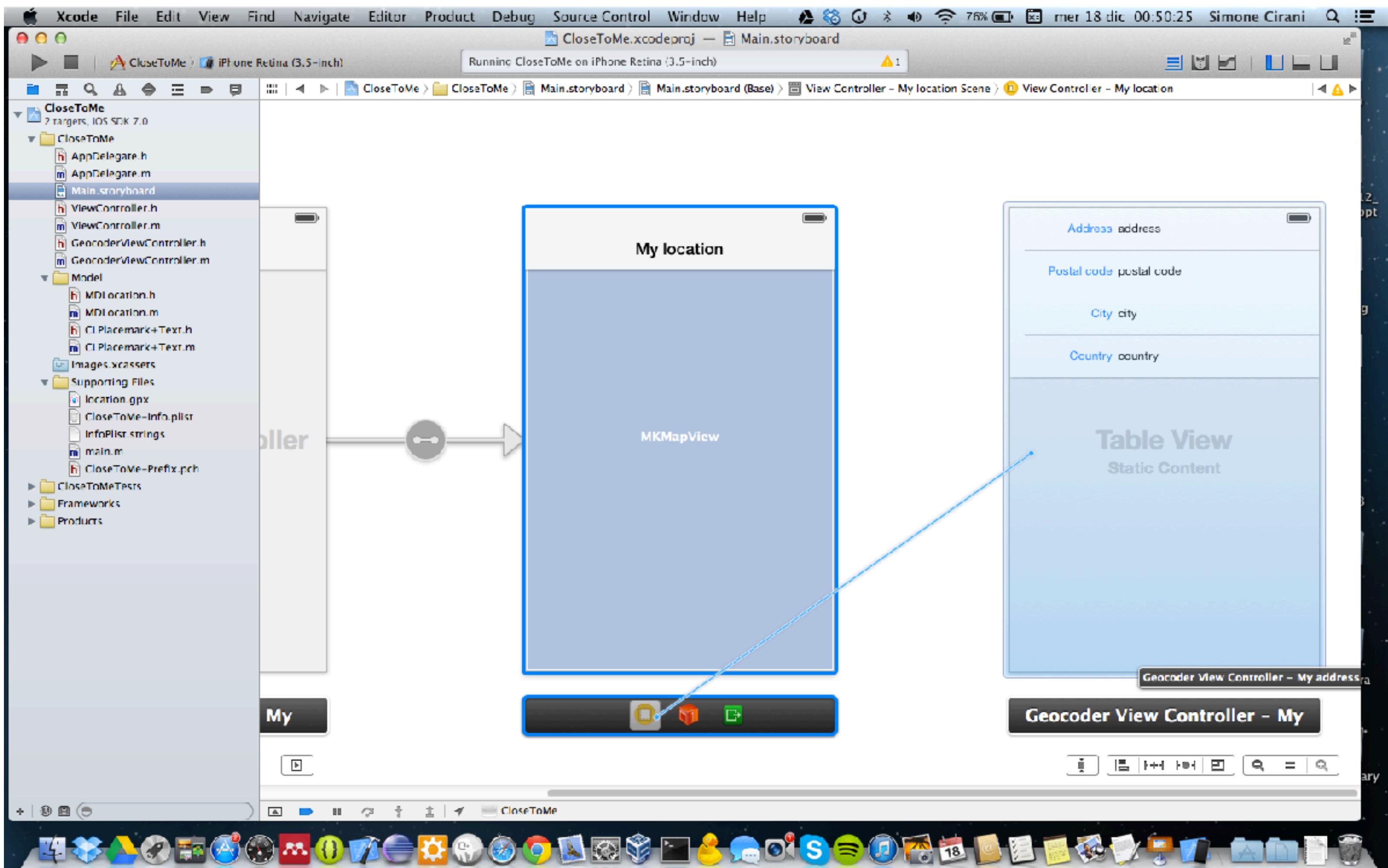
- If an accessory view of the callout is a UIControl, the delegate gets notified when the control is tapped with the following method
 - `(void)mapView:(MKMapView *)mapView annotationView:(MKAnnotationView *)view calloutAccessoryControlTapped:(UIControl *)control`
- At this point it is possible to segue to another view controller (e.g. to show some details) but this cannot be done in storyboard as usual since there is no object in storyboard that represents this callout
- To cope with this problem, we need to segue programmatically



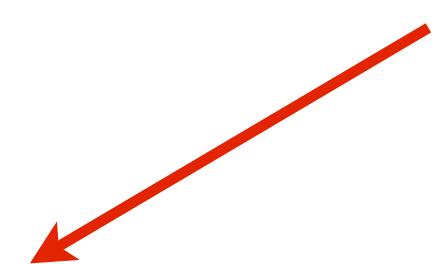
Segueing programmatically

- Sometimes it is not possible to define a segue directly in storyboard just because the element that we are segueing from does not exist in storyboard, such as a callout of a map annotation
- It is of course possible to segue from such an element, but it has to be done programmatically
- The procedure to segue from code is:
 1. in storyboard, create a (manual) segue by control-dragging from the origin view controller (not a specific control in the view) to the destination view controller
 2. assign an identifier to the segue in order to be able to refer to the segue in code
 3. execute the `UIViewController's performSegueWithIdentifier:sender:` method specifying the identifier given in step 2 and setting the correct sender

Segueing programmatically

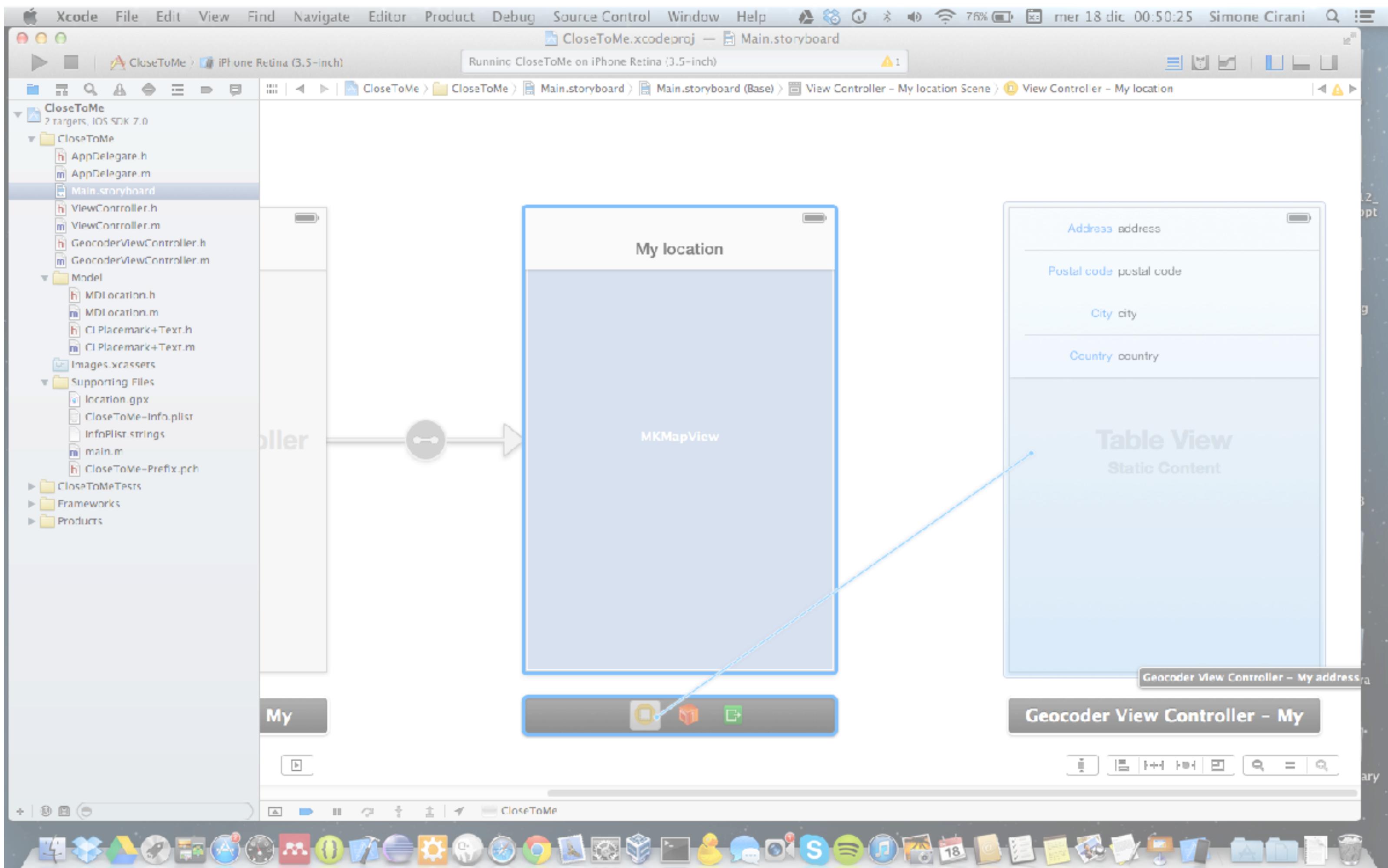


In storyboard, create a (manual) segue by control-dragging from the origin view controller to the destination view controller





Segueing programmatically



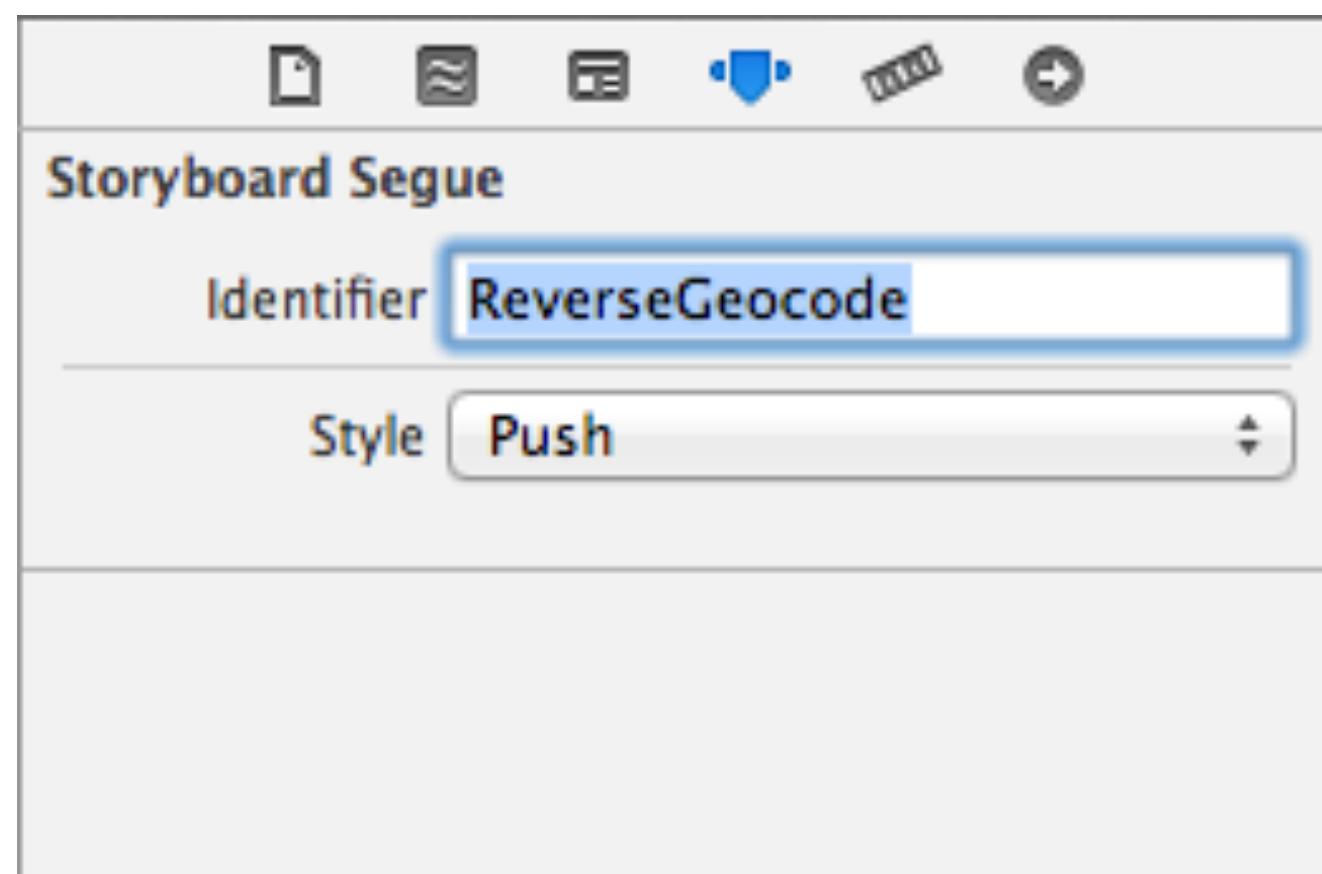
In storyboard, create a (manual) segue by control-dragging from the origin view controller to the destination view controller



Manual Segue
push
modal
custom



Segueing programmatically



Assign an identifier to the segue in order to be able to refer to the segue in code



Segueing programmatically

Call `performSegueWithIdentifier:sender:` method specifying the identifier and setting the correct sender



```
- (void)mapView:(MKMapView *)mapView  
annotationView:(MKAnnotationView *)view  
calloutAccessoryControlTapped:(UIControl *)control{  
  
    [self performSegueWithIdentifier:@"ReverseGeocode" sender:view];  
  
}
```



Working with JSON

- Working with JSON in iOS is pretty easy
- The **NSJSONSerialization** class provides methods to serialize objects to JSON strings and deserialize JSON strings to objects
- An object that may be converted to JSON must have the following properties:
 - The top level object is an **NSArray** or **NSDictionary**
 - All objects are instances of **NSString**, **NSNumber**, **NSArray**, **NSDictionary**, or **NSNull**
 - All dictionary keys are instances of **NSString**
 - Numbers are not NaN or infinity

Serialize to JSON

```
id obj = ...;
NSError *error;
if([NSJSONSerialization isValidJSONObject:obj]){

    NSData *data = [NSJSONSerialization dataWithJSONObject:obj
                                                options:0
                                              error:&error];

}
```

- It is best to check whether an object can be serialized with the `isValidJSONObject:` method
- The method return JSON data for `obj`, or `nil` if an error occurs
- The resulting data is a encoded in UTF-8
- If an error occurs, the `error` variable holds the error

Deserialize from JSON

```
NSData *data = ...;
NSError * error;
id obj = [NSJSONSerialization JSONObjectWithData:data
                                             options:NSJSONReadingAllowFragment
                                               error:&error];
```

- The returned object may be a **NSArray*** or a **NSDictionary***, or **nil** if an error occurs
- If an error occurs, the **error** variable holds the error



Deserialization example

- The following URL exposes a service which returns some JSON data:

<http://mobdev.ce.unipr.it/2013/index.php/service>

- The JSON data has the following format:

```
{  
  timestamp: 1387350841,  
  course: "Mobile Application Development 2013",  
  teachers:  
    [  
      {  
        name: "Simone Cirani",  
        status: "Teaching iOS",  
        photo: "http://www.tlc.unipr.it/cirani/images/simone.png",  
        latitude: 44.12345,  
        longitude: 10.12345  
      },  
      {  
        name: "Marco Picone",  
        status: "Teaching Android",  
        photo: "http://wasnlab.tlc.unipr.it/people/picone/img/picone\_150\_200.png",  
        latitude: 44.12345,  
        longitude: 10.12345  
      }  
    ]  
}
```



Deserialization example

- Suppose that we are interested in parsing the “teachers” field and create objects of a class Person defined as follows:

```
@interface Person : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSString *status;
@property (nonatomic, strong) NSURL *imageURL;
@property (nonatomic, readwrite) double latitude;
@property (nonatomic, readwrite) double longitude;

@end
```

Deserialization example

- The JSON object and the class are mapped as follows:

```
{  
  timestamp: 1387350841,  
  course: "Mobile Application Development 2013",  
  teachers:  
  [  
    {  
      name: "Simone Cirani",  
      status: "Teaching iOS",  
      photo: "http://www.tlc.unipr.it/cirani/images/simone.png",  
      latitude: 44.12345,  
      longitude: 10.12345  
    },  
    {  
      name: "Marco Picone",  
      status: "Teaching Android",  
      photo: "http://wasnlab.tlc.unipr.it/people/picone/img/picone 150 200.png",  
      latitude: 44.12345,  
      longitude: 10.12345  
    }  
  ]  
}
```

```
@interface Person : NSObject  
@property (nonatomic, strong) NSString *name;  
@property (nonatomic, strong) NSString *status;  
@property (nonatomic, strong) NSURL *imageURL;  
@property (nonatomic, readwrite) double latitude;  
@property (nonatomic, readwrite) double longitude;  
  
@end
```



Deserialization example

- The following snippet shows how to deserialize to the instance of Person:

```
NSData *data = [NSData dataWithContentsOfURL:url];
NSError * error;
NSDictionary *dictionary = [NSJSONSerialization JSONObjectWithData:data
options:NSJSONReadingAllowFragments error:&error];
if(!error){
    NSArray *teachers = [dictionary objectForKey:@"teachers"];
    [teachers enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
        if( [obj isKindOfClass:[NSDictionary class]]){
            NSDictionary *pDict = (NSDictionary *)obj;
            Person *person = [[Person alloc] init];
            person.name = pDict[@"name"];
            person.status = pDict[@"status"];
            person.imageURL = [NSURL URLWithString:pDict[@"photo"]];
            person.latitude = [pDict[@"latitude"] doubleValue];
            person.longitude = [pDict[@"longitude"] doubleValue];
        }
    }];
}
```



Subscripting

```
person.name = pDict[@"name"];
person.status = pDict[@"status"];
person.imageURL = [NSURL URLWithString:pDict[@"photo"]];
person.latitude = [pDict[@"latitude"] doubleValue];
person.longitude = [pDict[@"longitude"] doubleValue];
```

This is called “subscripting”: a faster way to access elements of arrays and dictionaries without the need to use `objectAtIndex:` or `objectForKey:`



Subscripting

Subscripting with NSDictionary

```
person.name = pDict[@"name"];
```



```
person.name = [pDict objectForKey:@"name"];
```

Subscripting with NSArray

```
id obj = locations[0];
```



```
id obj = [locations objectAtIndex:0];
```



Mobile Application Development



Lecture 8
Core Location and Map Kit



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).



Mobile Application Development



Lecture 9
Sensors and Multimedia



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

Lecture Summary

- Core Motion
- Working with Audio and Video
 - Media Player framework
 - System Sound Services
 - AVFoundation framework
- Camera and Photo Library





Core Motion

- Core Motion (`#import <CoreMotion/CoreMotion.h>`) provides a unified framework to receive motion data from device hardware and process that data
- The framework supports accessing both raw and processed accelerometer data using block-based interfaces
- Core Motion provides support for data coming from:
 - accelerometer (`CMAccelerometerData`)
 - gyroscope (`CMGyroData`)
 - magnetometer (`CMMagnetometerData`)

https://developer.apple.com/library/ios/documentation/CoreMotion/Reference/CoreMotion_Ref/index.html



CMMotionManager

- A **CMMotionManager** object is the gateway to the motion services provided by iOS
- Motion services provide accelerometer data, rotation-rate data, magnetometer data, and other device-motion data such as *attitude* (i.e., the orientation of a body relative to a given frame of reference)
- A **CMMotionManager** is created with alloc/init
- Only one **CMMotionManager** should be present in an app
- After a **CMMotionManager** has been created it can be configured to set some parameters, such as the update interval for each type of motion

https://developer.apple.com/library/ios/documentation/CoreMotion/Reference/CoreMotion_Ref/index.html



CMMotionManager

- After a **CMMotionManager** has been created and configured it can be asked to handle 4 types of motion:
 - raw accelerometer data
 - raw gyroscope data
 - raw magnetometer data
 - processed device-motion data (which includes accelerometer, rotation-rate, and attitude measurements)
- To receive motion data at specific intervals, the app calls a “start” method that takes an operation queue and a block handler of a specific type for processing those updates
- The motion data is passed into the block handler

```
startAccelerometerUpdatesToQueue: (NSOperationQueue *)  
    withHandler:^(CMAccelerometerData *accelerometerData, NSError *error)  
  
startGyroUpdatesToQueue: (NSOperationQueue *)  
    withHandler:^(CMGyroData *gyroData, NSError *error)  
  
startMagnetometerUpdatesToQueue: (NSOperationQueue *)  
    withHandler:^(CMMagnetometerData *magnetometerData, NSError *error)
```



CMMotionManager

- To stop receiving updates the appropriate stop methods of CMMotionManager must be called:
 - `stopAccelerometerUpdates`
 - `stopGyroUpdates`
 - `stopMagnetometerUpdates`
 - `stopDeviceMotionUpdates`



Core Motion Cookbook

- **Accelerometer:**
 - Set the `accelerometerUpdateInterval` property to specify an update interval
 - Call the `startAccelerometerUpdatesToQueue:withHandler:` method, passing in a block of type `CMAccelerometerHandler`
 - Accelerometer data is passed into the block as `CMAccelerometerData` objects
- **Gyroscope:**
 - Set the `gyroUpdateInterval` property to specify an update interval
 - Call the `startGyroUpdatesToQueue:withHandler:` method, passing in a block of type `CMGyroHandler`
 - Rotation-rate data is passed into the block as `CMGyroData` objects



Core Motion Cookbook

- **Magnetometer:**
 - Set the `magnetometerUpdateInterval` property to specify an update interval
 - Call the `startMagnetometerUpdatesToQueue:withHandler:` method, passing a block of type `CMMagnetometerHandler`
 - Magnetic-field data is passed into the block as `CMMagnetometerData` objects
- **Device motion:**
 - Set the `deviceMotionUpdateInterval` property to specify an update interval
 - Call the `startDeviceMotionUpdatesUsingReferenceFrame:` or `startDeviceMotionUpdatesUsingReferenceFrame:toQueue:withHandler:` or `startDeviceMotionUpdatesToQueue:withHandler:` method, passing in a block of type `CMDeviceMotionHandler`
 - Rotation-rate data is passed into the block as `CMDeviceMotion` objects



CMMotionManager example

```
CMMotionManager *manager = [[CMMotionManager alloc] init];  
  
[manager  
    startAccelerometerUpdatesToQueue:[[NSOperationQueue alloc] init]  
        withHandler:^(CMAccelerometerData *accelerometerData, NSError *error){  
  
    /* process data block */  
  
}];
```



Multimedia: Audio

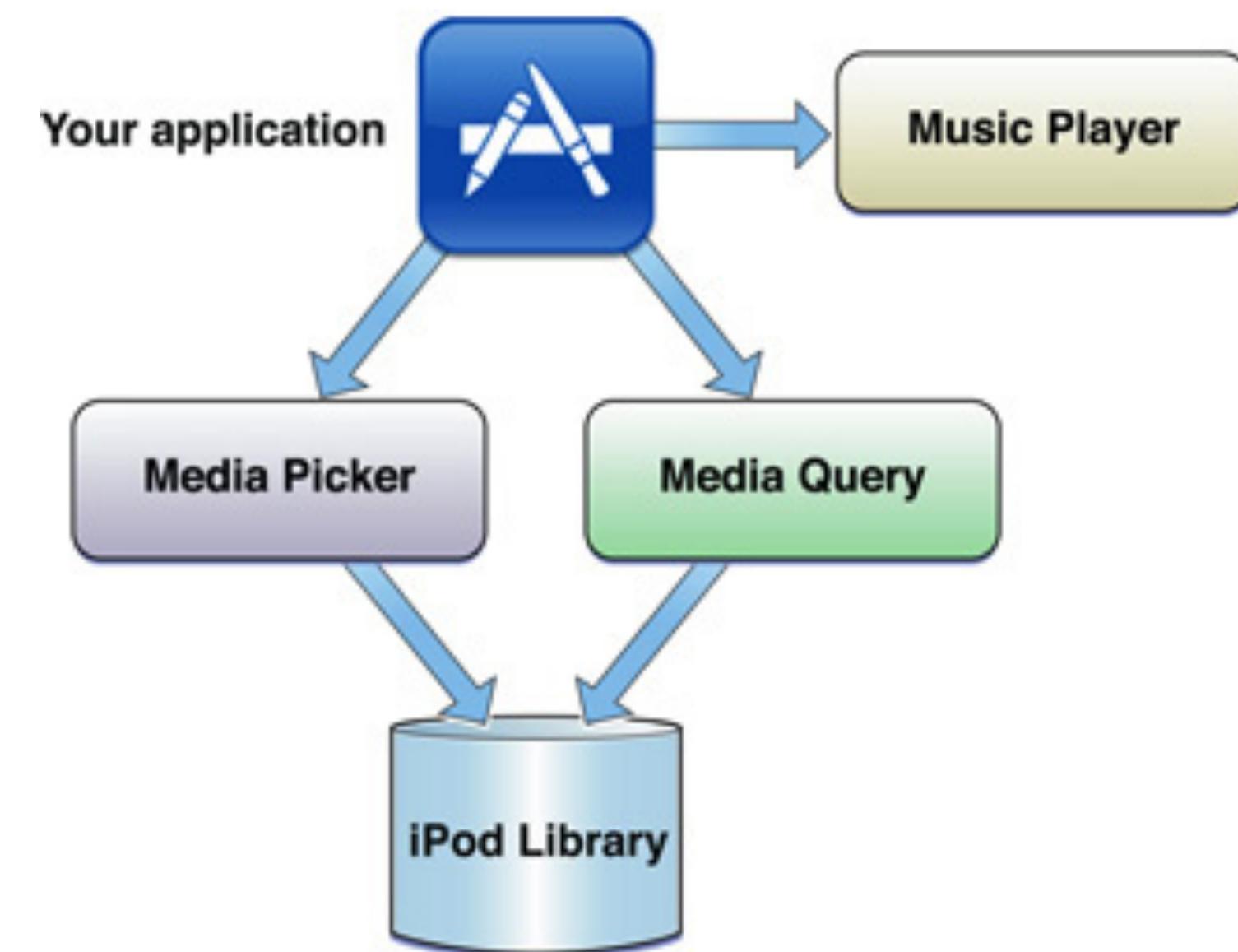
- iOS provides several frameworks to be used to deal with audio in applications at different levels
- Each framework has different features and the use of a framework over another depends on the type of operations that are needed inside an app

Framework	Usage
Media Player	play songs, audio books, or audio podcasts from a user's iPod library https://developer.apple.com/library/ios/documentation/MediaPlayer/Reference/MediaPlayer_Framework
AV Foundation	play and record audio using a simple Objective-C interface
Audio Toolbox	play audio with synchronization capabilities, access packets of incoming audio, parse audio streams, convert audio formats, and record audio with access to individual packets
Audio Unit	connect to and use audio processing plug-ins
OpenAL	provide positional audio playback in games and other applications. iOS supports OpenAL 1.1

<https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/MultimediaPG>

Accessing the iPod Library

- iPod library access provides mechanisms to play songs, audio books, and audio podcasts
- Basic playback and advanced searching and playback control
- There are two ways to get access to media items:
 1. the media picker is a view controller that behaves like the built-in iPod application's music selection interface
 2. the media query interface supports predicate-based specification of items from the device iPod library
- The retrieved media items are played using the music player `MPMusicPlayerController`



https://developer.apple.com/library/ios/documentation/Audio/Conceptual/iPodLibraryAccess_Guide



Accessing the iPod Library

- The Media Player framework must be linked to the project and imported in code using the import directive `#import <MediaPlayer/MediaPlayer.h>`
- There are two types of music player:
 - the *application music player* plays music locally within your app
 - the *iPod music player* employs the built-in iPod app
- An app music player can be instantiated with the following line of code

```
MPMusicPlayerController *player = [MPMusicPlayerController applicationMusicPlayer];
```

- After a music player has been created, a playback queue can be set using the method `setQueueWithQuery:` or `setQueueWithItemCollection:`
- The playback queue can then be played with the player's `play` method



Managing playback

- The `MPMusicPlayerController` class conforms to the `MPMediaPlayback` protocol
- This means that the player implements the following methods to change the playback state:
 - `play`
 - `pause`
 - `stop`
 - ... and other methods
- To control the playback of the queue, the following methods can also be used:
 - `skipToNextItem`
 - `skipToBeginning`
 - `skipToPreviousItem`



MPMediaItem

- A media item (**MPMediaItem**) represents a single media (a song or a video podcast) in the iPod library
- A media item has a unique identifier, which persists across application launches
- Media items can have a wide range of metadata associated, such as the song title, artist, album, ...
- Metadata are accessed using the **valueForProperty:** method
- Some valid properties (constants): **MPMediaItemPropertyTitle**, **MPMediaItemPropertyAlbumTitle**, **MPMediaItemPropertyArtist**, **MPMediaItemPropertyAlbumArtist**, **MPMediaItemPropertyGenre**
- Each property has a different return type: for instance **MPMediaItemPropertyTitle** returns an **NSString**, while **MPMediaItemPropertyArtwork** return an instance of **MPMediaItemArtwork**



MPMediaQuery

- Media items in the iPod library can be queried using the `MPMediaQuery` class
- A media item collection (`MPMediaItemCollection`) is an array of media items
- Collections from the device iPod library can be obtained by accessing a media query's `collections` property



MPMediaPickerController

- An **MPMediaPickerController** is a specialized view controller used to provide a graphical interface for selecting media items
- The **MPMediaPickerController** displays a table view with media items that can be selected
- Tapping on the done button will dismiss the media item picker
- The **MPMediaPickerController** has a delegate (**MPMediaPickerControllerDelegate**) which gets notified when the user selects a media item or cancels the picking, with the methods **mediaPicker:didPickMediaItems:** and **mediaPickerDidCancel:** respectively



Presenting View Controllers modally

- To present a view controller modally, it should have its `modalTransitionStyle` property set and then the current view controller should call `presentViewController:animated:completion:`

```
MPMediaPickerController *picker = [[MPMediaPickerController alloc] init];
picker.modalTransitionStyle = UIModalTransitionStyleCoverVertical;
[self presentViewController:picker animated:YES completion:nil];
```

- The media picker delegate should explicitly dismiss the picker in `mediaPickerControllerDidCancel:` with `dismissModalViewControllerAnimated:` (this is automatic when tapping the done button)



System Sound Services

- System Sound Services are used to play user-interface sound effects (such as button clicks), or to invoke vibration on devices that support it (`#import <AudioToolbox/AudioToolbox.h>`)
- Sound files must be:
 - less than 30 seconds long
 - in linear PCM or IMA4 (IMA/ADPCM) format
 - .caf, .aif, or .wav files
- First, first create a sound ID object with the `AudioServicesCreateSystemSoundID()` function
- Then, play the sound with the `AudioServicesPlaySystemSound()` function
- To trigger a vibration, just use the following line of code:

`AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);`

<https://developer.apple.com/library/ios/documentation/AudioToolbox/Reference/SystemSoundServicesReference>



AVAudioPlayer

- The AVAudioPlayer class (`#import <AVFoundation/AVFoundation.h>`) provides a simple Objective-C interface for playing sounds
- Apple recommends to use this class for playback of audio that does not require stereo positioning or precise synchronization, and audio was not captured from a network stream
- AVAudioPlayer can:
 - Play sounds of any duration
 - Play sounds from files or memory buffers
 - Loop sounds
 - Play multiple sounds simultaneously (although not with precise synchronization)
 - Control relative playback level for each sound you are playing
 - Seek to a particular point in a sound file, which supports application features such as fast forward and rewind
 - Obtain audio power data that you can use for audio level metering
- The AVAudioPlayer class has a delegate (AVAudioPlayerDelegate) which gets notified of audio interruptions, audio decoding errors, and completion of playback



AVAudioPlayer example

```
/* get the URL of an audio file in the bundle */
NSString *soundFilePath = [[NSBundle mainBundle] pathForResource: @"sound" ofType: @"wav"];
NSURL *fileURL = [[NSURL alloc] initFileURLWithPath: soundFilePath];

/* create an AVAudioPlayer for the given audio file */
AVAudioPlayer *player = [[AVAudioPlayer alloc] initWithContentsOfURL:fileURL error:nil];

/* prepare the audio player for playback by preloading its buffers */
[player prepareToPlay];

/* set the delegate for the player */
[player setDelegate:self];

/* start the playback */
[player play];
```



AVAudioRecorder

- The AVAudioRecorder class (`#import <AVFoundation/AVFoundation.h>`) provides a simple Objective-C interface for recording sounds
- AVAudioRecorder can:
 - Record until the user stops the recording
 - Record for a specified duration
 - Pause and resume a recording
 - Obtain input audio-level data that you can use to provide level metering
- The AVAudioRecorder class has a delegate (AVAudioRecorderDelegate) which gets notified of audio interruptions, audio decoding errors, and completion of recording
- The usage of an audio recorder is very similar to that of an audio player, but the recording settings, such as the bitrate and the number of channels, should be configured
- An audio recorder are to be used inside an AVAudioSession, which is used to set the audio context for the app



AVAudioRecorder example

```
/* set the AVAudioSession parameters */
[[AVAudioSession sharedInstance] setCategory:AVAudioSessionCategoryRecord error:nil];
[[AVAudioSession sharedInstance] setMode:AVAudioSessionModeVideoRecording error:nil];
[[AVAudioSession sharedInstance] setActive:YES error:nil];

/* get the URL of an audio file in the bundle */
NSString *soundFilePath = [[NSBundle mainBundle] pathForResource: @"sound" ofType: @"wav"];
NSURL *fileURL = [[NSURL alloc] initFileURLWithPath: soundFilePath];

/* get the URL for the recorded audio file */
NSURL *url = ...;

/* set the audio recorder parameters (format, quality, bitrate, number of channels, sample rate) */
NSDictionary *settings = [NSDictionary dictionaryWithObjectsAndKeys:
                           @(kAudioFormatMPEG4AAC), AVFormatIDKey,
                           @(AVAudioQualityMax), AVEncoderAudioQualityKey,
                           @(16), AVEncoderBitRateKey,
                           @(1), AVNumberOfChannelsKey,
                           @(44100), AVSampleRateKey,
                           nil];

/* create the audio recorder */
AVAudioRecorder *recorder = [[AVAudioRecorder alloc] initWithURL:url settings:settings error:nil];

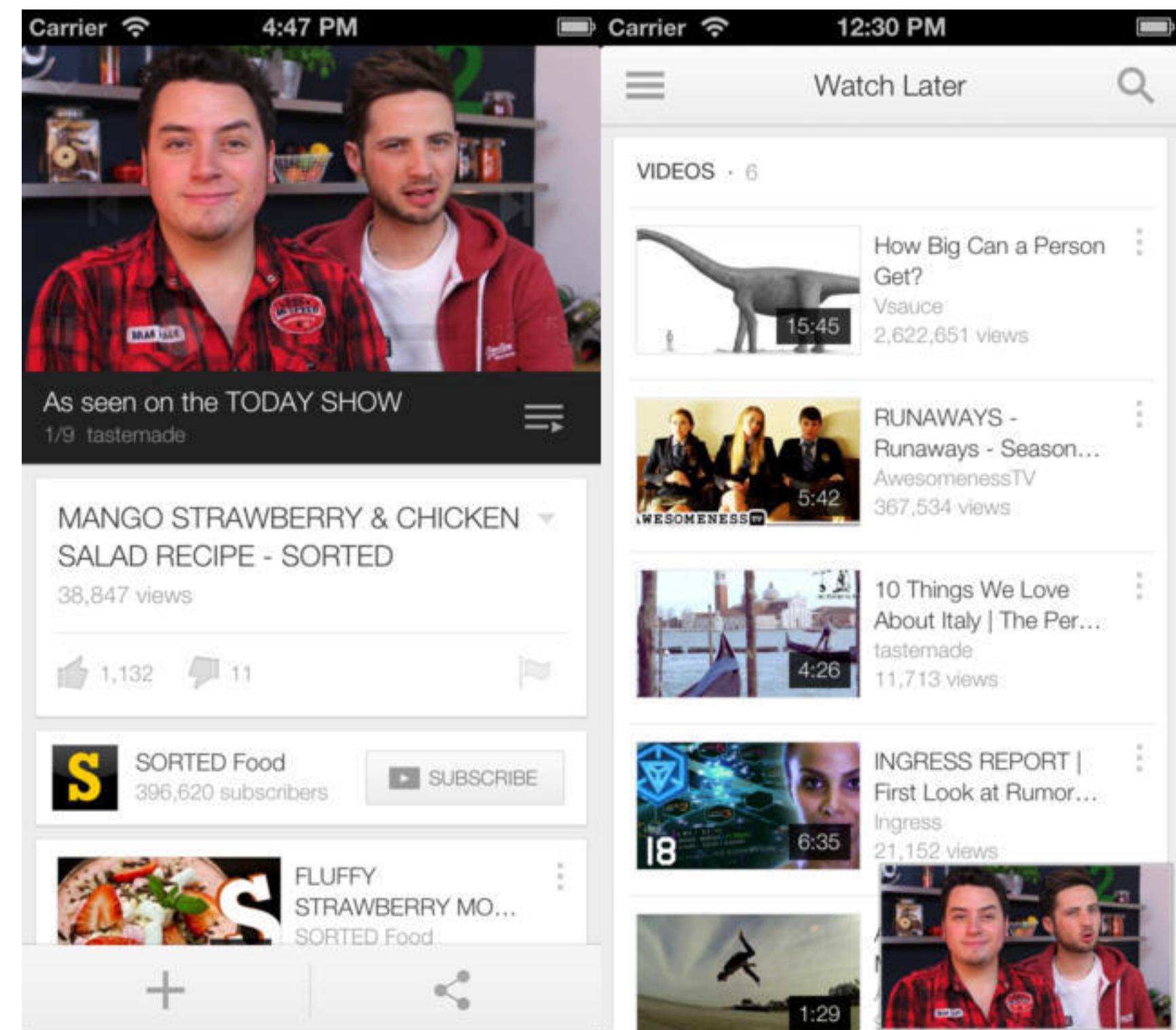
/* start recording */
[recorder record]; /* or use recordForDuration: */
```

Video Playback with MPMoviePlayerController

- An **MPMoviePlayerController** manages the playback of a movie from a file or a network stream
- Playback occurs in a view owned by the movie player and takes place either fullscreen or inline
- **MPMoviePlayerController** supports wireless movie playback to AirPlay devices like the Apple TV



- An **MPMoviePlayerController** object is initialized with the URL of the movie to be played
- After it has been created, the view of the **MPMoviePlayerController** (accessible via its `view` property) can be added as a subview of the current view



Video Playback with MPMoviePlayerController

- **MPMoviePlayerController** has many properties that can be used to affect its behavior:
 - `allowsAirPlay` specifies whether the movie player allows AirPlay movie playback
 - `fullscreen` indicates whether the movie player is in full-screen mode (read-only); the value can be changed with the method `setFullscreen:animated:`
 - `controlStyle` specifies the style of the playback controls (no controls, embedded, fullscreen)
 - `initialPlaybackTime` is the time of the video, in seconds, when playback should start
 - `endPlaybackTime` is the time of the video, in seconds, when playback should stop
 - ...
- **MPMoviePlayerController** also generates notifications related to the state of movie playback:
 - `MPMoviePlayerPlaybackStateDidChangeNotification`
 - `MPMoviePlayerContentPreloadDidFinishNotification`
 - `MPMoviePlayerDidEnterFullscreenNotification/MPMoviePlayerDidEnterFullscreenNotification`
 - ...
- To receive the notifications, it is necessary to register for the appropriate notification



MPMoviePlayerController example

```
/* create a player to play a video at the given URL */
MPMoviePlayerController *player = [[MPMoviePlayerController alloc] initWithContentURL:url];

/* set the player's view frame */
[player.view setFrame:self.view.bounds];

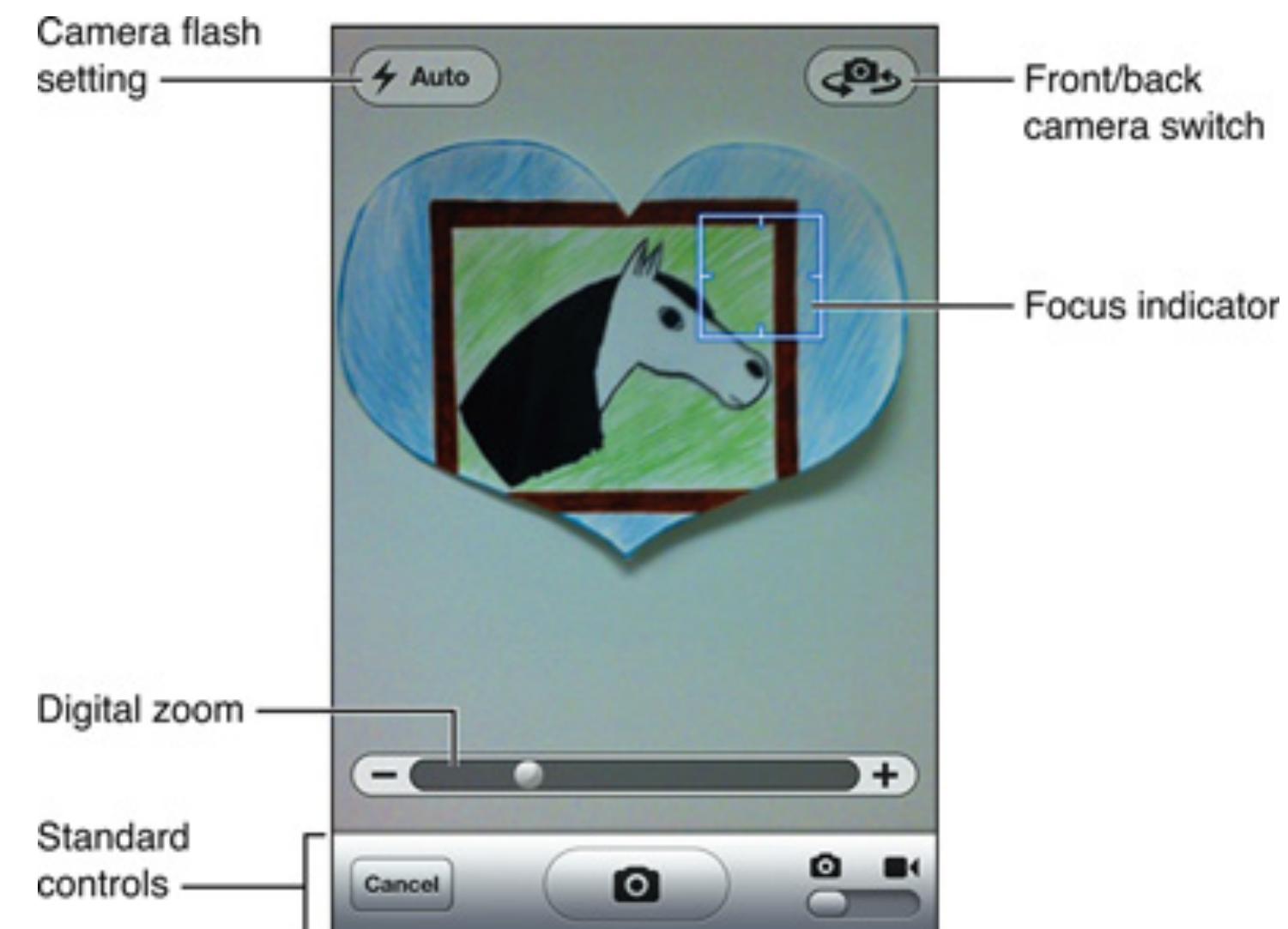
/* add the player's view to the current view controller's view */
[self.view addSubview:player.view];

/* configure player's settings */
player.allowsAirPlay = YES;
player.initialPlaybackTime = 10;
player.endPlaybackTime = 20;

/* start playback */
[player play];
```

Taking pictures and videos

- Taking pictures and videos is a (pretty standard) process that involves three steps:
 - creation and (modal) presentation of the camera interface (`UIImagePickerController`)
 - the user takes a pictures/video or cancels
 - the `UIImagePickerController` notifies its delegate about the result of the operation
- A `UIImagePickerController` displays a camera interface with a number of controls:





Taking pictures and videos

- Before using `UIImagePickerController`, the following conditions are required:
 - the device must have a camera (if using the camera is essential for the app, it must be added to the required device capabilities in the Info.plist of the project; if using the camera is optional, the app should have a different flow if a camera is not available)
 - the device's camera must be available for use with the given source type (photo camera, photo library, camera roll), by checking the return value of the method `isSourceTypeAvailable`:
 - a delegate object (conforming to `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate` protocol) to respond to the user's interaction with the image picker controller must have been implemented
- To specify whether the user can take still images, movies, or both, the `mediaTypes` property must be set; `mediaTypes` is a non-empty array whose elements can be the constant values `kUTTypeImage` and `kUTTypeMovie` (link the Mobile Core Services framework to the project and `#import <MobileCoreServices/MobileCoreServices.h>`)



Showing the image picker controller

```
/* assume the view controller conforms to UIImagePickerControllerDelegate and UINavigationControllerDelegate */
- (BOOL)startCameraController{

    /* first check whether the camera is available for use */
    if ([[UIImagePickerController isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera] == NO])
        return NO;

    /* create an image picker controller and set its sourceType property */
    UIImagePickerController *camera = [[UIImagePickerController alloc] init];
    camera.sourceType = UIImagePickerControllerSourceTypeCamera;

    /* Displays a control that allows the user to choose picture or movie capture, if both are available */
    camera.mediaTypes = [UIImagePickerController
                           availableMediaTypesForSourceType:UIImagePickerControllerSourceTypeCamera];

    /* hide the controls for moving and scaling pictures */
    camera.allowsEditing = NO;
    /* set the delegate for the image picker controller */
    camera.delegate = self;

    /* present the image picker controller modally */
    camera.modalTransitionStyle = UIModalTransitionStyleCoverVertical;
    [self presentViewController:camera animated:YES completion:nil];

    return YES;
}
```



Implementing delegate methods for the image picker

```
/* Responding to the user Cancel (dismiss the image picker controller) */
- (void) imagePickerControllerDidCancel:(UIImagePickerController *)picker{
    /* Dismiss the image picker controller */
    [picker.parentViewController dismissViewControllerAnimated:YES completion:nil];
}

/* Responding to the user user accepting a newly-captured picture or movie */
/* The info argument is a dictionary containing the original image and the edited image, if an
image was picked; or a filesystem URL for the movie, if a movie was picked */
- (void) imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info{

    NSString *mediaType = [info objectForKey:UIImagePickerControllerMediaType];

    /* Handle still image capture */
    if (CFStringCompare((CFStringRef) mediaType, kUTTypeImage, 0) == kCFCompareEqualTo){
        /* ... */
    }

    /* Handle video capture */
    if (CFStringCompare((CFStringRef) mediaType, kUTTypeMovie, 0) == kCFCompareEqualTo){
        /* ... */
    }

    /* Dismiss the image picker controller */
    [picker.parentViewController dismissViewControllerAnimated:YES completion:nil];
}
```



Picking pictures and videos from the Photo Library

- The `UIImagePickerController` can be used also to pick items from the device's photo library
- The steps are similar to those for capturing media, with some slight differences:
 - use the Camera Roll album or Saved Photos album, or the entire photo library as the media source
 - the user picks previously-saved media instead of capturing new media
- To configure the picker for browsing saved media the `sourceType` property must be set to:
 - `UIImagePickerControllerSourceTypePhotoLibrary` (all photo albums on the device, including the Camera Roll album)
 - `UIImagePickerControllerSourceTypeSavedPhotosAlbum` (access to the Camera Roll album only)



Mobile Application Development



Lecture 9
Sensors and Multimedia



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).



Mobile Application Development



Lecture 10
Auto Layout



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).



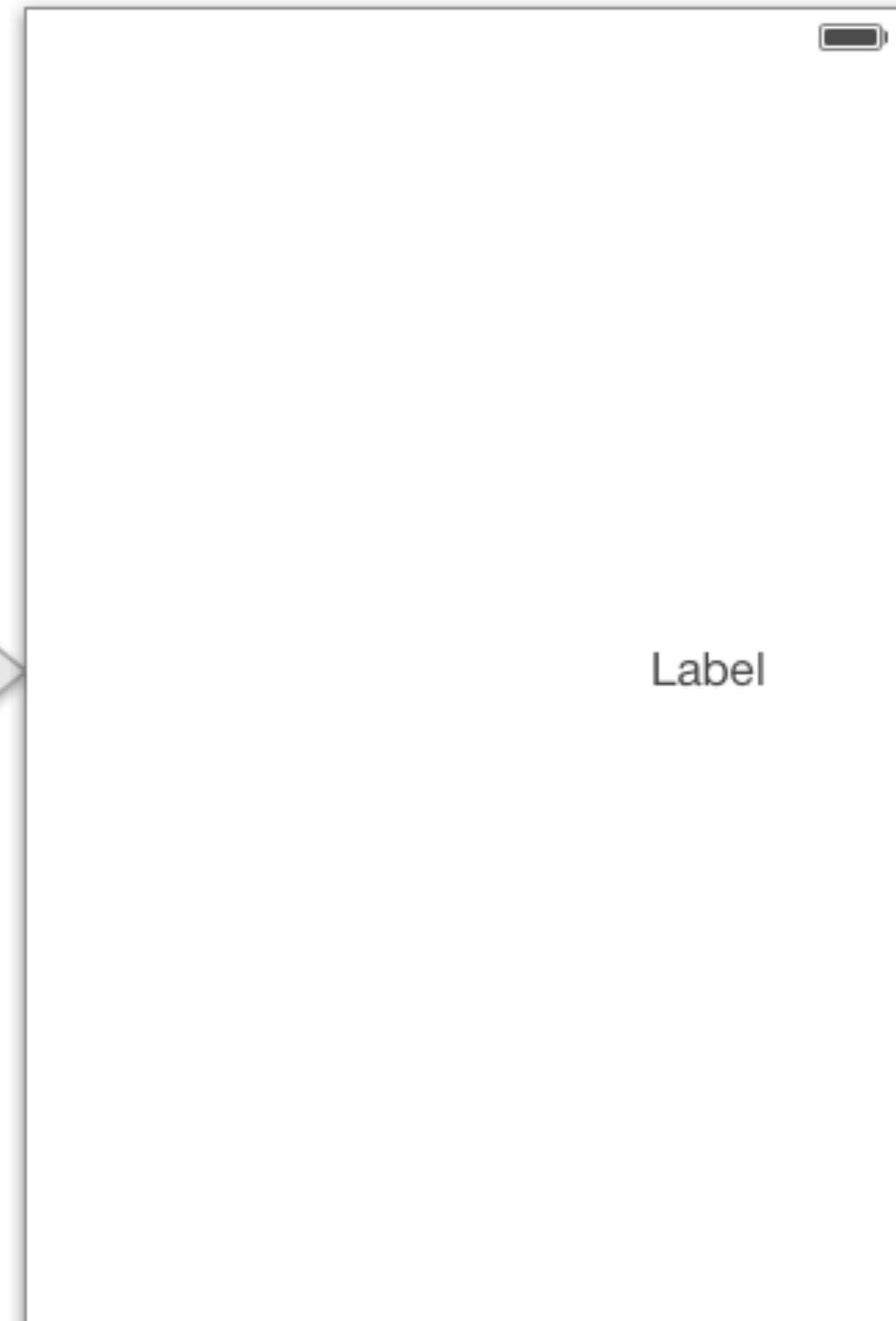
Lecture Summary

- Auto Layout



Hard-coded Layout

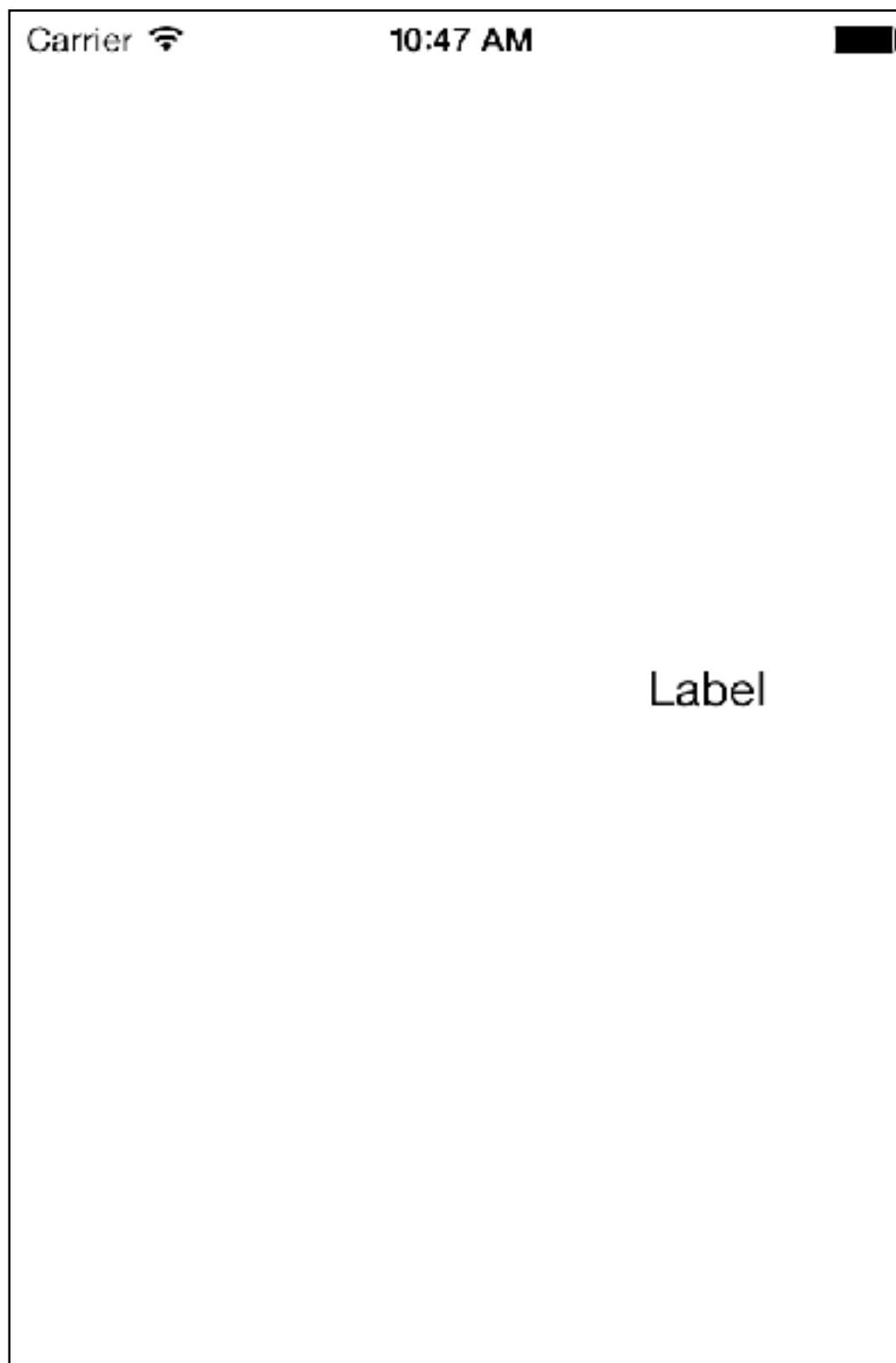
- In storyboard, we can add subviews to a view controller's view by dragging it a view object from the object palette
- Let's place it in a "random" place - also using the blue guidelines
- The label has a fixed (static) frame starting at point (226,229)



<https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG>

Hard-coded Layout

- Everything ok, the label is just where it was placed



Portrait

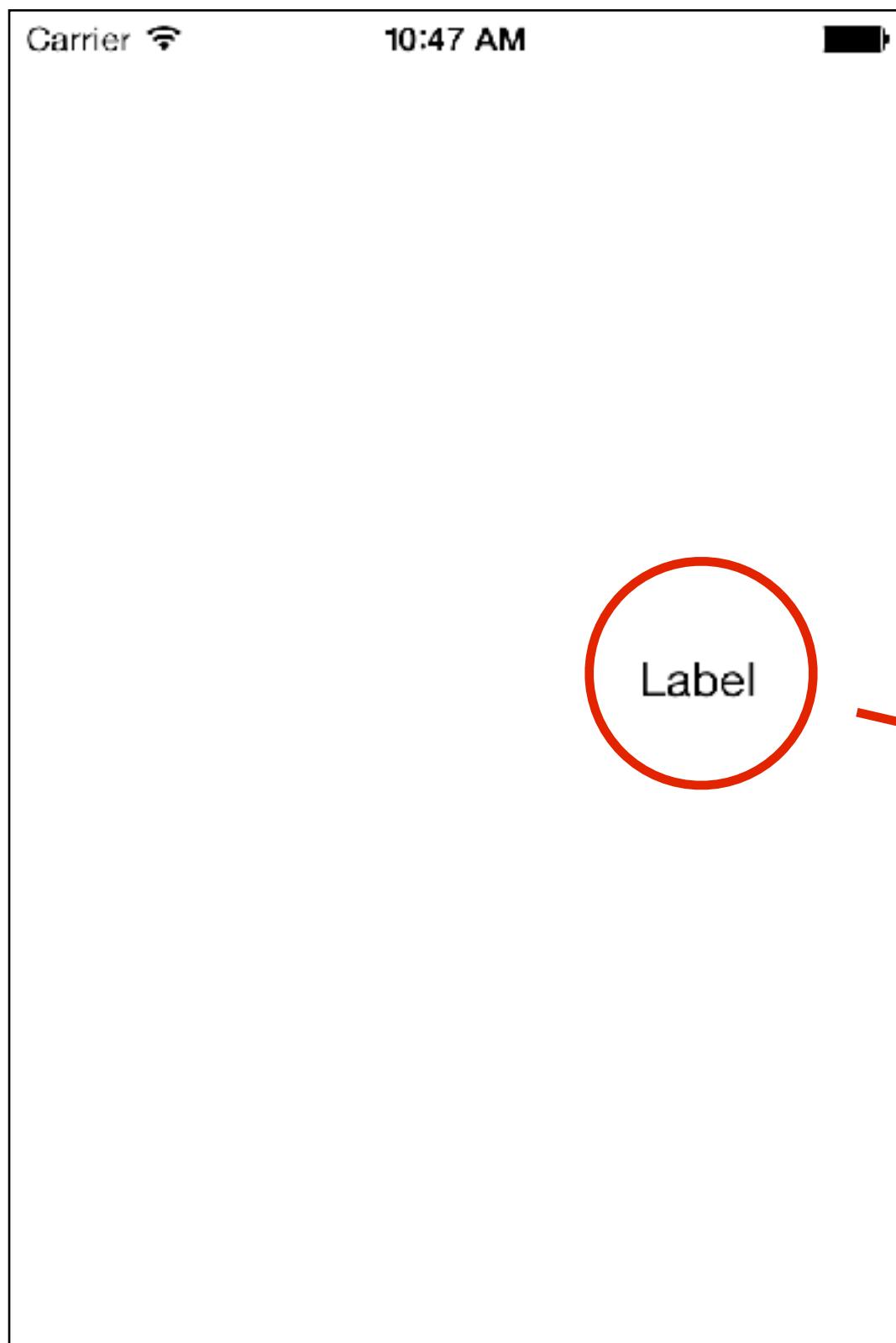
Hard-coded Layout

- What happens when the device is rotated?

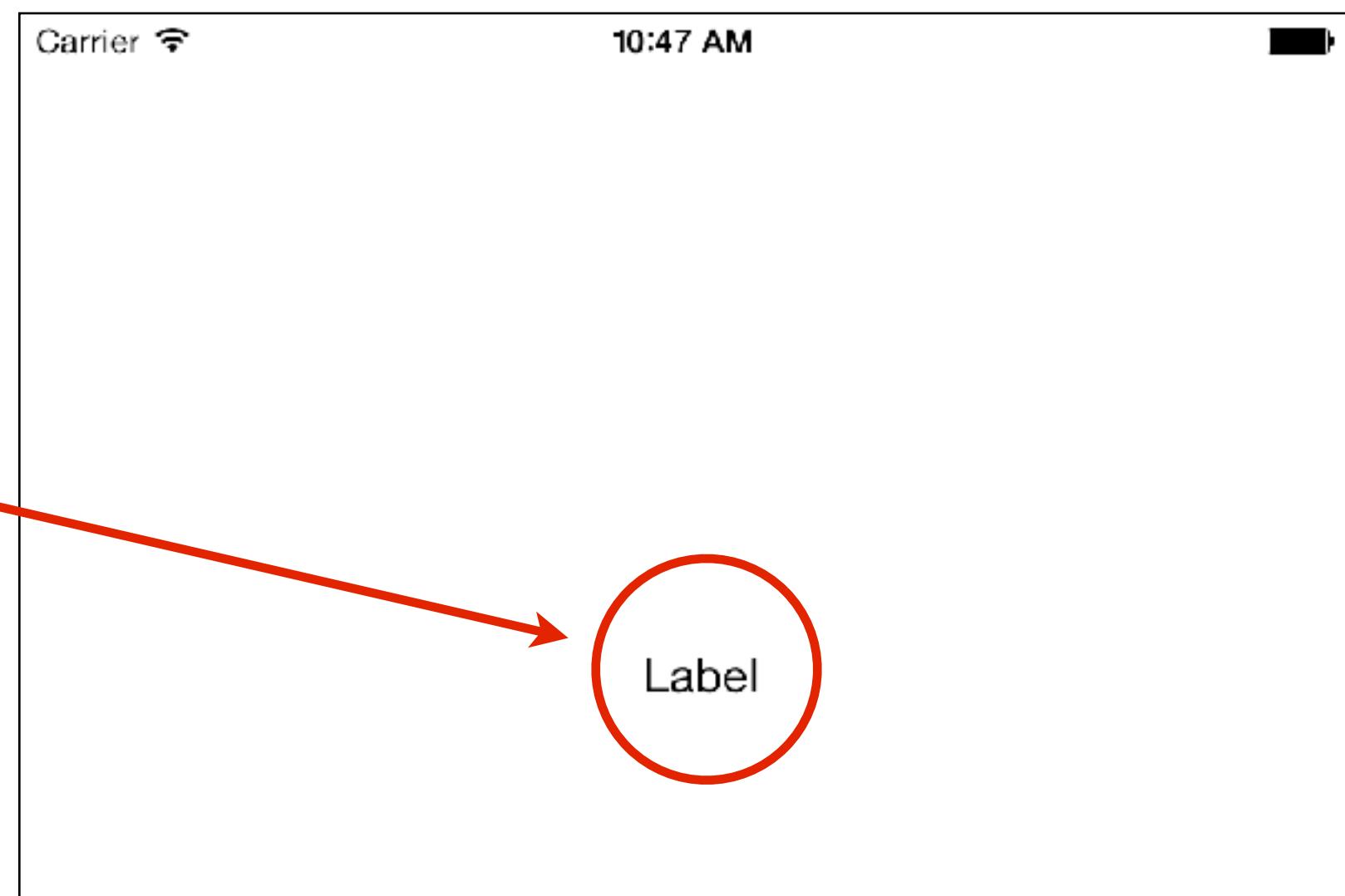


Hard-coded Layout

- The label has a fixed (static) frame starting at point (226,229)
- When switching to landscape mode, the label maintains its original frame
- It would be best to have the UI adapt to the orientation automatically (e.g. stay at the center vertically)

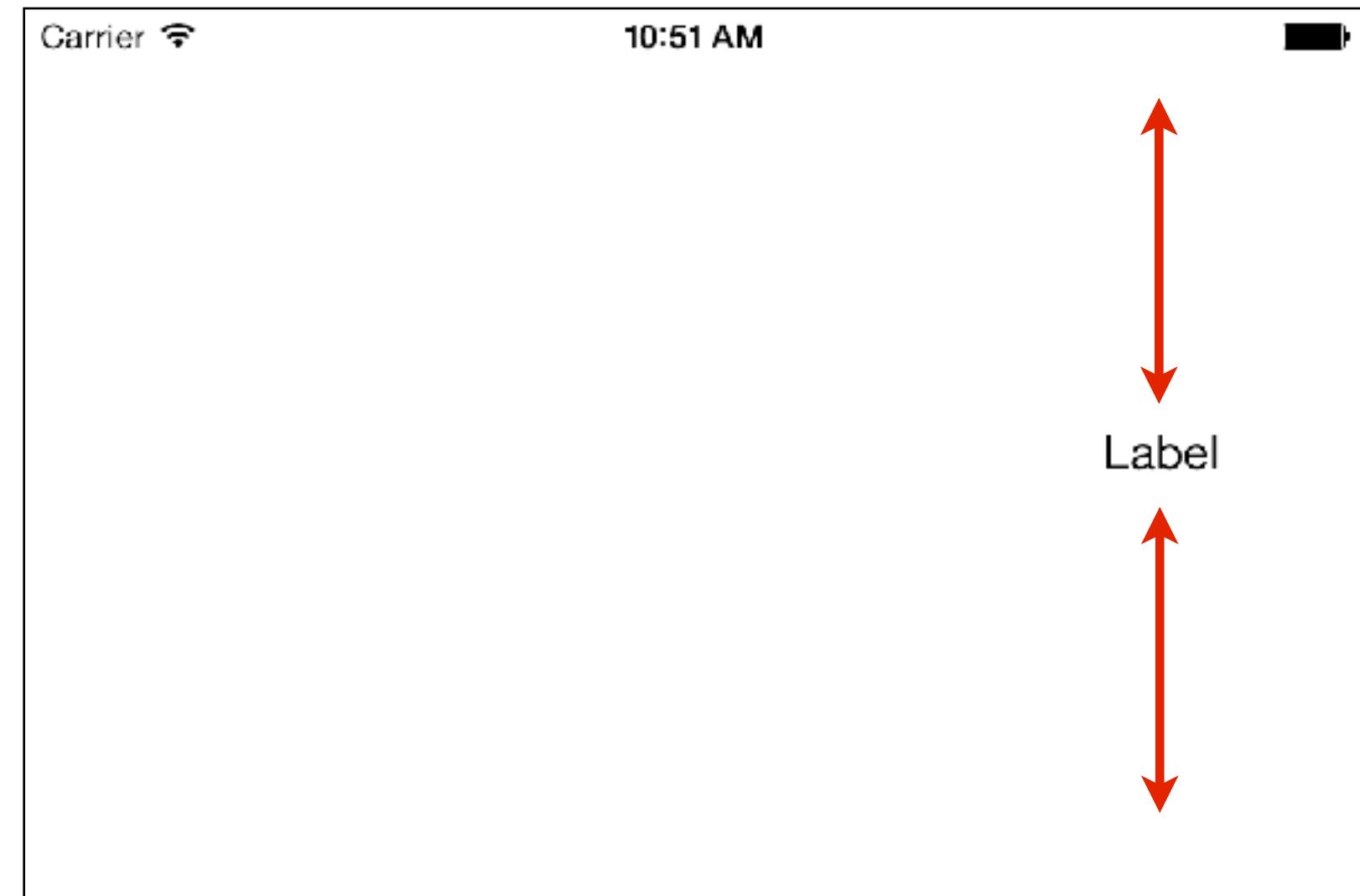
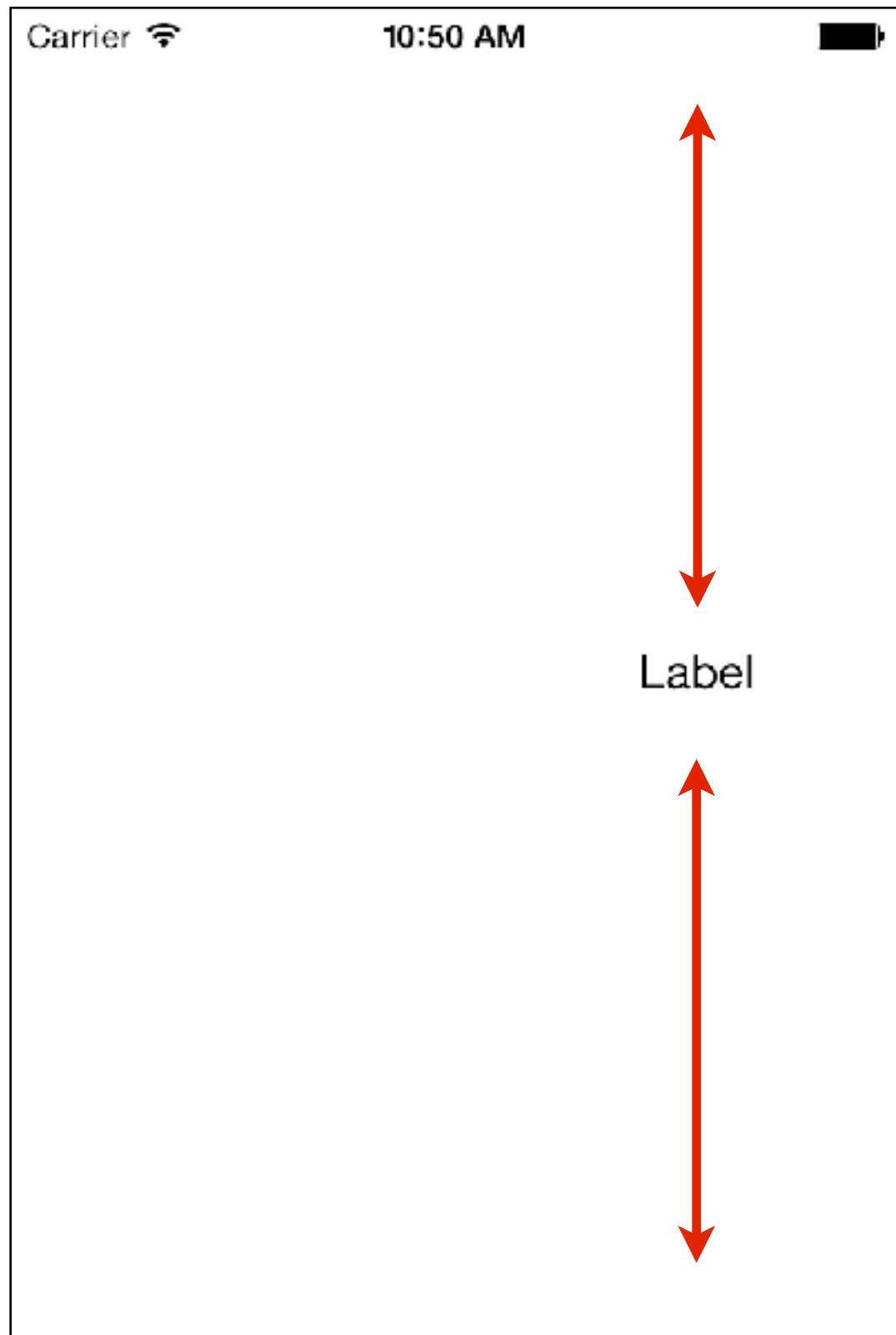


Portrait



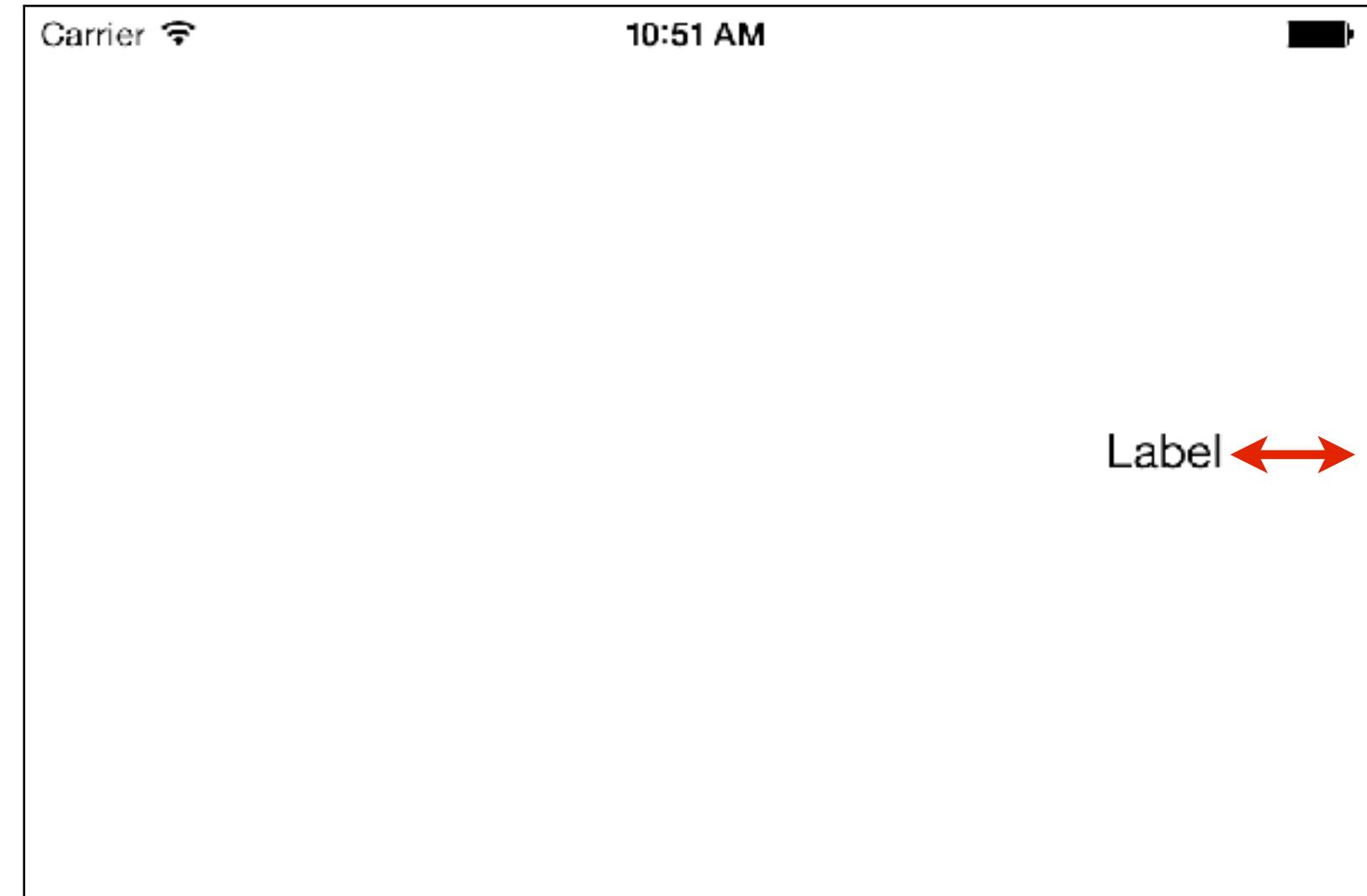
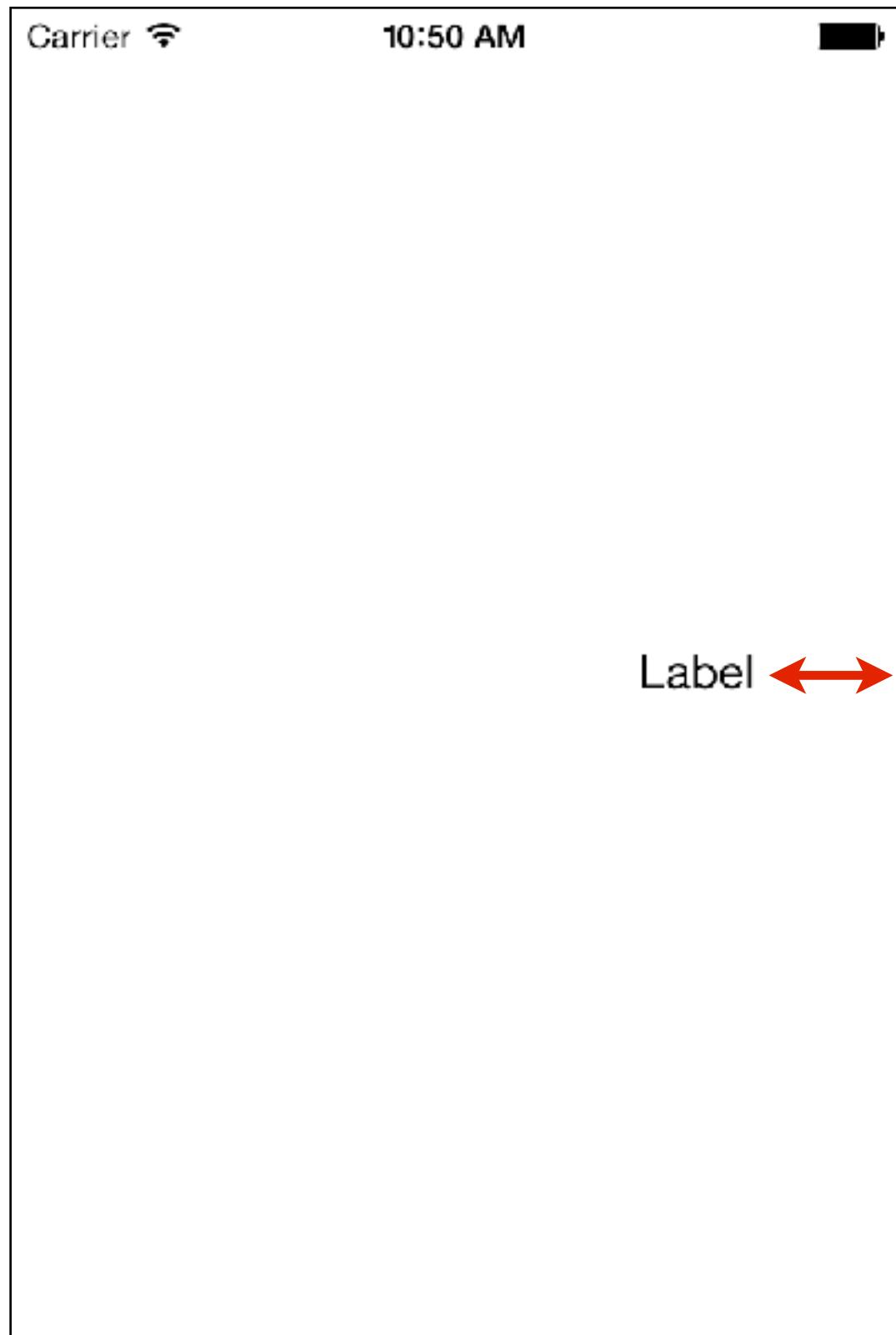
Landscape

Auto Layout



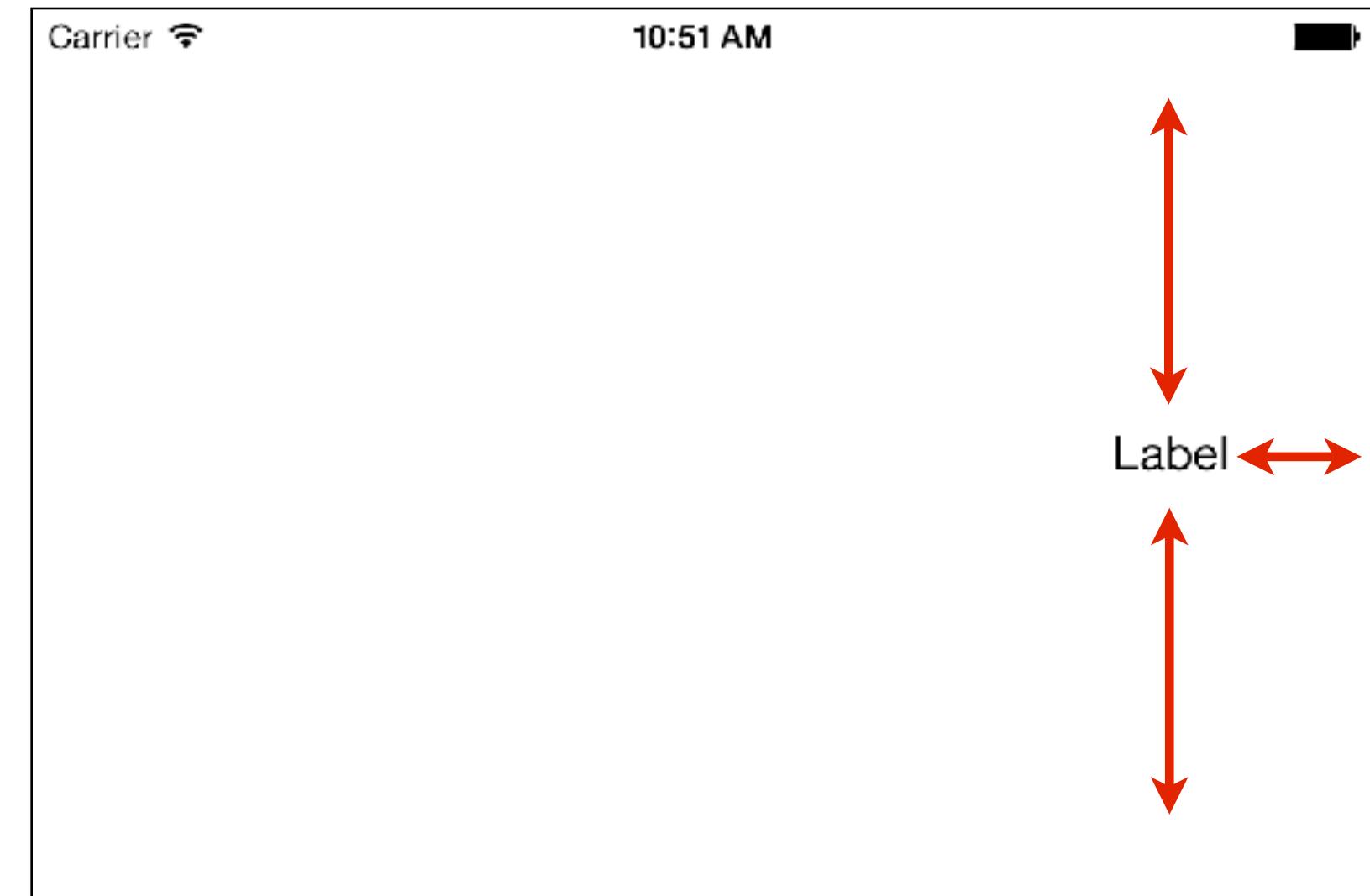
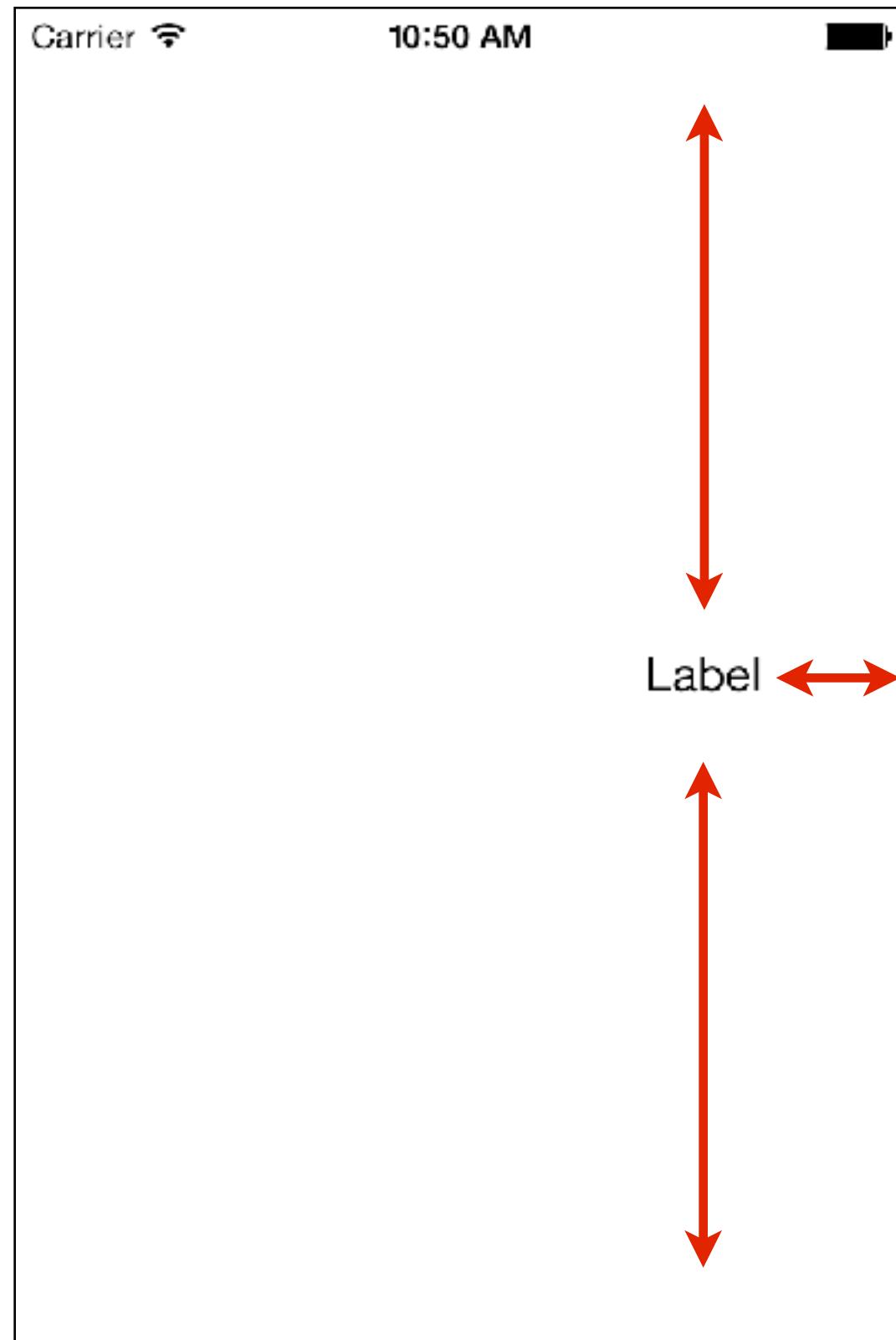
- The label is vertically centered in the container view

Auto Layout



- The label has a fixed trailing space (to the right edge of the container)

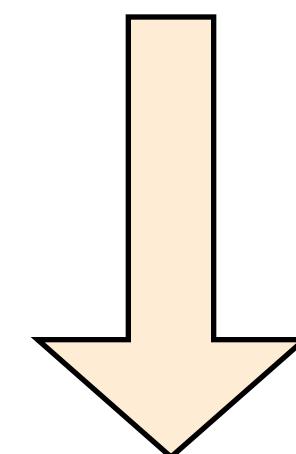
Auto Layout



- The label is vertically centered in the container view
- The label has a fixed trailing space (to the right edge of the container)

Auto Layout

- The label is vertically centered in the container view
- The label has a fixed trailing space (to the right edge of the container)



```
Control.centerY =Superview.centerY  
  
Control.right =Superview.right - <padding>
```

- Auto Layout takes these human-readable rules defined and transforms them into **constraints**

Auto Layout

- Auto Layout is a constraint-based, descriptive layout system
- Auto Layout makes it easy to solve many complex layout problems automatically, without the need for manual view manipulation
- With Auto Layout, the layout with constraints is described, and frames are calculated automatically
- Constraints are defined as relationships between the view objects (allows to describe very complex relationships)
- Auto Layout keeps software less bug-prone: it avoids bugs that might be introduced if manual view manipulation were adopted (no code required!)
- Auto Layout makes it easy to deal with:
 - orientation (portrait vs. landscape orientation)
 - different screen sizes (e.g., iPhone 4 (3.5" screen) vs. iPhone 5 (4" screen))



Constraints

- A constraint is a sort of mathematical representation of a human-expressable statement
- *The left edge should be 20 points from the left edge of its containing view* translates to
`button.left = (container.left + 20)`
- The constraint can be therefore represented by a formula: $y = m*x + b$, where
 - y and x are attributes of views
 - m and b are floating point values
- An attribute is one of *left*, *right*, *top*, *bottom*, *leading*, *trailing*, *width*, *height*, *centerX*, *centerY*, and *baseline*

Using Auto Layout

- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout



Using Auto Layout

- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout



Switch between iPhone 3.5" and iPhone 4" screen sizes

Using Auto Layout

- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout



Set alignment constraints: create alignment constraints, such as centering a view in its container, or aligning the left edges of two views

Using Auto Layout

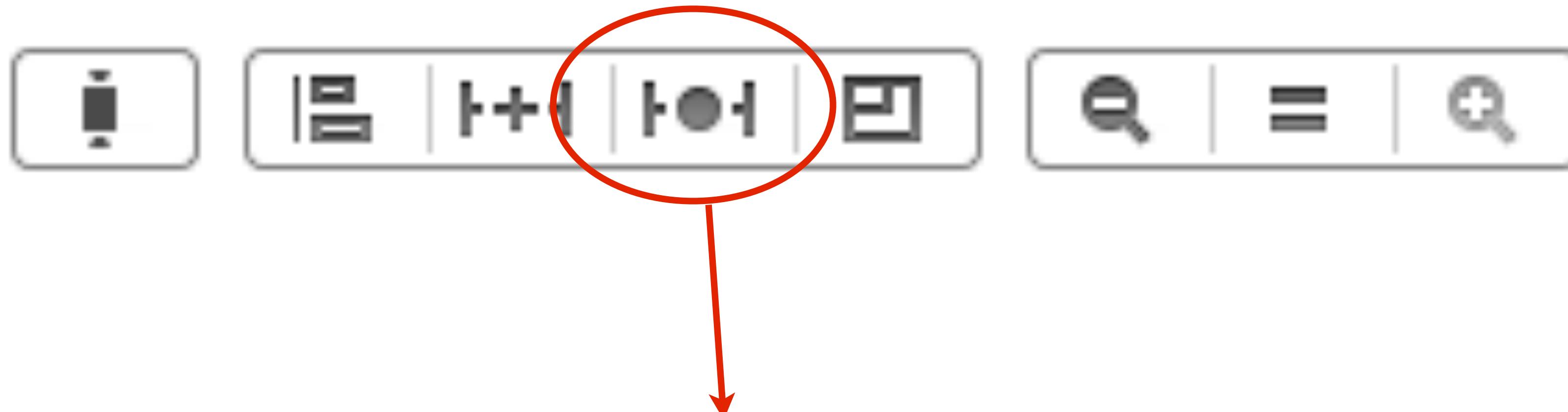
- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout



Set pin constraints: create spacing constraints, such as defining the height of a view, or specifying its horizontal distance from another view

Using Auto Layout

- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout



Resolve Auto Layout issues: resolve layout issues by adding or resetting constraints based on suggestions

Using Auto Layout

- Auto Layout is managed visually in Xcode
- In storyboard, in the lower-right corner, are the controls for using Auto Layout



Resizing: specify how resizing affects constraints

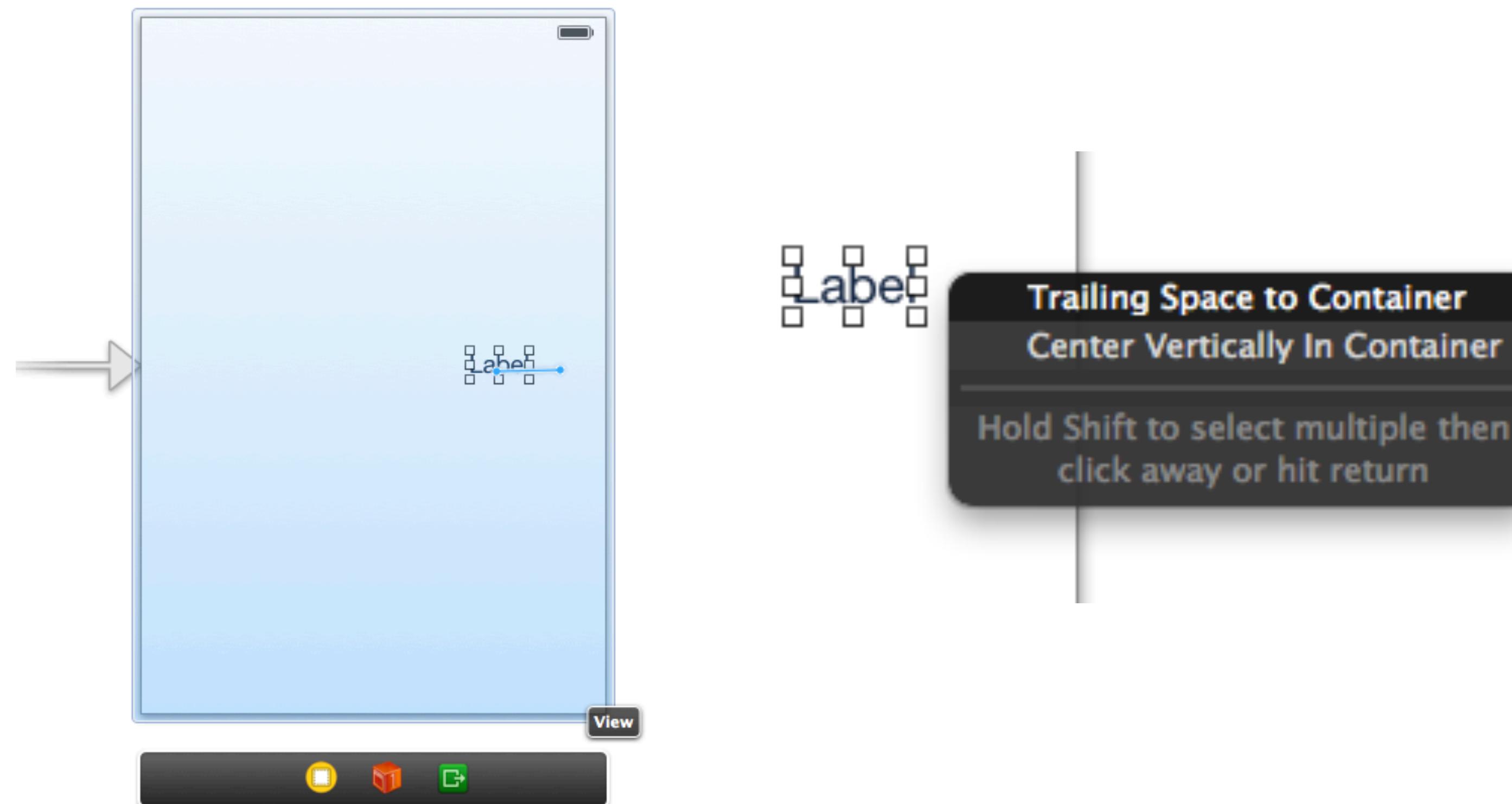


Adding constraints

- Constraints can be added in 3 ways:
 - Control-Drag
 - Align and Pin Menus
 - Adding Missing or Suggested Constraints

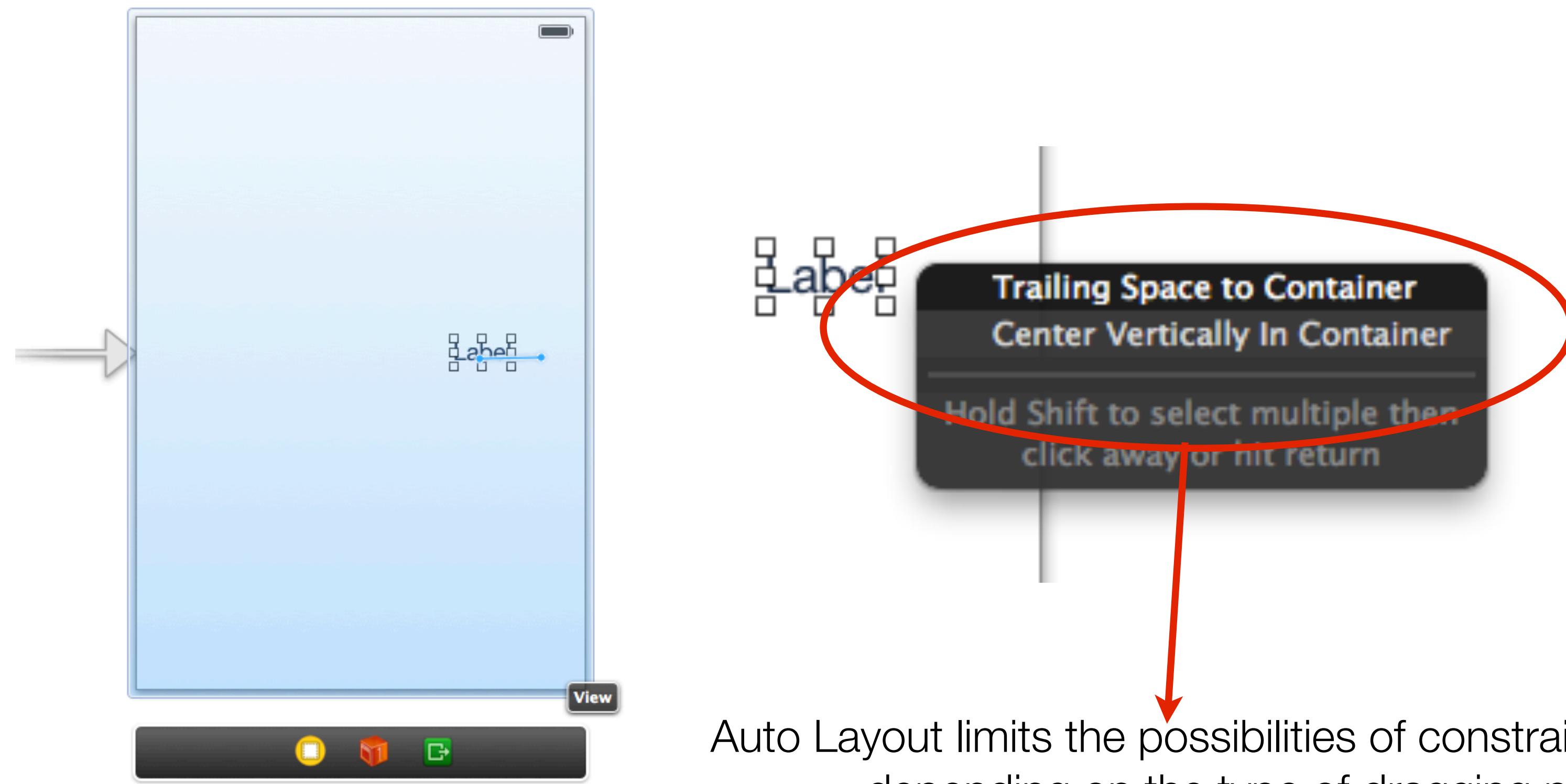
Adding Constraints with Control-Drag

- Fastest way to add a single, specific constraint
- Hold down the Control key and dragging from a view on the canvas (or another view, for relative positioning)



Adding Constraints with Control-Drag

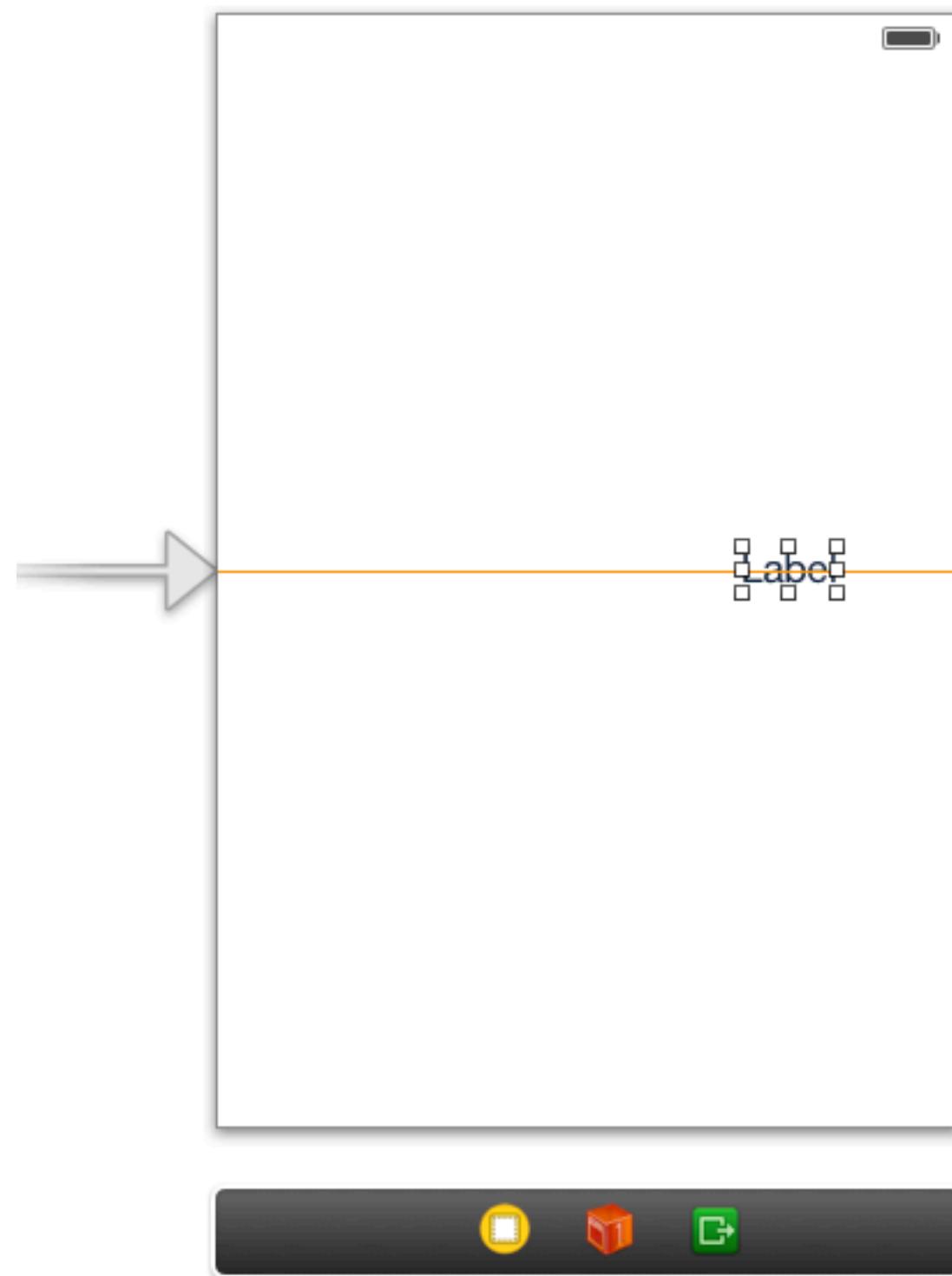
- Fastest way to add a single, specific constraint
- Hold down the Control key and dragging from a view on the canvas (or another view, for relative positioning)



Auto Layout limits the possibilities of constraints appropriately, depending on the type of dragging performed

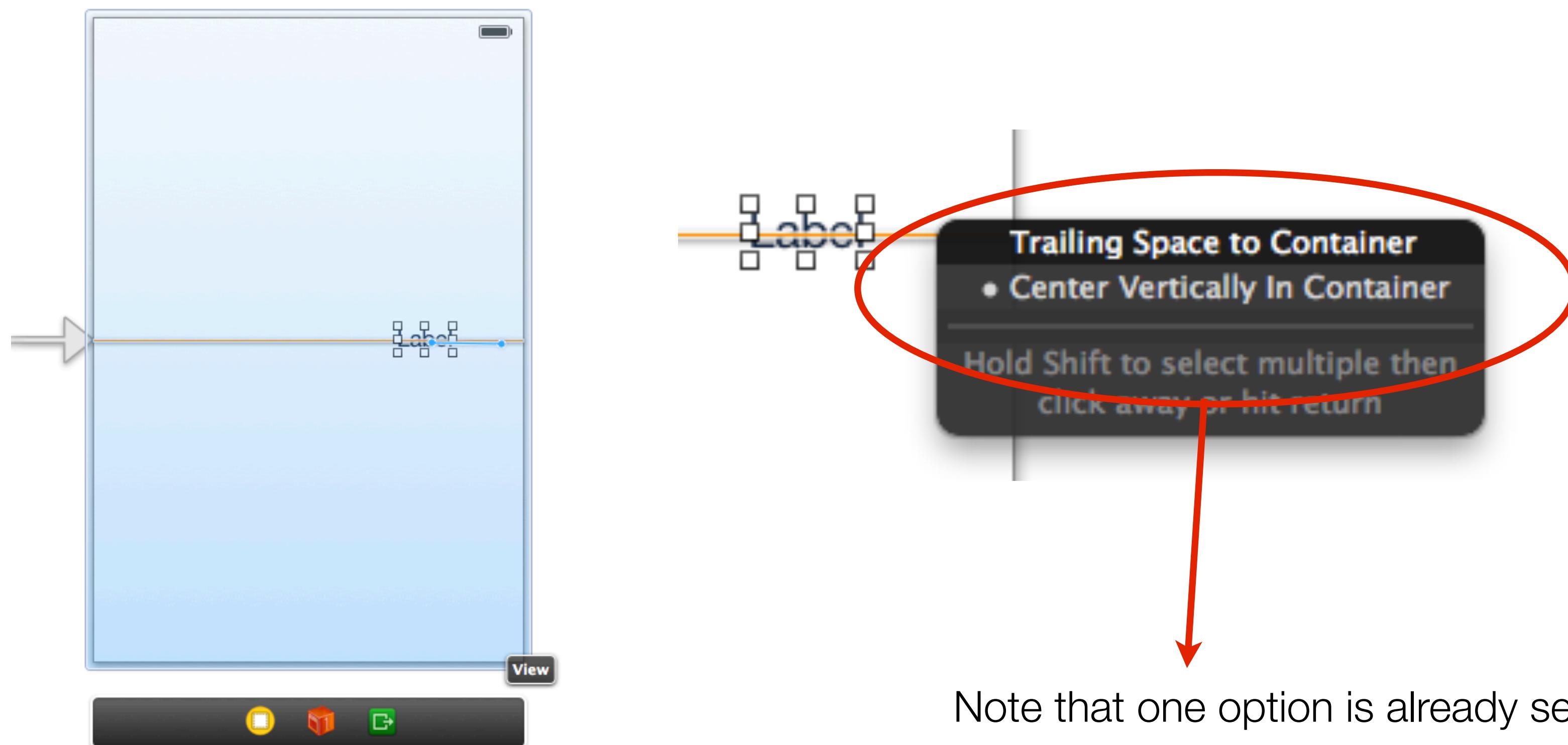
Adding Constraints with Control-Drag

- Select **Center Vertically in Container**
- The constrained is added
- The yellow line is a warning: some constraint is missing (what about the horizontal position?)



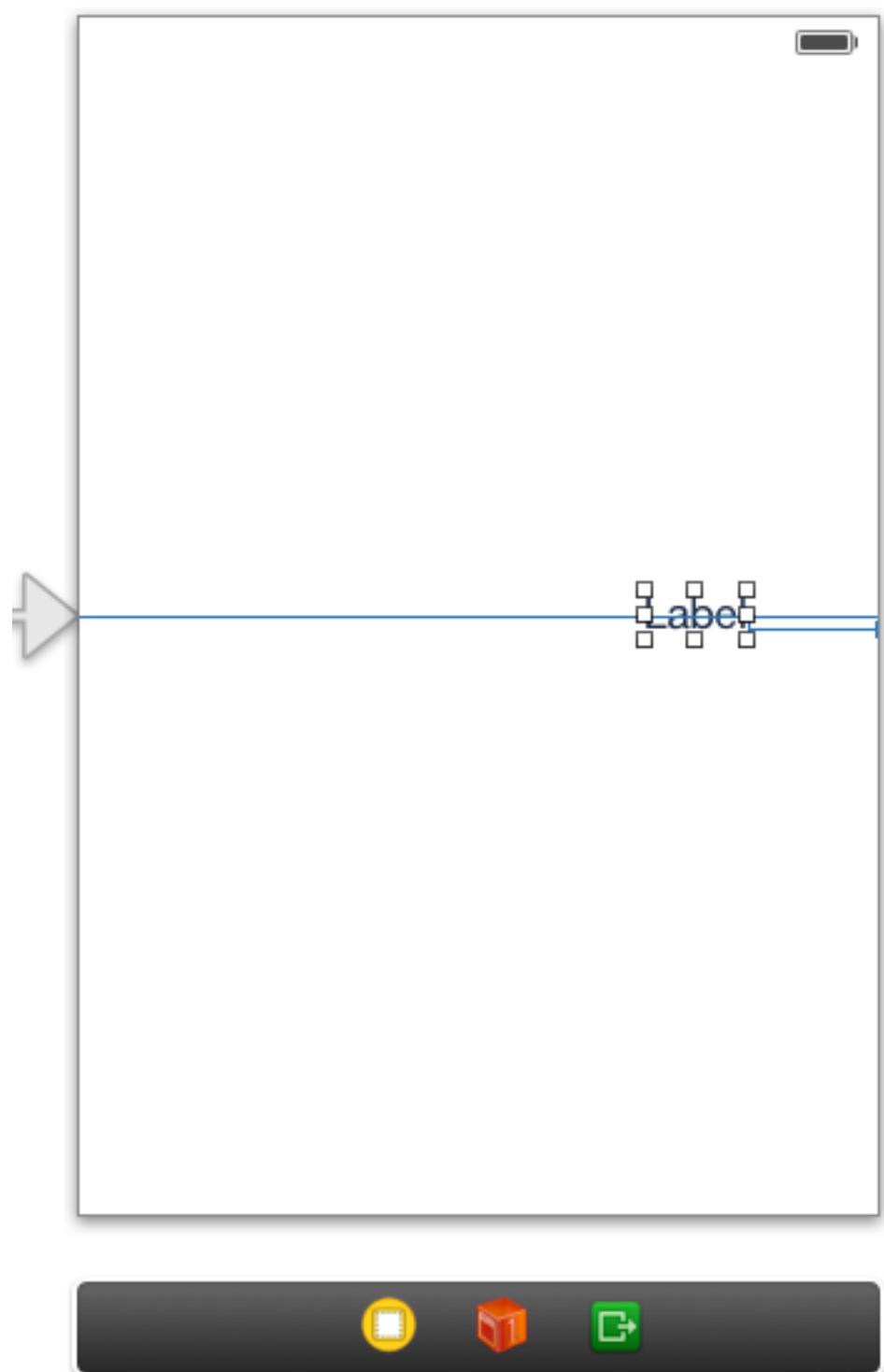
Adding Constraints with Control-Drag

- Repeat the Ctrl-drag procedure
- Select **Trailing Space to Container**



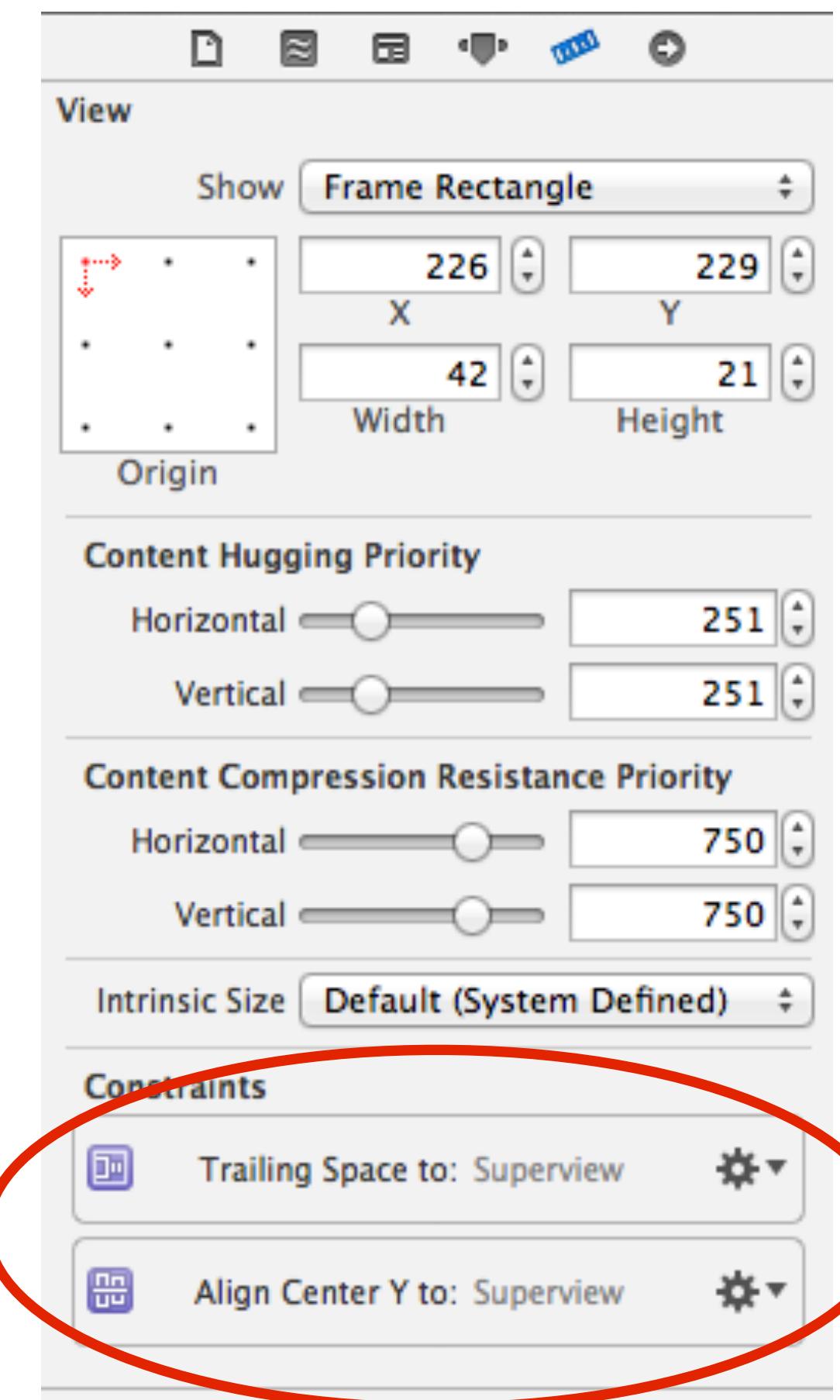
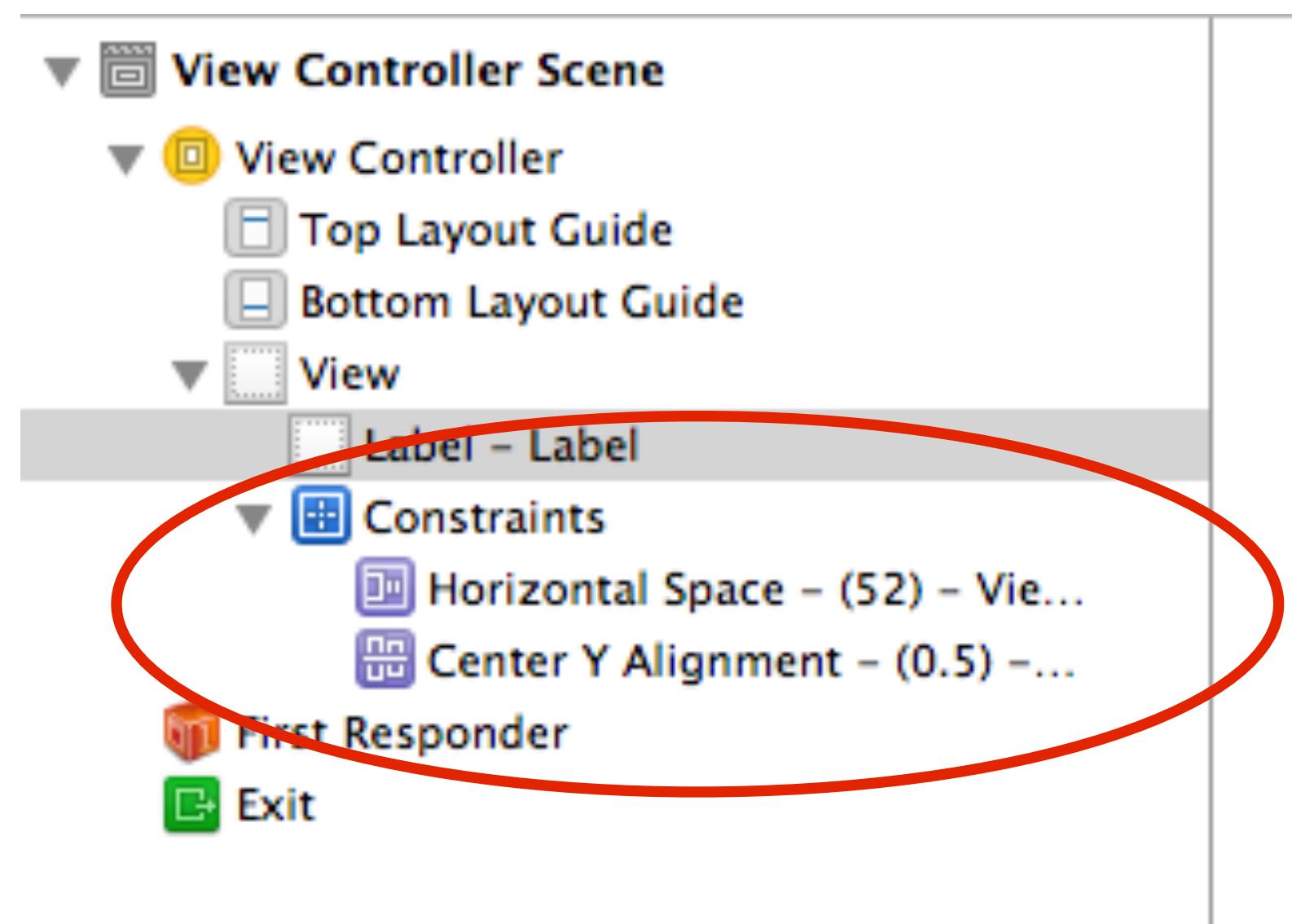
Adding Constraints with Control-Drag

- The second constrained is added
 - The yellow line has disappeared and has become blue: all good!
 - A new line has appeared for the second constraint



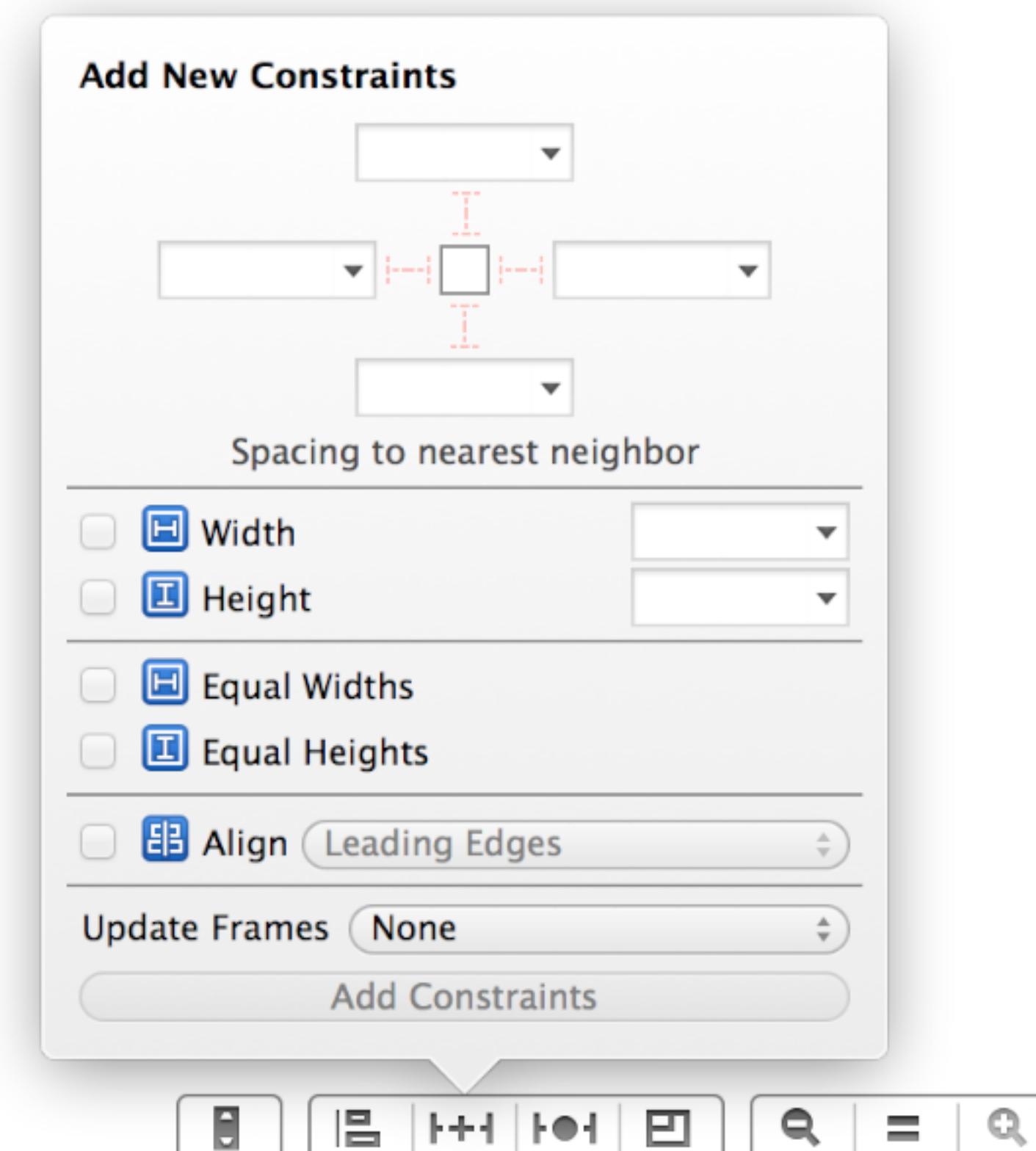
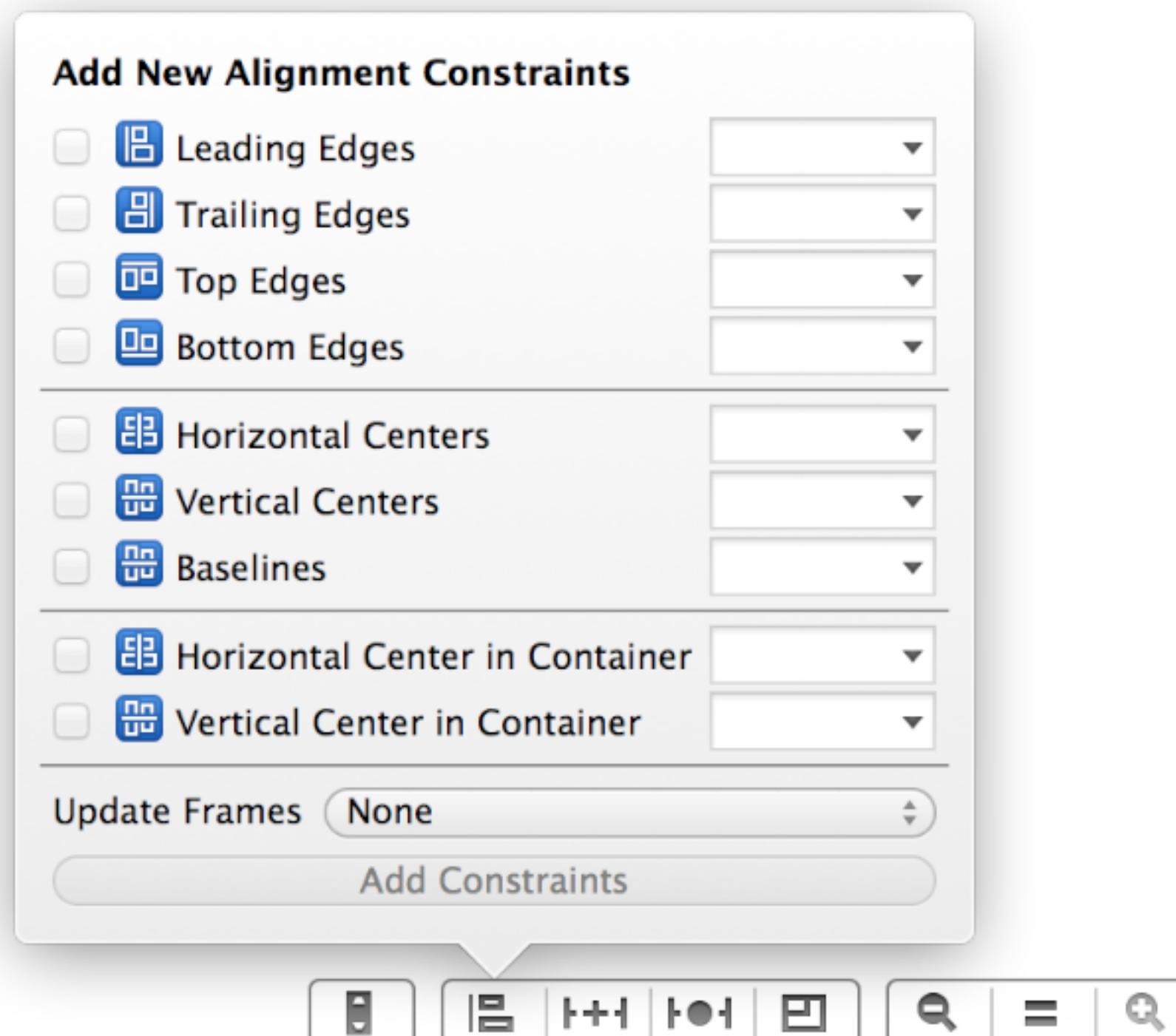
Adding Constraints with Control-Drag

- Constraints are also visible:
 - in the Document Layout view in storyboard
 - in the Attributes inspector



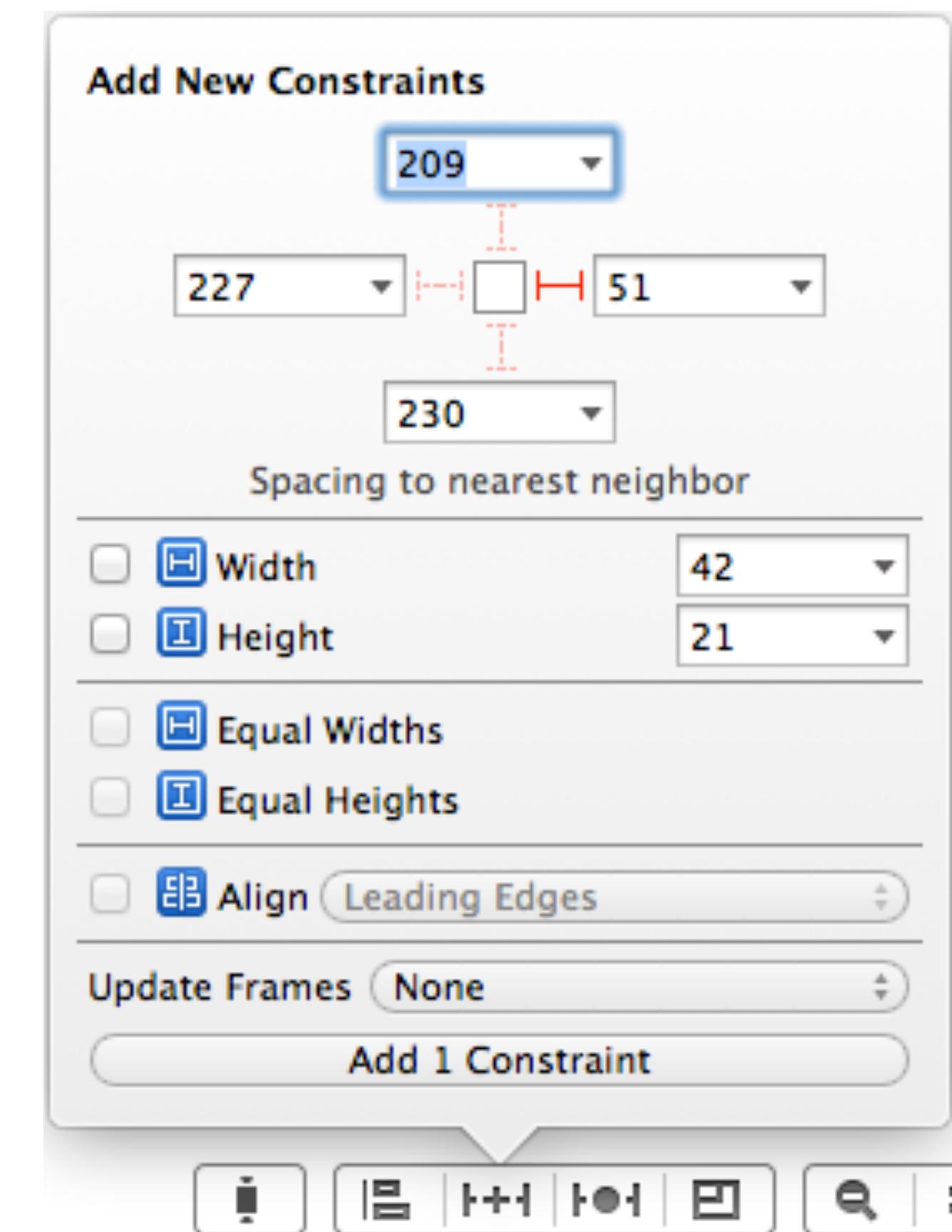


Adding constraints with align and pin



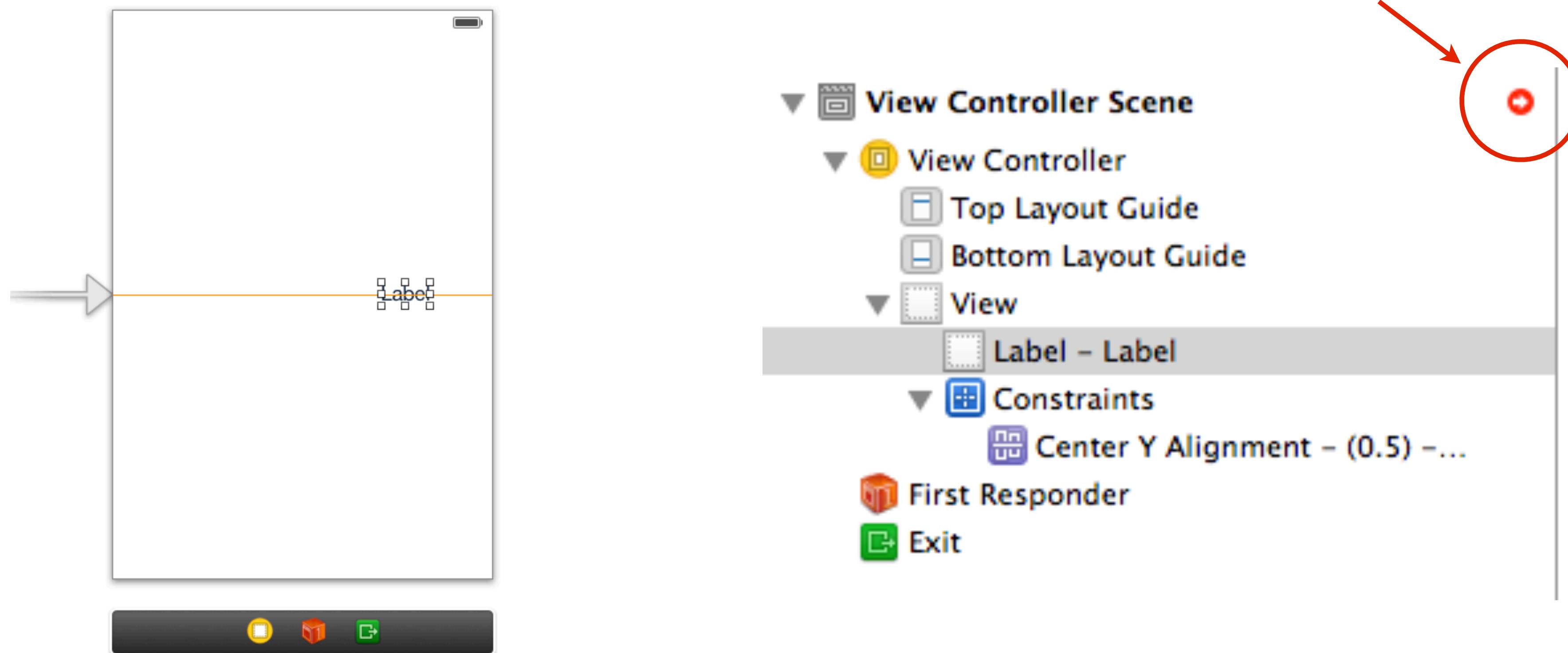
Adding constraints with align and pin

1. Select the checkbox next to the appropriate constraint:
 - to select a “Spacing to nearest neighbor” constraint, select the red constraint corresponding to the appropriate side of the element
2. Enter the constant value
3. Create the constraints by clicking the Add Constraints button: the new constraints are added to the selected elements



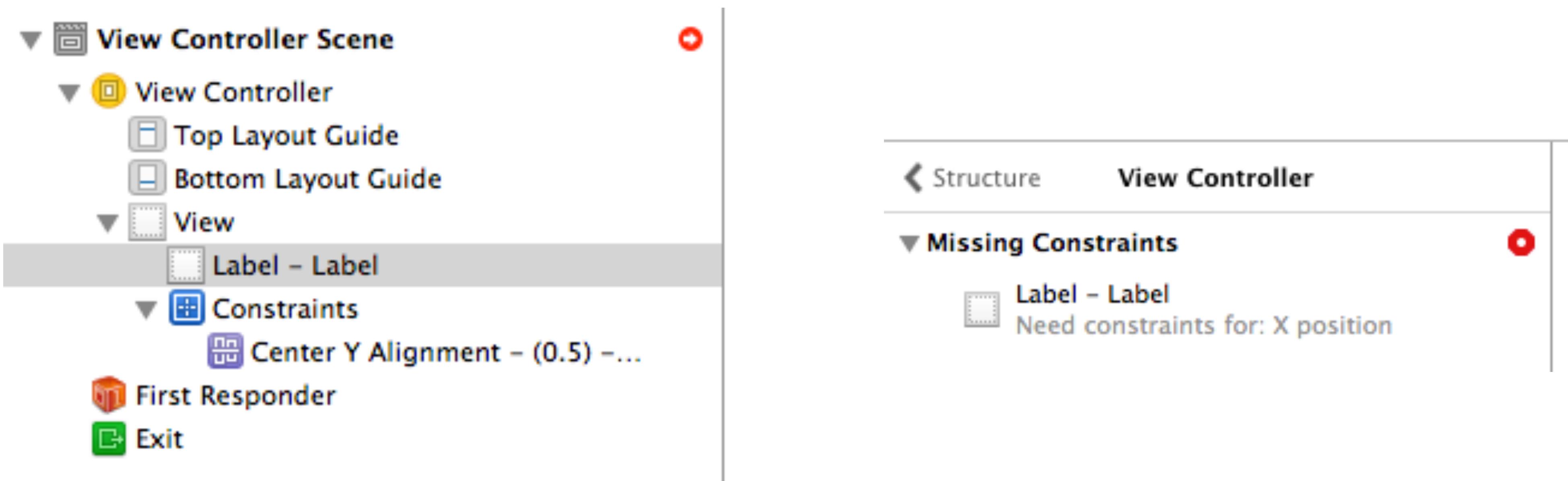
Adding missing constraints

- Let's go back when only the first constraint was set
- In the Document Layout a red arrow appears (it means there are problems)



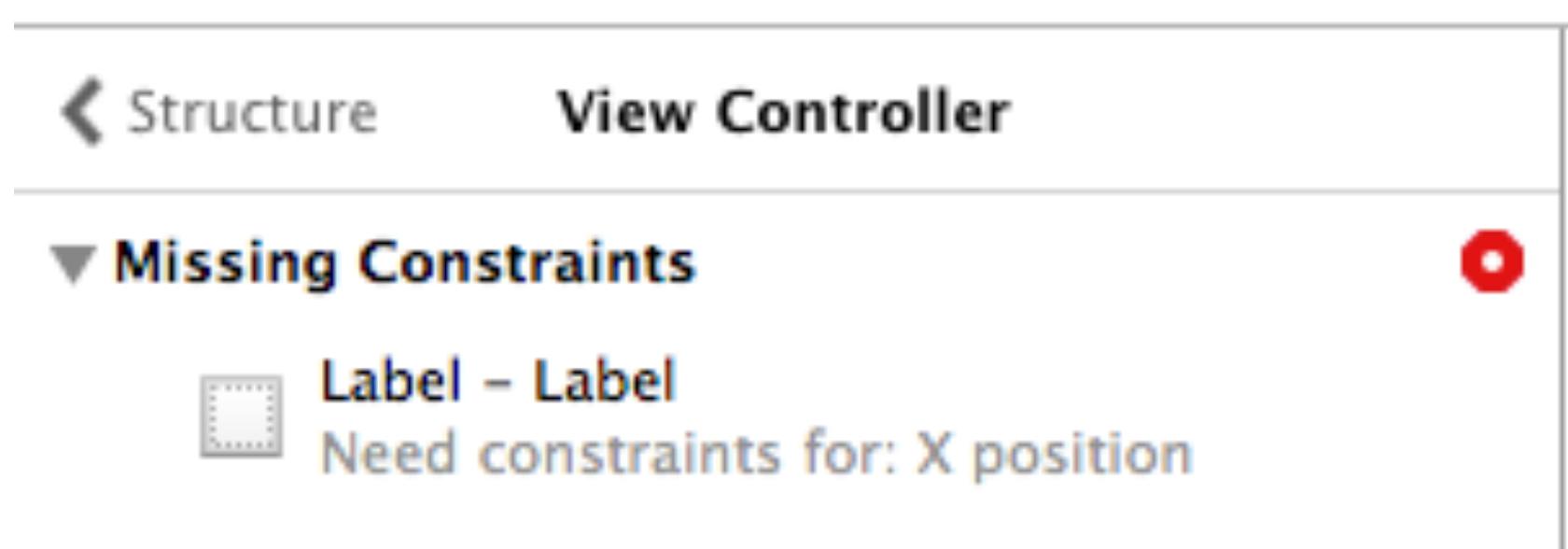
Adding missing constraints

- By clicking on the red arrow a new view appear, showing which elements have layout issues (only one in this case)

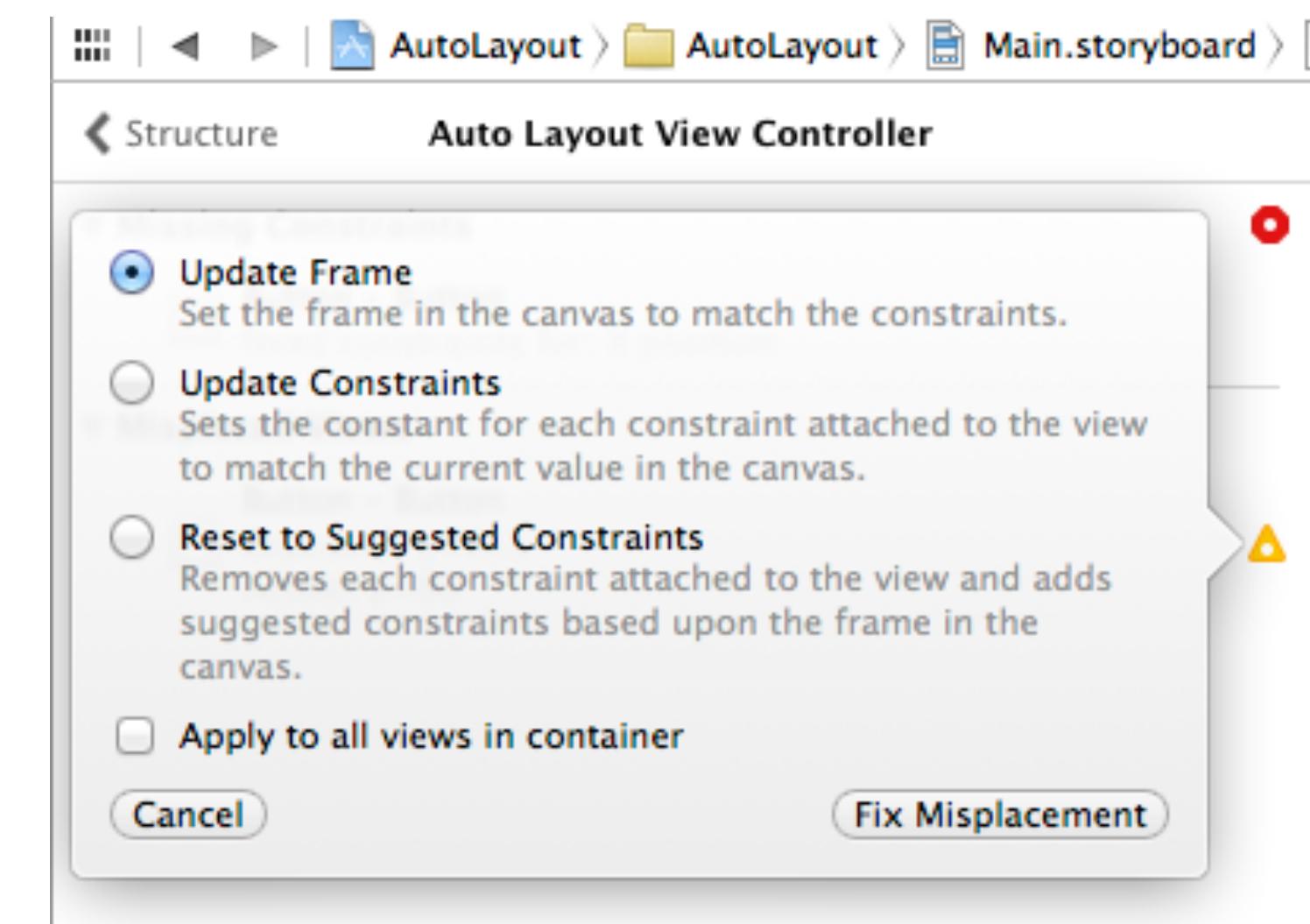


Adding missing constraints

- By clicking on the red circle, a popup appears
- Auto Layout is able to add the missing constraints automatically

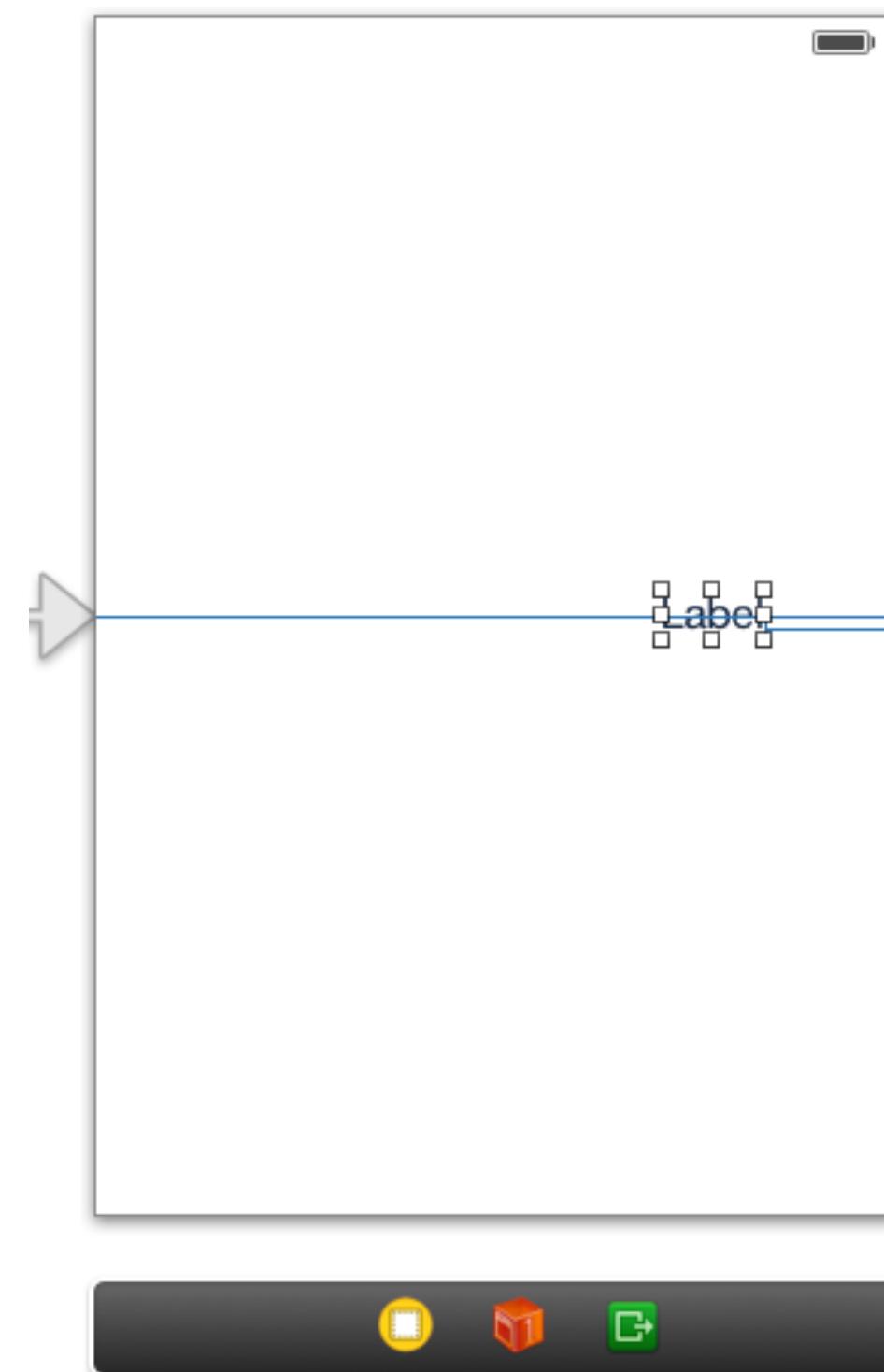


- If many solutions are available (not in this case), another popup shows all the possible actions that may be taken



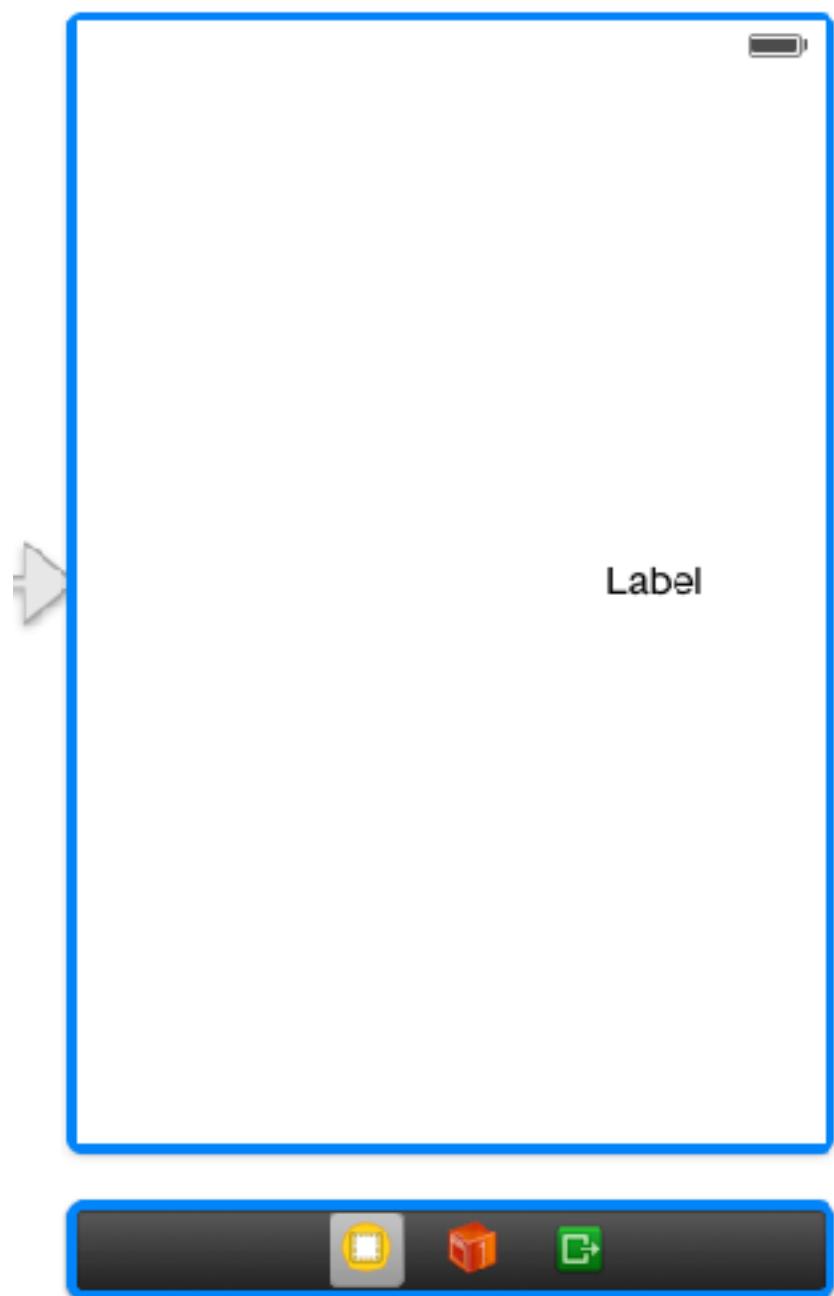
Adding missing constraints

- By selecting Add Missing Constraints, Auto Layout creates the new constraints
- Since this was the only issue, all lines in the view become blue



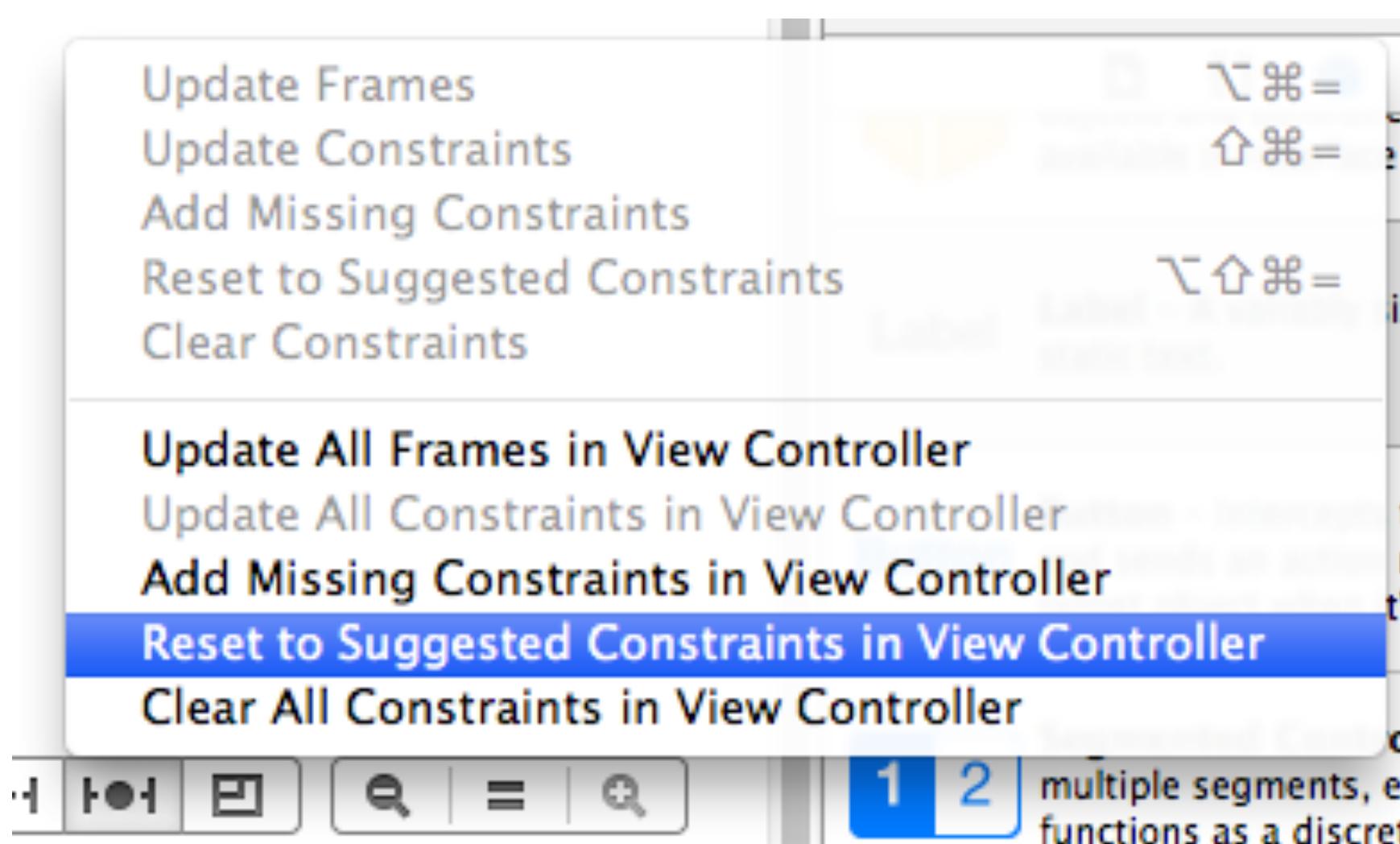
Adding missing constraints

- If no constraints have been, it is possible to ask Auto Layout to set all the constraints for the view:
 1. Select the View Controller, by clicking on the View Controller icon



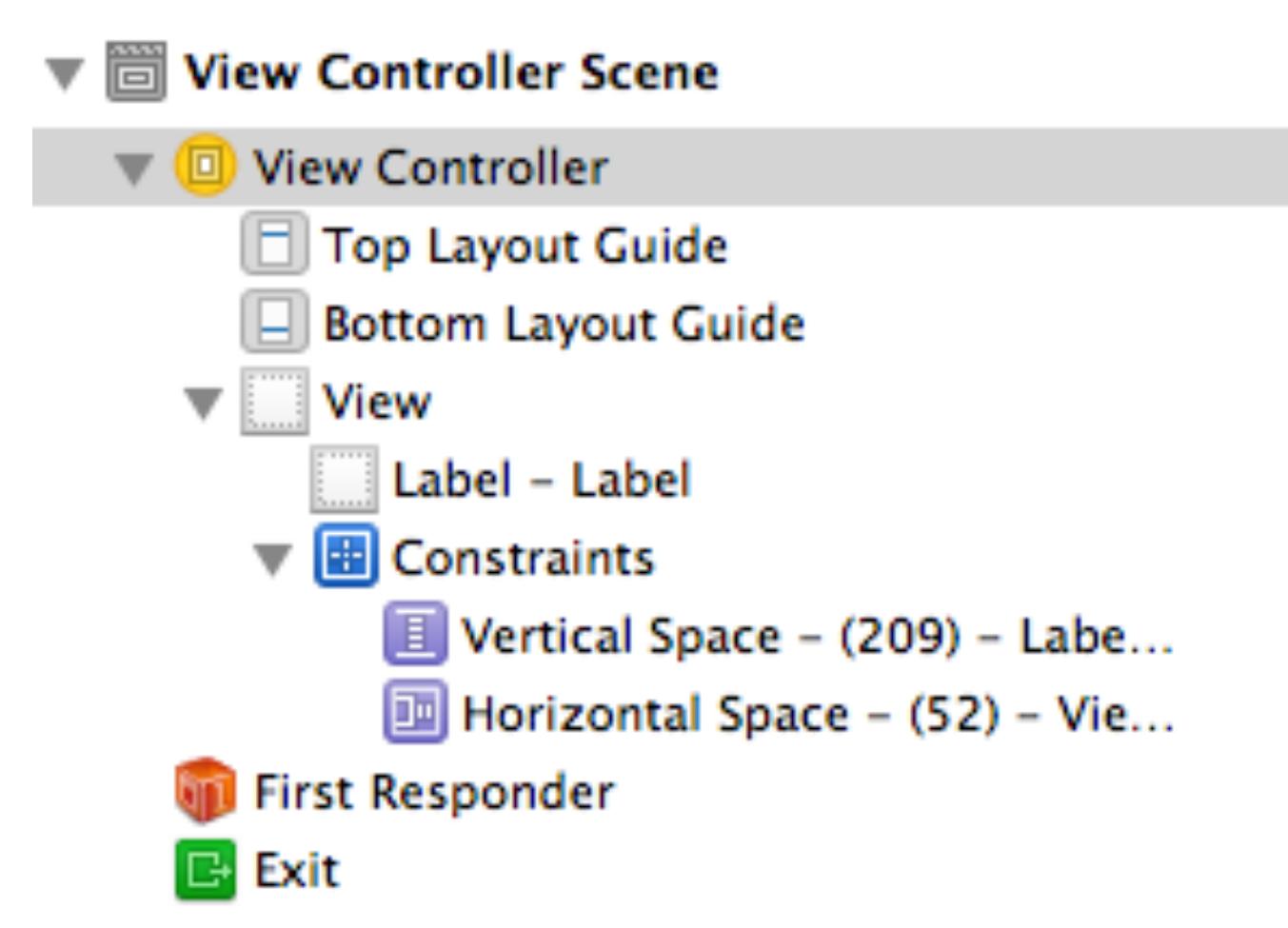
Adding missing constraints

- If no constraints have been, it is possible to ask Auto Layout to set all the constraints for the view:
 1. Select the View Controller, by clicking on the View Controller icon
 2. In the Resolve Issues menu, select **Reset to Suggested Constraints in View Controller**

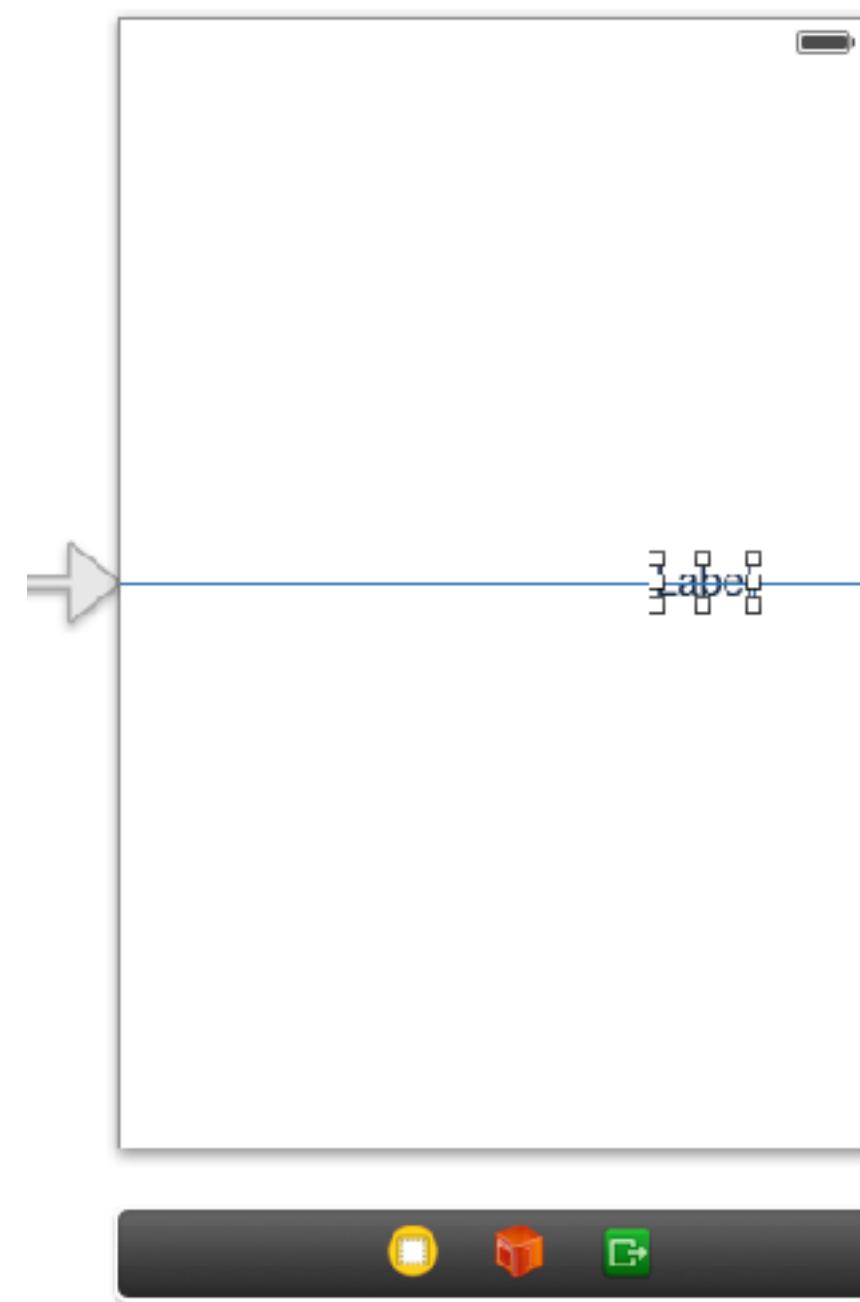


Adding missing constraints

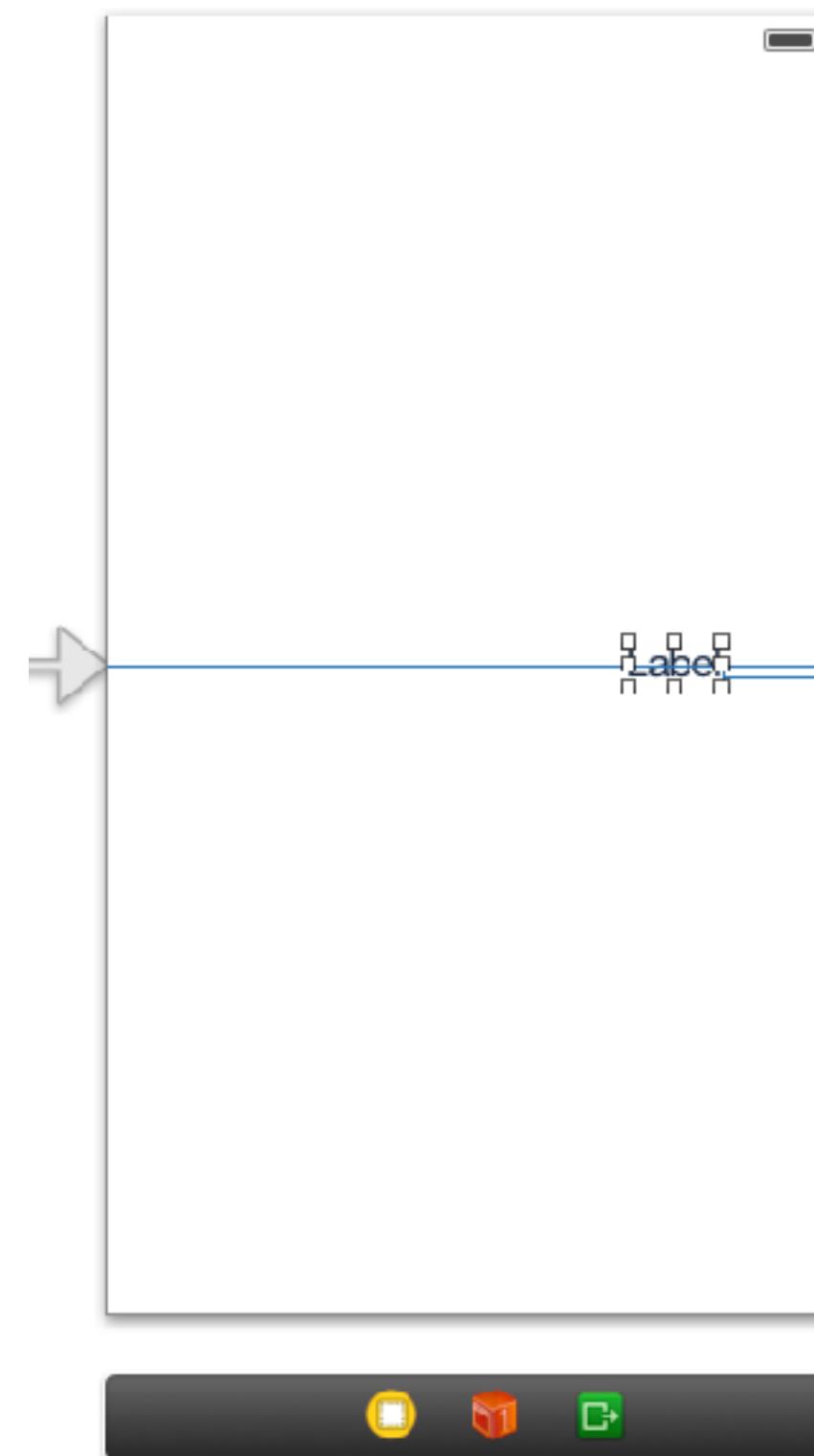
- If no constraints have been, it is possible to ask Auto Layout to set all the constraints for the view:
 1. Select the View Controller, by clicking on the View Controller icon
 2. In the Resolve Issues menu, select **Reset to Suggested Constraints in View Controller**
 3. The constraints are added (note that center vertically is not there!)



Checking constraints for different screen sizes

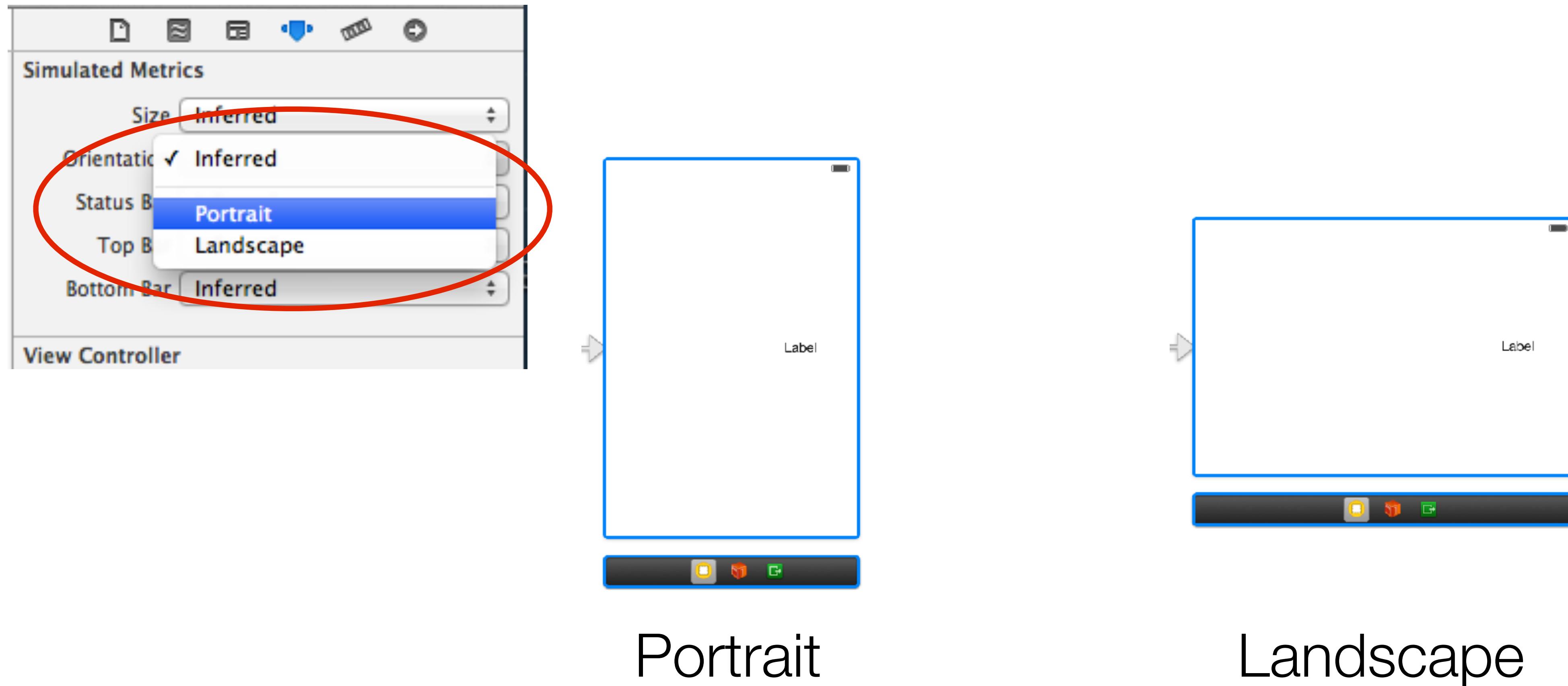


3.5" screen



4" screen

Checking constraints for different orientations



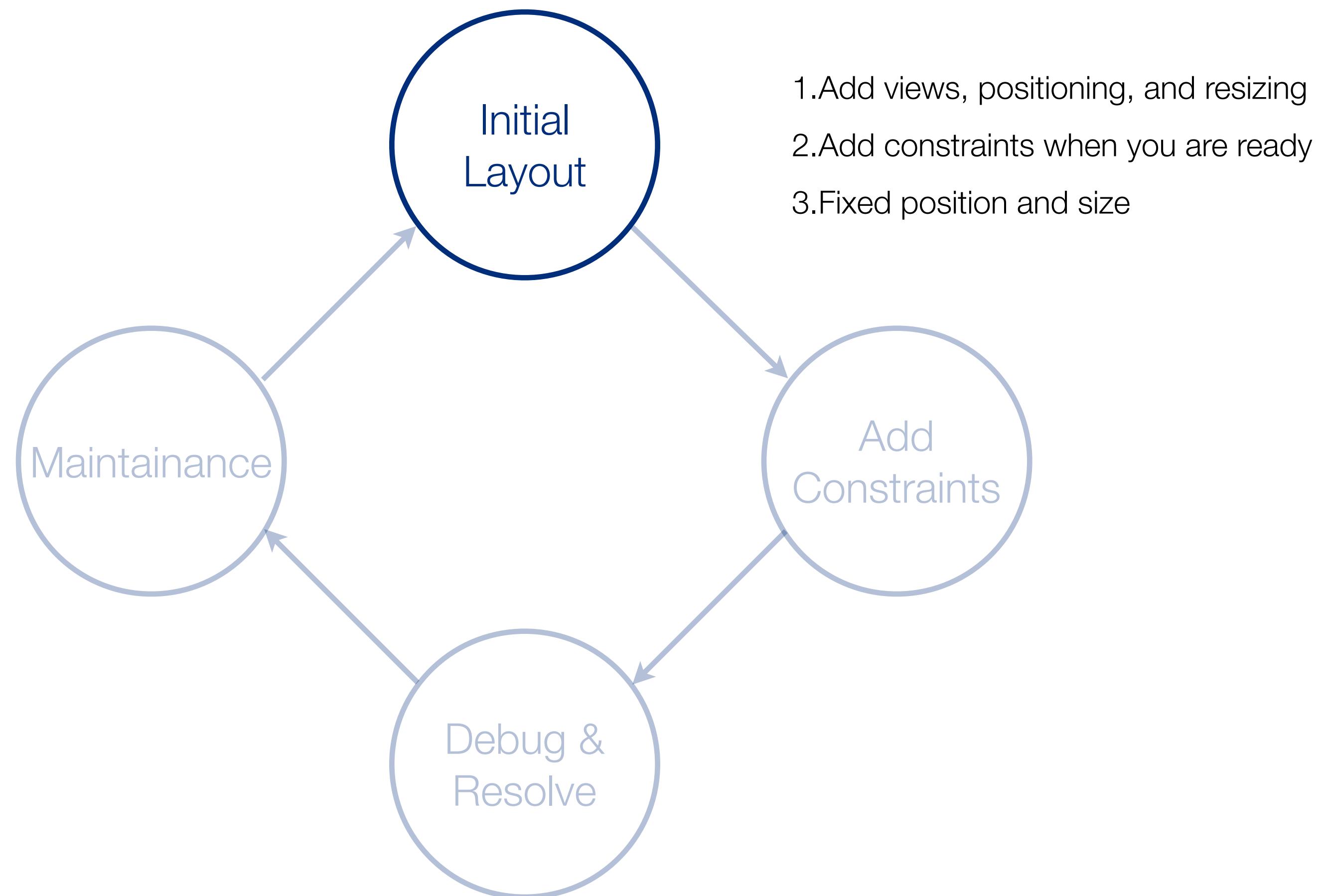


Deleting and editing constraints

- Constraints can be selected and deleted or edited
- Delete by using “delete” on the keyboard
- Editing can be made by using the menus shown previously

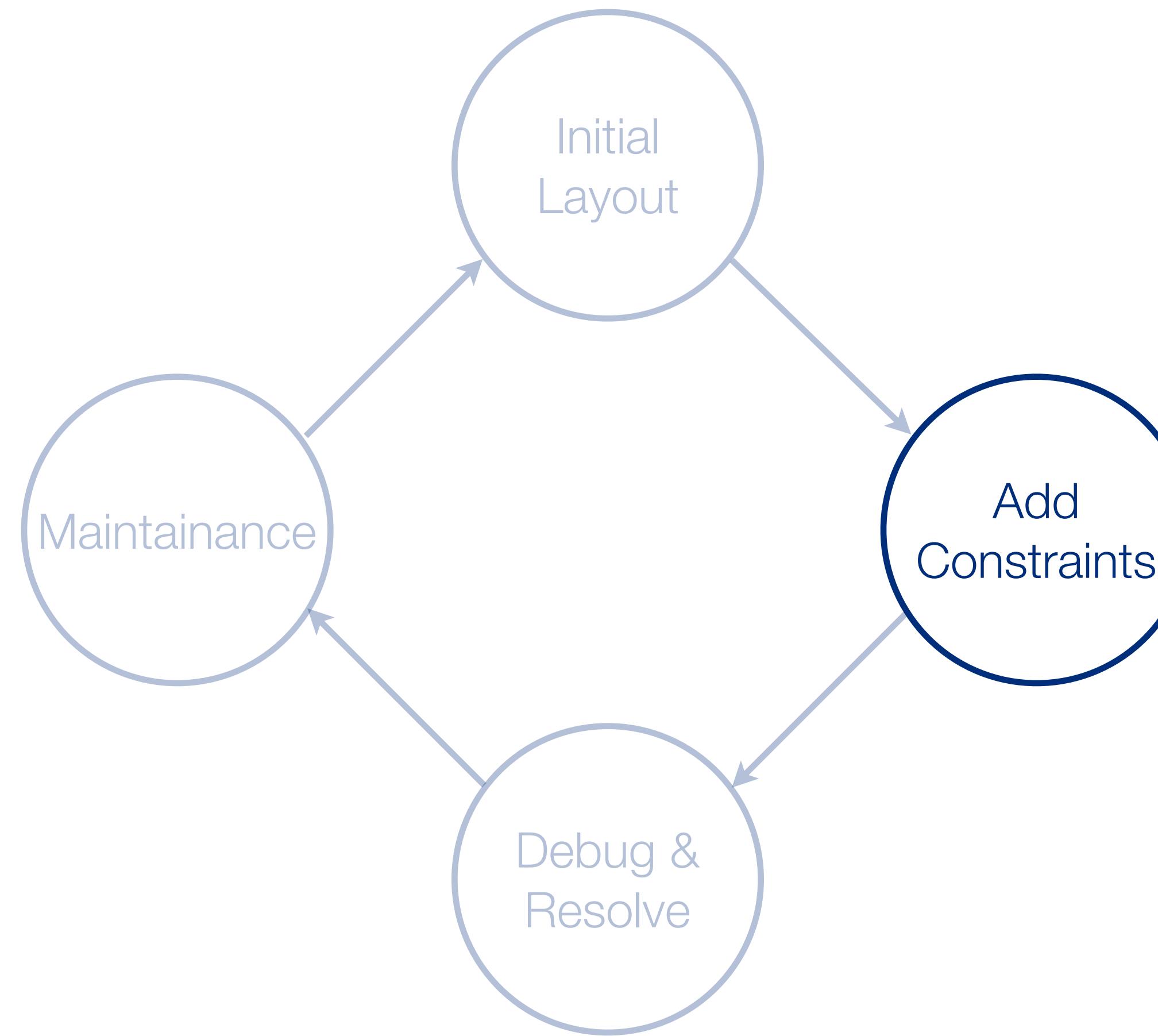
Auto Layout process

- When working with Auto Layout, typically an iterative process is performed



Auto Layout process

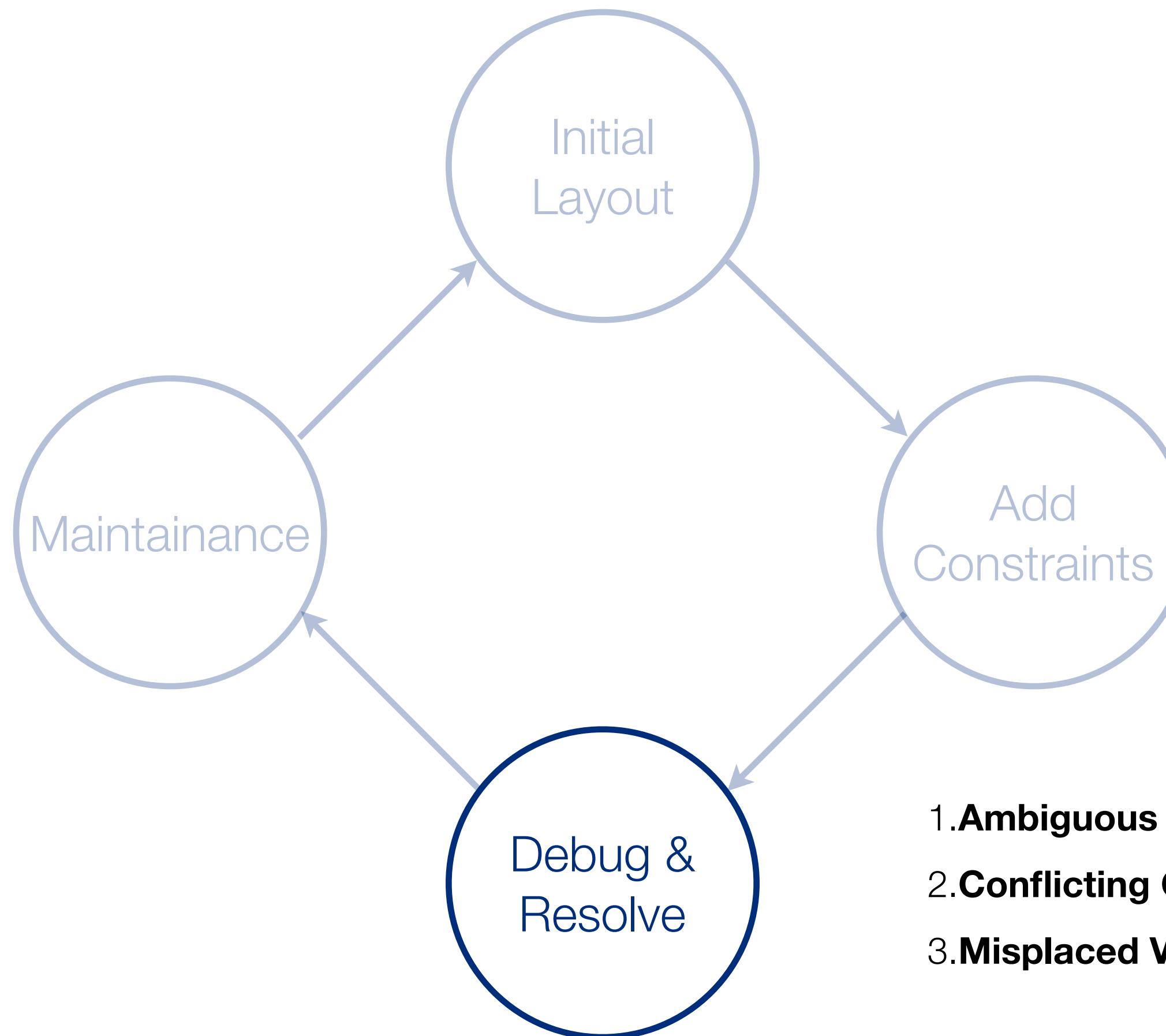
- When working with Auto Layout, typically an iterative process is performed



1. Direct manipulation: Control drag between views
2. Auto Layout resolving menu
3. Constraint addition popovers

Auto Layout process

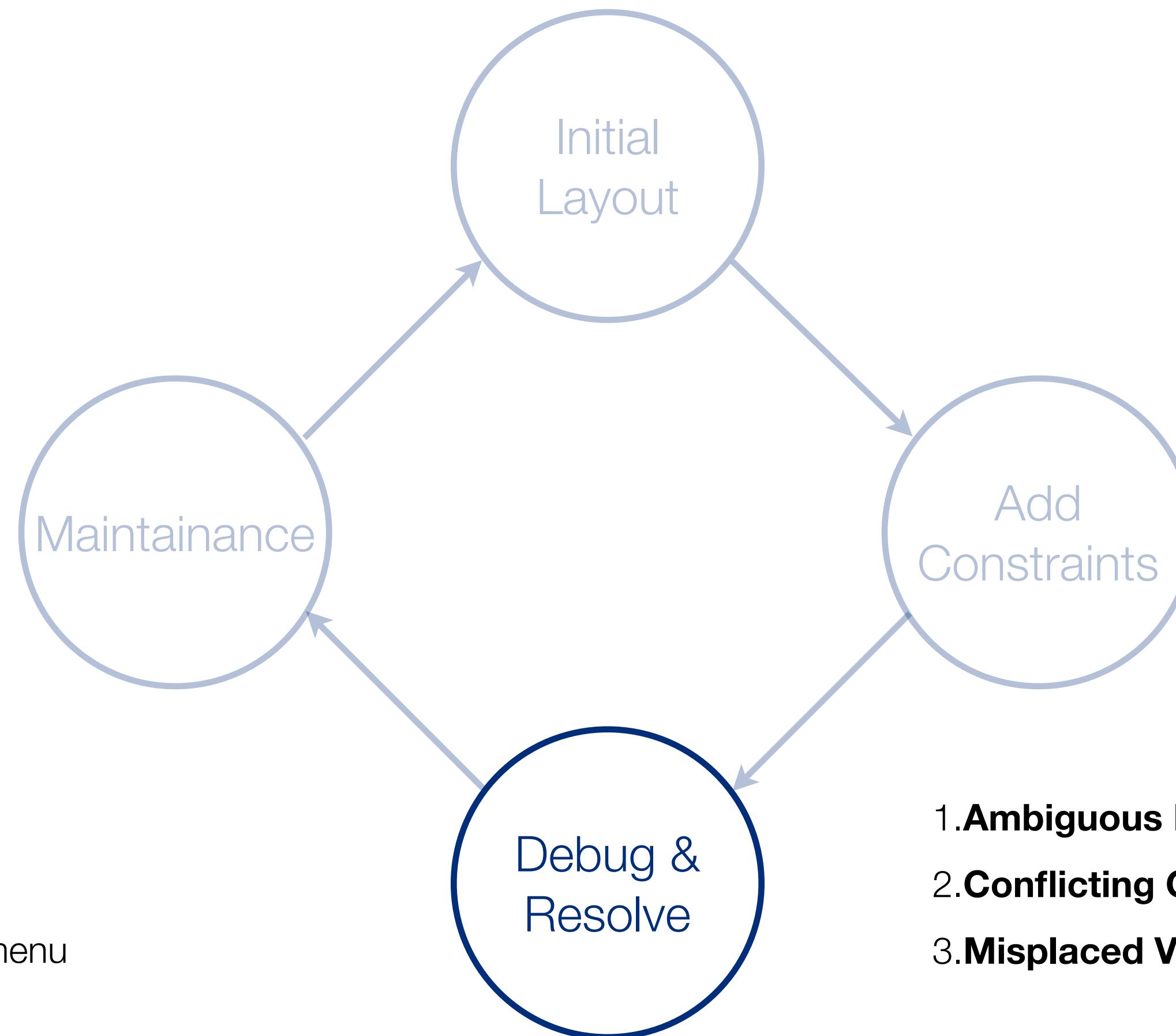
- When working with Auto Layout, typically an iterative process is performed



1. **Ambiguous Frames:** Not enough information
2. **Conflicting Constraints:** Too much information
3. **Misplaced Views:** Mismatched position or size

Auto Layout process

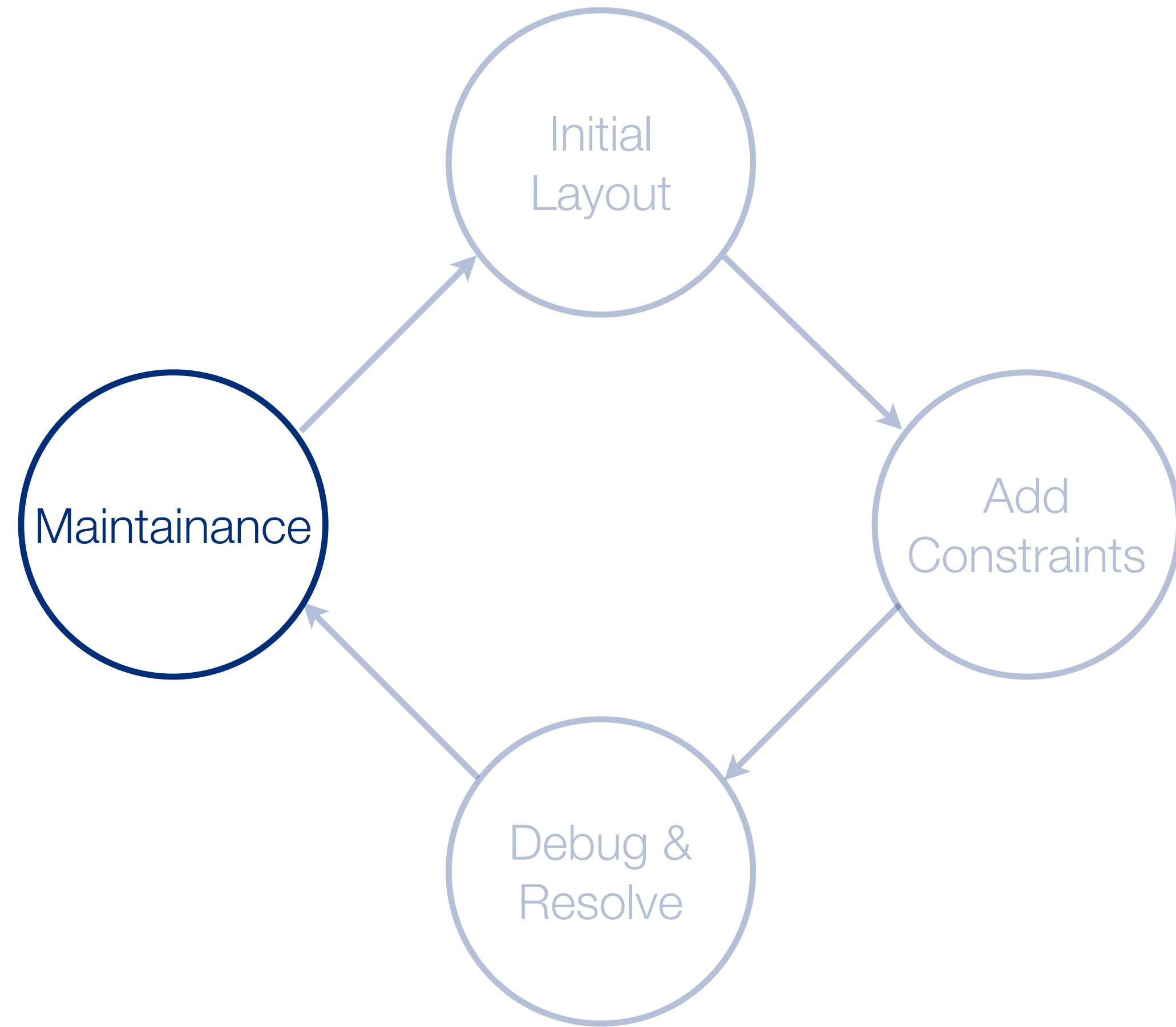
- When working with Auto Layout, typically an iterative process is performed



Auto Layout process

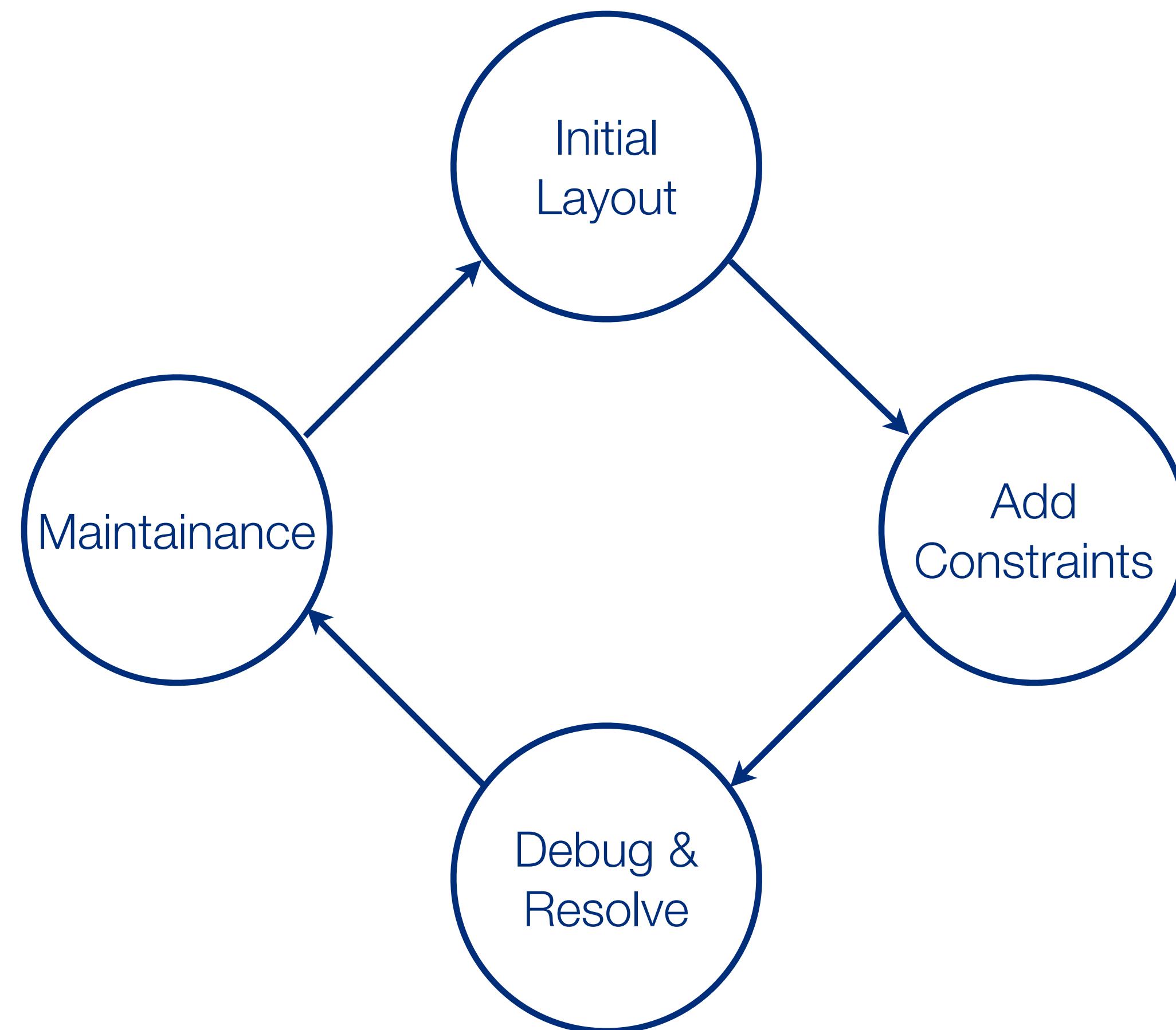
- When working with Auto Layout, typically an iterative process is performed

1. Check all changes keep things as they are supposed to be



Auto Layout process

- When working with Auto Layout, typically an iterative process is performed





Mobile Application Development



Lecture 10
Auto Layout



This work is licensed under a [Creative Commons Attribution-
NonCommercial-ShareAlike 4.0 International License](#).