Fully updated for iOS 7!

iOS Apprentice

SECOND EDITION

Tutorial 3: MyLocations

By Matthijs Hollemans

The iOS Apprentice:
MyLocations

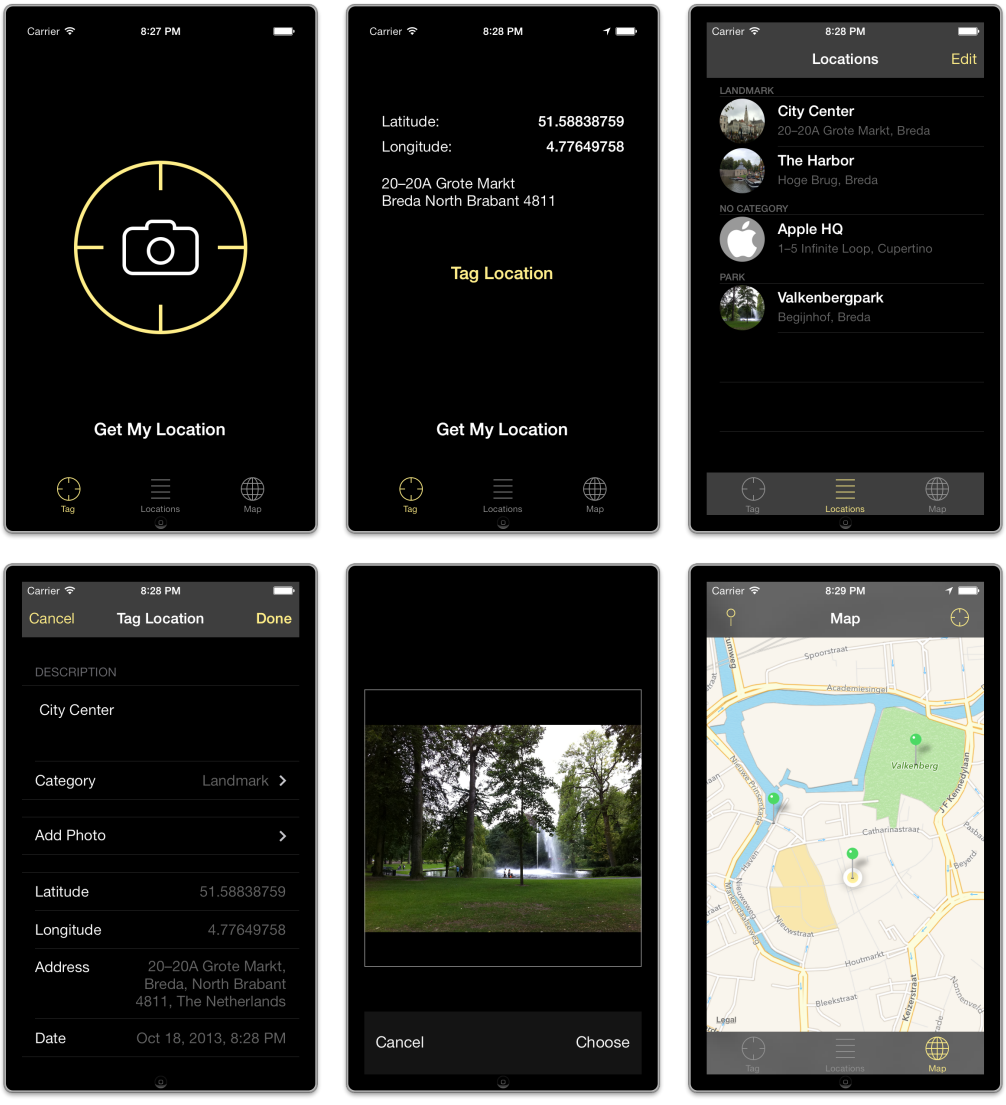
By Matthijs Hollemans

Table of Contents

In this tutorial you will look at some of the more alluring technologies from the iOS SDK and you'll dive a lot deeper into the Objective-C language.

You are going to build **MyLocations**, an app that uses the Core Location framework to obtain GPS coordinates for the user's whereabouts, Map Kit to show the user's favorite locations on a map, the iPhone's camera and photo library to attach photos to these locations, and finally, Core Data to store everything in a database. Phew, that's a lot of stuff!

The finished app looks like this:



The MyLocations app

MyLocations lets you keep a list of spots that you find interesting. You go somewhere with your iPhone or iPod touch and press the Get My Location button to obtain GPS coordinates and the corresponding street address. You can save this

location along with a description and a photo in your list of favorites for reminiscing about the good old days. Think of this app as a "location album" instead of a photo album.

To make the workload easier to handle, you'll split up the project into smaller chunks:

1. You will first figure out how to obtain GPS coordinates from the Core Location framework and how to convert these coordinates into an address, a process known as **reverse geocoding**. Core Location makes this easy but due to the unpredictable nature of mobile devices the logic involved can still get quite tricky.
2. Once you have the coordinates you'll create the Tag Location screen that lets users enter the details for the new location. This is a table view controller with static cells, very similar to what you've done in the previous tutorial.
3. You'll store the location data into a Core Data store. In the previous tutorial you saved the app's data into a .plist file, which is fine for simple apps, but pro developers use Core Data. It's not as scary as it sounds! You'll also show the locations as pins on a map.
4. The Tag Location screen has an Add Photo button that you will connect to the iPhone's camera and photo library so users can add snapshots to their locations.
5. Finally, you'll make the app look good with custom graphics. You will also add sound effects and some animations into the mix.

But before you get to all that, first some theory. There is still a lot more to learn about Objective-C and object-oriented programming!

Objective-C review

In the past tutorials I've shown you a fair bit of the Objective-C programming language already, but not quite everything. I've glossed over some of the details and told you a few small lies just so you wouldn't drown in new information. Previously it was good enough if you could more-or-less follow along with what we were doing, but now is the time to fill in the gaps in the theory.

First, let's do a little refresher on what we've talked about so far.

Variables and types

A **variable** is a temporary container for a specific type of value:

```
int count;
BOOL shouldRemind;
NSMutableArray *list;
NSString *text;
```

The **datatype**, or just **type**, of a variable determines what kind of values it can contain. Some variables hold primitive values such as `int` and `BOOL`, others hold objects such as `NSMutableArray` and `NSString`. You can tell the difference by the `*` symbol. More about the difference between primitive types and objects shortly.

The primitive types you've used so far are: `int` for whole numbers, `float` for decimals, and `BOOL` for boolean values (`YES` and `NO`).

There are a few other primitive types as well:

- `double`. Similar to a `float` but with more precision. You will use `double`s later in this tutorial for storing latitude and longitude data.
- `char` and `unichar`. These hold just a single character (as opposed to a string, which can have more than one character). Often `char` is also used to mean a single *byte*, which is the smallest unit of storage that computers can use (that's why the size of memory or disk drives is always measured in mega-bytes and giga-bytes).
- `short`, `long`, `long long`. These are all variations on `int`. The difference is in how many bytes they have available to store their values. The more bytes, the bigger the values can be. In practice you almost always use `int`, which uses 4 bytes for storage (a fact that you may immediately forget) and can fit whole numbers ranging from minus 2 billion to plus 2 billion.
- `unsigned`. Another variation on `int` that you may encounter occasionally. "Unsigned" means a type will hold positive values only. An `unsigned int` holds values between 0 and plus 4 billion but no negative values.

The iOS SDK also uses synonyms for these primitive types:

- `NSInteger`, same as `int`.
- `NSUInteger`, same as `unsigned int` (positive values only).
- `CGFloat`, same as `float`.

Important: Even though the names of these types may look like objects, they are not because their names do not include a `*`.

All of these different type names can be confusing at first but fortunately the compiler can easily convert between them. For instance, it is usually no problem if you're using an `int` when the SDK expects an `NSInteger`.

However, on the new 64-bit iPhone 5S, an `int` is still 32-bits but an `NSInteger` is twice as big, 64-bits. That's why it's generally better to use `NSInteger` or `NSUInteger` instead of a plain `int`, to prevent any weird situations on the new 64-bit devices.

Should your type be incompatible with a variable or method, Xcode will let you know with a warning. Pay attention to those warnings!

Variables with primitive types can be used directly,

```
count = 10;
```

```
shouldRemind = YES;
```

but objects must be allocated first:

```
list = [[NSMutableArray alloc] initWithCapacity:10];
```

Strings, even though they are objects, can also be used directly:

```
NSString *text = @"Hello, world";
```

The integer value `10`, the boolean `YES` and the string `@"Hello, world"` are named **literal constants** or just **literals**.

You've seen two types of variables: **local variables**, whose existence is limited to the method they are declared in, and **instance variables** (also known as "ivars") that belong to the whole object and therefore can be used from within any method. The lifetime of a variable is called its **scope**. The scope of a local variable is smaller than that of an instance variable.

```
@implementation MyObject
{
  int _count;     // an instance variable
}

- (void)myMethod
{
  int temp;       // a local variable
  temp = _count;  // OK to use the instance variable here
}
```

I recommend that you declare your instance variables after the `@implementation` line in the .m file. You may also run into code that puts them below the `@interface` line in the .h file. That is still valid but no longer the best way.

If you have a local variable with the same name as an instance variable, then it is said to **shadow** (or **hide**) that instance variable. You should avoid these situations as they can lead to subtle bugs where you may not be using the variable that you think you are:

```
@implementation MyObject {
  int count;      // an instance variable
}

- (void)myMethod
{
  int count = 0;  // local variable shadows instance variable
}
```

Xcode, helpful as ever, will warn you about such situations. However, you can avoid this problem altogether if you always place an underscore in front of your instance variable names: `_count` instead of `count`.

A variable stores only a single value. To keep track of multiple objects you can use a so-called **collection** object. Naturally, I'm talking about arrays (`NSArray`) and dictionaries (`NSDictionary`), both of which you've seen in the previous tutorial.

An **array** stores a list of objects. The objects it contains are ordered sequentially and you retrieve them by index. A **dictionary** stores key-value pairs. One object, usually a string, is the key that retrieves another object. There are other sorts of collections as well but array and dictionary are the most common ones.

By the way, the C language that Objective-C is based on has its own built-in array type but you'll hardly ever need to use it. Just so you know, it looks like this:

```
int oldSchoolArray[10];

. . .

oldSchoolArray[0] = 100;
oldSchoolArray[1] = 200;
```

The main problem with these arrays is that they aren't very flexible. If you declare a C array that can hold 10 items as in the above example, then you can't add more than those ten – the array won't expand to make room for additional elements. C-style arrays can be practical, but only hardcore (or old-fashioned) programmers use them these days.

Methods

You've learned that objects, the basic building blocks of all apps, have both data and functionality. The variables provide the data, **methods** provide the functionality.

These are examples of methods:

```
// method that doesn't have any parameters,
// doesn't return a value
- (void)viewDidLoad;

// method that has one parameter, sender, but doesn't
// return a value (IBAction means the same as void)
- (IBAction)showAlert:(id)sender;

// method that has one parameter, returns a BOOL value
(YES or NO)
```

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
                  (UIInterfaceOrientation)interfaceOrientation;

// method with two parameters, tableView and section,
// that returns an NSInteger value (synonym for int)
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section;

// method with two parameters, tableView and indexPath,
// that returns an NSIndexPath object
- (NSIndexPath *)tableView:(UITableView *)tableView
              willSelectRowAtIndexPath:(NSIndexPath *)indexPath;
```

Methods in Objective-C often have very long names. The full names of the methods from the above example are:

```
viewDidLoad

showAlert:

shouldAutorotateToInterfaceOrientation:

tableView:numberOfRowsInSection:

tableView:willSelectRowAtIndexPath:
```

Note that the colon `:` is part of the name. `viewDidLoad` has no parameters so its name does not end in a colon, but `showAlert:` does have a parameter and therefore needs that colon. We've only briefly talked about **selectors** and dynamic method name resolution, but using the proper method name is important in those circumstances. Get the name wrong and the app crashes.

```
// this refers to a method with no parameters
@selector(showAlert)

// this refers to a method with one parameter
@selector(showAlert:)
```

More about this sort of thing later in this lesson.

To call a method on an object, you use `[ ]` brackets:

```
[self loadChecklists];

[lists addObject:checklist];
```

```
[self configureTextForCell:cell withChecklistItem:item];
```

The first word inside the brackets indicates the object whose method you're calling. `self` is a special keyword that lets an object call its own methods.

When you call a method, the app jumps to that section of the code, executes all the statements in the method one-by-one and optionally returns a value to the caller:

```
- (int)performUselessCalculation:(int)a
{
  int b = arc4random_uniform(100);
  int c = a / 2;
  return (a + b) * c;
}
```

Due to its C heritage, Objective-C allows you to call **functions** such as `arc4random_uniform()`. Functions serve the same purpose as methods – they bundle functionality into small re-usable units – but live outside of any objects. You can make your own functions but it's usually better to stick to methods.

Often methods have one or more **parameters**, so you can make them work on data that comes from different sources. A method that is limited to a fixed set of data is not very useful or reusable.

For example, take this method that has no parameters:

```
- (int)addValuesFromArray
{
  int total = 0;
  for (NSNumber *number in _array) {
    total += [number intValue];
  }
  return total;
}
```

The `_array` variable here is an instance variable. The `addValuesFromArray` method is tied closely to that instance variable, and is useless without it.

Suppose you add a second array to your app that you also want to apply this calculation to. One approach is to copy-paste the above method and change the name of the variable to that of the new array. That certainly works but it's not smart programming!

It is better to give the method a parameter that allows you to pass in the array object that you wish to examine, so the method becomes independent from any instance variables:

```
- (int)addValuesFromArray:(NSArray *)array
```

```
{
  int total = 0;
  for (NSNumber *number in array) {
    total += [number intValue];
  }
  return total;
}
```

Now you can call this method with any `NSArray` object as its parameter.

This doesn't mean methods should never use instance variables, but if you can make a method more generic by giving it a parameter then that is usually a good idea.

Inside a method you can do the following things:

- Create local variables.
- Do simple arithmetic with mathematical operators such as `+`, `-`, `*`, `/`, `%` (modulo).
- Put new values into variables (both local and instance variables).
- Call other methods.
- Make decisions with the `if` or `switch` statements.
- Perform repetitions with the `for` or `while` statements.
- Return a value to the caller.

Let's look at the `if` and `for` statements in more detail.

Making decisions

The if-statement looks like this:

```
if (count == 0) {
  text = @"No Items";
} else if (count == 1) {
  text = @"1 Item";
} else {
  text = [NSString stringWithFormat:@"%d Items", count];
}
```

The expression between the `( )` parentheses is called the **condition**. If a condition is true (or `YES` in Objective-C speak) then the statements in the following `{ }` block are executed. The `else` section gets performed if none of the conditions are true.

You use **relational operators** to perform comparisons between two values:

`==` equal to

`!=` not equal

- `>` greater than
- `>=` greater than or equal
- `<` less than
- `<=` less than or equal

You can use **logical** operators to combine two expressions:

- `a && b` is true if both `a` *and* `b` are true
- `a || b` is true when either `a` *or* `b` is true (or both)

There is also the logical **not** operator, `!`, that turns `YES` into `NO`, and `NO` into `YES`.

You can group expressions with `( )` parentheses:

```
if (((this && that) || (such && so)) && !other) {
  // statements
}
```

This reads as:

```
if (((this and that) or (such and so)) and not other) {
  // statements
}
```

Or if you want to see clearly in which order these operations are performed:

```
if (
      (this and that)
            or
      (such and so)
   )
   and
   (
      (not other)
   )
```

Of course, the more complicated you make it, the harder it is to remember exactly what you're doing!

When you use the `==` operator, the actual objects are compared:

```
- (BOOL)compare:(NSString *)a with:(NSString *)b
{
  return (a == b);
}
```

This only returns `YES` if `a` and `b` are the same object:

```
NSString *a = @"Hello, world";
NSString *b = a;
if ([self compare:a with:b]) {
  NSLog(@"They are equal!");
}
```

In the following case the objects aren't considered equal even though they have the same value, because they are two different **instances** in memory:

```
NSString *a = @"Hello, world";
NSString *b = [NSString stringWithFormat:
                                    @"Hello, %@", @"world"];
if ([self compare:a with:b]) {
  NSLog(@"They are equal!");    // won't print now
}
```

If you're interested in comparing the values of two objects but don't care whether they actually occupy the same spot in memory (i.e. are the same instance), then you should use `isEqual` or `isEqualToString` instead:

```
- (BOOL)compare:(NSString *)a with:(NSString *)b
{
  return [a isEqualToString:b];
}
```

There is also a lesser-used construct in the language for making decisions, the `switch` statement:

```
switch (condition) {
  case value1:
    // statements
    break;

  case value2:
    // statements
    break;

  case value3:
    // statements
    break;

  default:
    // statements
    break;
}
```

It works the same way as an `if` statement with a bunch of `else ifs`. The following is equivalent:

```
if (condition == value1) {
  // statements
} else if (condition == value2) {
  // statements
} else if (condition == value3) {
  // statements
} else {
  // statements
}
```

The `switch` statement is sometimes more convenient to use and can be faster than `if – else if` because the condition is only evaluated once, but it is also less flexible as the `case` values must be integer constants. That means you can't do something like this:

```
NSString *text = ...;

switch (text) {
  case @"A":
    // statements
    break;

  case @"B":
    // statements
    break;
}
```

You'll see the `switch` statement in action a little later in this tutorial.

Note that `if` and `return` can be used to return early from a method:

```
- (int)divide:(int)a by:(int)b
{
  if (b == 0) {
    NSLog(@"You really shouldn't divide by zero");
    return 0;
  }

  return a / b;
}
```

This can even be done for methods that don't return a value:

```
- (void)performDifficultCalculation:(NSArray *)list
```

```
{
  if ([list count] < 2) {
    NSLog(@"Too few items in list");
    return;
  }

  // perform the very difficult calculation
}
```

In this case, `return` simply means: "We're done with the method". Any statements following the `return` are skipped and execution immediately returns to the caller.

You could also have written it like this:

```
- (void)performDifficultCalculation:(NSArray *)list
{
  if ([list count] < 2) {
    NSLog(@"Too few items in list");
  } else {
    // perform the very difficult calculation
  }
}
```

Whichever you use is up to personal preference. If there are only two cases such as in the above example, I prefer to use an `else` but the early `return` will work just as well.

Sometimes you see code like this:

```
- (void)someMethod
{
  if (condition1) {
    if (condition2) {
      if (condition3) {
        if (condition4) {
          // statements
        } else {
          // statements
        }
      } else {
        // statements
      }
    } else {
      // statements
    }
  } else {
    // statements
```

```
  }
}
```

This can become very hard to read, so I like to restructure that kind of code as follows:

```
- (void)someMethod
{
  if (!condition1) {
    // statements
    return;
  }

  if (!condition2) {
    // statements
    return;
  }

  if (!condition3) {
    // statements
    return;
  }

  if (!condition4) {
    // statements
    return;
  }

  // statements
}
```

Both do exactly the same thing, but I find the second one much easier to understand. As you become more experienced, you'll start to develop your own taste for what looks good and what is readable code.

Loops

You've seen the for-statement for looping through an array:

```
for (ChecklistItem *item in self.items) {
  if (!item.checked) {
    count += 1;
  }
}
```

This performs the statements inside the `for` block once for each object from the `self.items` array. This type of for-loop is called **fast enumeration** and can be used only with certain collection objects such as `NSArray` and `NSDictionary`.

There is a more general purpose version of the for-statement that looks like this:

```
for (int i = start; i < end; i++) {
  // statements
}
```

You should read the section between the `( )` parentheses as:

```
(start condition; end condition; increment)
```

For example, when you see,

```
for (int i = 0; i < 4; i++) {
  NSLog(@"%d", i);
}
```

it means you have a for-loop,

- that has a **loop counter** named `i` starting at 0,
- that ends when `i` is no longer smaller than 4 (i.e. as soon as it becomes equal to 4),
- and increments `i` by 1 at the end of each **iteration** (one pass through the loop).

When you run this code, it will print to the Debug area:

```
0
1
2
3
```

It may look a little funky, but `for (int i = 0; i < X; i++)` is a very common idiom that simply means the loop performs X repetitions.

The notation `i++` means the same as `i += 1`. If you wanted to show just even numbers, you could change the `for` loop to:

```
for (int i = 0; i < 4; i += 2) {
  NSLog(@"%d", i);
}
```

Note that the scope of the variable `i` is limited to just this for-statement. You can't use it outside this statement, so its lifetime is even shorter than a local variable.

To write the loop through the `ChecklistItems` array using this general purpose form of the `for` statement, you'd do something like this:

```
for (int i = 0; i < [self.items count]; i++) {
  ChecklistItem *item = [self.items objectAtIndex:i];
  if (!item.checked) {
    count += 1;
  }
}
```

This is slightly more cumbersome (and potentially slower) than `for (item in array)`, which is why you'll see fast enumeration used most of the time.

The for-statement is not the only way to perform loops. Another very useful loop construct is the while-statement:

```
while (something is true) {
  // statements
}
```

The while-loop keeps repeating the statements until its condition becomes false. You can also write it as follows:

```
do {
  // statements
} while (something is true);
```

In the latter case, the condition is evaluated after the statements have been executed at least once.

You can rewrite the `ChecklistItems` loop as follows using a `while` statement:

```
int i = 0;
while (i < [self.items count]) {
  ChecklistItem *item = [self.items objectAtIndex:i];
  if (!item.checked) {
    count += 1;
  }
  i += 1;
}
```

Most of these looping constructs are really the same, they just look different. Each of them lets you repeat a bunch of statements until some ending condition is met. There really is no significant difference between using a `for`, `while` or `do - while` loop, except that one may be easier to read than the others, depending on what you're trying to do.

Just like you can prematurely exit from a method using the `return` statement, you can exit a loop at any time using the `break` statement:

```
BOOL found = NO;
for (NSString *string in _array)
{
  if ([string isEqualToString:text])
  {
    found = YES;
    break;
  }
}
```

This example loops through the array until it finds a string that is equal to the value of `text` (presumably another string) and then sets the variable `found` to `YES` and jumps out of the loop using `break`. You've found what you were looking for, so it makes no sense to look at the other objects in that array (for all you know there could be hundreds).

There is also a `continue` statement that is somewhat the opposite of `break`. It doesn't exit the loop but immediately skips to the next iteration.

Objects

Objects are what it's all about. They combine data (instance variables) with functionality (methods) into coherent, reusable units – that is, if you write them properly!

In your Objective-C programs you will use existing objects, such as `NSString`, `NSArray`, and `UITableView`, and you will also make your own.

To define a new object, you need a **.h** file that contains an `@interface` section:

```
@interface MyObject : NSObject

@property (nonatomic, strong) NSString *text;

- (void)myMethod;

@end
```

And a **.m** file that contains the `@implementation` section:

```
#import "MyObject.h"

@implementation MyObject
{
  int _anInstanceVariable;
```

```
}

- (id)init
{
  if ((self = [super init])) {
    // statements
  }
  return self;
}

- (void)myMethod
{
  // statements
}

@end
```

The .h file, also known as the **header file**, contains all the stuff (properties and methods) that you want to make visible to other objects. The .m file contains the implementation details of the object. Where the .h file describes *what* the object does, the .m describes *how* it does those things.

Instance variables are usually interesting only to the implementation of the object and not to any other objects, so they are kept "hidden" inside the .m file. The implementation often also contains more methods than are visible in the interface. These methods are meant to be used inside that object only, not by other objects. Think of the .m file as an object's private parts that no one else needs to see.

(Previous versions of the Objective-C compiler required that instance variables were specified in the `@interface` section in the header file, but that is no longer necessary. You'll see a lot of code on the internet and in books that still does this but nowadays instance variables really belong in the `@implementation` section in the .m file.)

When another object, let's say a view controller, wants to use `MyObject`, it needs to import the .h file to learn about the methods and properties that it can use from `MyObject`:

```
#import "MyObject.h"

@implementation SomeViewController

- (void)someMethod
{
  MyObject *myObject = [[MyObject alloc] init];
  myObject.text = @"Hello, world";
  [myObject myMethod];
```

```
}

. . .
```

To create a new object, you first call `alloc` to reserve memory for its data, followed by `init` to properly make the object ready for use (known as **initialization**).

`init` is a method like any other, and you can provide it yourself:

```
@implementation MyObject

- (id)init
{
  if ((self = [super init])) {
    // perform your own initialization here
    self.text = @"Hello, world";
  }
  return self;
}
```

There are a few rules to writing an `init` method. You always have to call `super`'s version of `init` and assign the result to `self`. You're supposed to continue with the initialization only when `self` is not `nil`. When you're done, you return the value of `self` to the caller. Weird, but that's the way it is done in Objective-C.

Objects can have more than one `init` method. You only initialize an object once (right after the call to `alloc`) but which `init` method you use depends on the circumstances. A `UITableViewController`, for example, can be initialized either with `initWithCoder` when automatically loaded from a storyboard, with `initWithNibName` when manually loaded from a nib file, or with `initWithStyle` when manually constructed without a storyboard or nib. Sometimes you use the one, sometimes the other.

Many objects have **convenience constructors** that combine the call to `alloc` and `init` into one statement:

```
// using alloc and init:
NSString *text = [[NSString alloc] initWithFormat:
                                              @"Item-%d", index];

// shorter version:
NSString *text = [NSString stringWithFormat:@"Item-%d", index];
```

Properties

Methods provide the object's functionality and instance variables contain the object's data, so then what is the purpose of **properties**? They are provided for convenience and to make code easier to read and easier to write.

`MyObject` has a property named `text`. This means you can do the following:

```
MyObject *myObject = [[MyObject alloc] init];
. . .
myObject.text = @"Hello, world";
```

This is known as **dot syntax** because it separates the different elements by a dot. If the `MyObject` instance was a property of another object, for example a view controller, then you might be able to change its text as follows:

```
someViewController.myObject.text = @"How's the weather?";
```

Without properties all of this becomes a lot more cumbersome. You'd have to change the definition of `MyObject` to something such as:

```
@interface MyObject : NSObject

- (void)setText:(NSString *)newText;
- (NSString *)text;

@end
```

The `setText` method is known as a **setter**, because it changes a value inside the object; the `text` method is a **getter** because it returns the value from that object.

The implementation of these methods looks something like this. Note that you also require an explicit instance variable to hold the text value.

```
@implementation MyObject
{
  NSString *_text;
}

- (void)setText:(NSString *)newText
{
  if (newText != _text) {
    _text = newText;
  }
}

- (NSString *)text
{
```

```
  return _text;
}

@end
```

To change the text, you'd have to write:

```
MyObject *myObject = [[MyObject alloc] init];
. . .
[myObject setText:@"Hello, world"];
```

Or if `MyObject` was part of that view controller:

```
[[someViewController myObject] setText:@"How's the weather?"];
```

Not only is all of this a lot more typing, it's also less readable. Instead of using dots to access the various elements, you're calling methods all the time. Too many of those square brackets will drive you crazy!

Properties were added to place the burden of writing these setter and getter methods on the Objective-C compiler rather than on the programmer.

That means when you do,

```
@property (nonatomic, strong) NSString *text;
```

the compiler automatically adds the `setText` and `text` methods to your object, as well as the instance variable that will hold the value for this property. Now you can simply use dot syntax to read and write the value of `text`.

Properties save you a bunch of typing, but there's another reason for using them. Just like you put methods in the object's `@interface` declaration to make functionality available to other objects, you put properties in the `@interface` to make data available to others. Because you use properties these other objects never directly access your internal instance variables, they always go through a getter or setter method first.

The keyword `IBOutlet` means that a property can be hooked up to a control in Interface Builder. The property declaration usually also includes other special keywords such as `nonatomic` and `strong`; we'll get into what these mean later in this tutorial.

Note: With older versions of Xcode you always needed to "synthesize" your properties. That meant you had to add a line such as this one to your .m file for each of your properties:

```
@synthesize text = _text;
```

> Fortunately, the designers of Objective-C decided that the compiler could figure out for itself that the properties need to be synthesized. There are still a couple of situations where you have to provide the `@synthesize` statement, but most of the time you can simply leave it out.

Protocols

Besides objects you can also define **protocols**. A protocol is simply a list of method names:

```
@protocol MyProtocol <NSObject>
- (void)requiredMethod;
@optional
- (void)optionalMethod;
@end
```

Whenever you see `< >` brackets in an object's `@interface` line, it means that object conforms to a protocol:

```
@interface MyObject : NSObject <MyProtocol>
. . .
@end
```

This object now has to provide an implementation for the `requiredMethod` in its `@implementation` section. It may also provide a method body for the `optionalMethod`, but that is not required (any methods that follow the `@optional` directive in a protocol declaration may be left out).

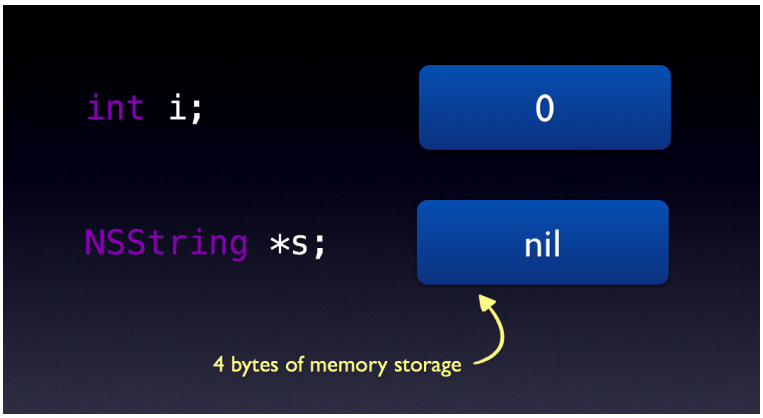
Protocols are often used to define **delegates**.

Objects and pointers

So what exactly is the difference between a primitive type and an object?

When you declare a variable as an `int`, the compiler reserves a small portion of memory to hold the value that you wish to store (4 bytes, to be exact).

Variables for objects are no different. When you declare a variable as an `NSString *`, the compiler also reserves 4 bytes of memory, exactly the same as for an `int`. But an `NSString` object can hold many pages of text – surely that won't all fit into those 4 bytes?



The memory that is reserved for two variables

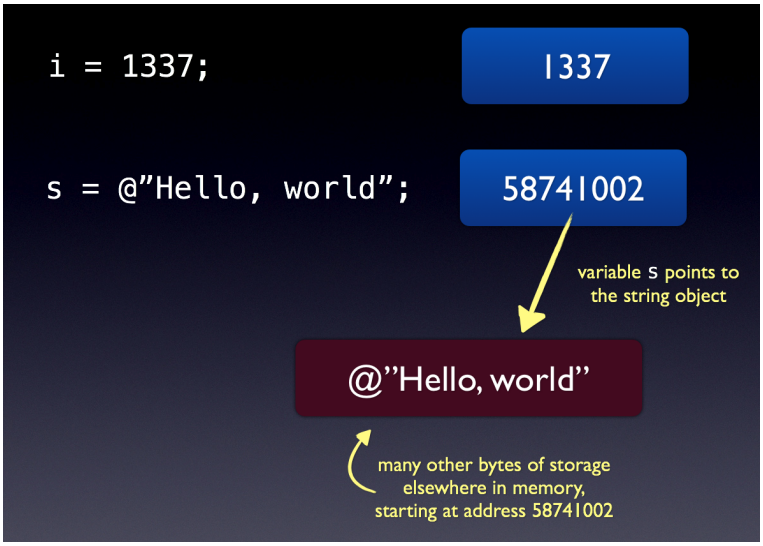
Declared like this, variable `i` holds the value 0 and `s` contains the value `nil`, i.e. it doesn't actually have an object yet. These are the default values for new variables.

When you put some values of your own into these variables, the picture looks like this:

The memory after the variables have values

The value 1337 is put directly into `i`'s memory but the value of `s` is some weird number. It sure doesn't look like a string of text. That number is not the actual `NSString` object but the memory address where that `NSString` object lives. Every byte in memory has its own unique address, and in this case address 58741002 refers to the location in memory of the first byte of the string object.

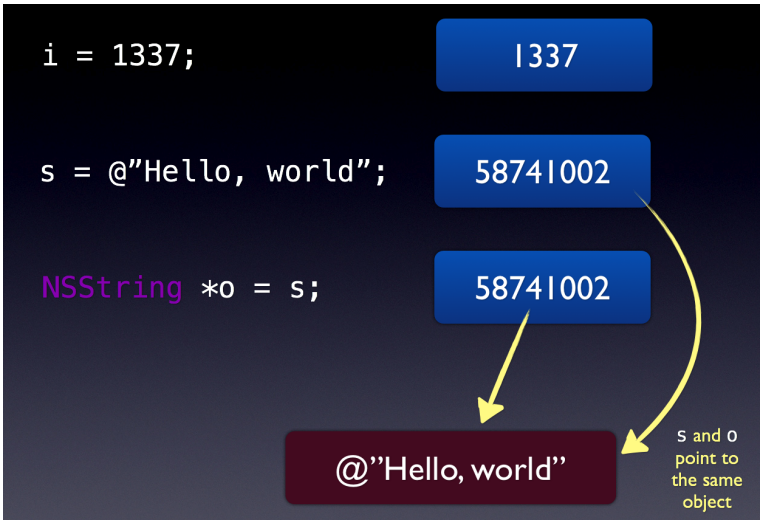
Obviously the 4 bytes that are reserved for the `s` variable are not enough to hold a string object with the text `@"Hello, world"`. So when the `NSString` object is allocated, the computer puts it somewhere else in memory and returns the address of that memory location. That address – PO Box 58741002, Joe's iPhone, USA – is what gets put in `s`.



Variable 's' contains the address of the string object

Variable `s` is called a **pointer**, because it contains an address of an object rather than the object itself. It is said to "point to" that object. The `*` asterisk symbol is used to make the distinction between a variable that contains a regular value and one that stores a pointer to another value.

You can create a new pointer variable `o` that is also an `NSString *` and give it the same value as `s`. That doesn't make a copy of the entire object, just of the address. The result is that both variables now point at the same object.



Variable 's' and 'o' both refer to the same object

Pointer indirection can be a difficult concept to grasp and that's why we've simply ignored it until now. There are very good reasons why Objective-C doesn't put objects directly into variables, but they're highly technical and boring (unless you like designing programming languages). Just accept it as a fact of life and move on: in Objective-C you refer to objects through pointers.

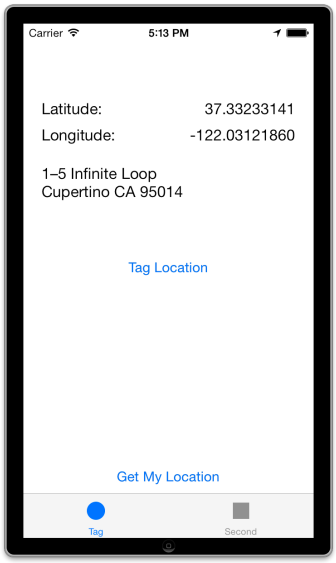
Note: The * asterisk symbol is mostly used for pointers to objects but you can also have a pointer to an primitive type, such as `int*` – that's simply a variable that refers to another `int` variable – although that kind of thing isn't very common in Objective-C. Because programming language designers like to see our brains melt, you can also have pointers to pointers. That's like trying to imagine what something looks like in five dimensions.

This concludes the quick recap of what you've seen so far of the Objective-C language. After all that theory, it's time to write some code!

Getting GPS coordinates

In this section you'll create the project in Xcode and then use the Core Location framework to find the latitude and longitude of the user's location.

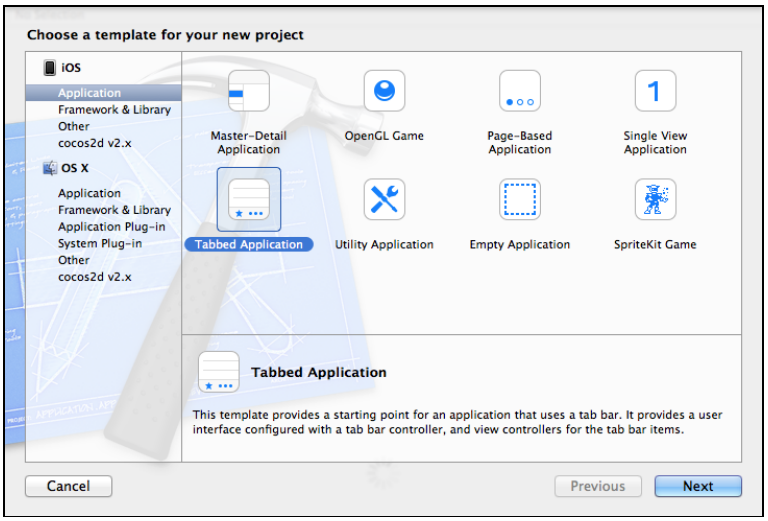
When you're done with this section, the app will look like this:



The first screen of the app

I know it's not very good looking yet, but you'll fix that later. For now, it's only important that you can obtain the GPS coordinates and show them on the screen. As always, you first make things work and then make them look good.

› Fire up Xcode and make a new project. Choose the **Tabbed Application** template.



Choosing the Tabbed Application template

➤ Fill in the options as follows:

- Product Name: **MyLocations**
- Organization Name: Your own name or the name of your company
- Company Identifier: Your own identifier in reverse domain notation
- Class Prefix: Leave this empty
- Devices: iPhone

➤ Save the project.

If you now run the app, it looks like this:

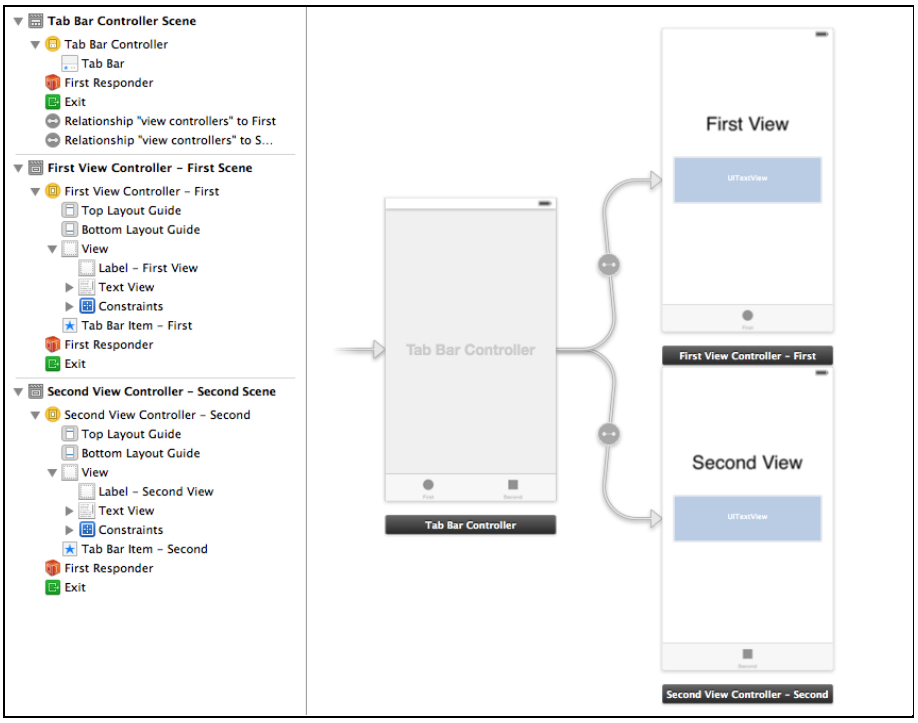
The app from the Tabbed Application template

The app has a tab bar along the bottom with two tabs: First and Second.

Even though it doesn't do much yet, the app already employs three view controllers: its *root controller* is the `UITabBarController` that contains the tab bar and performs the switching between the different screens.

The two tabs each have their own view controller. By default the Xcode template names them `FirstViewController` and `SecondViewController`.

The storyboard looks like this:



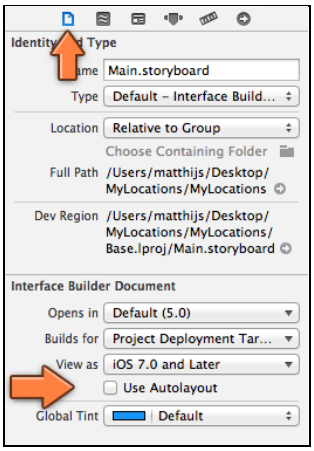
The storyboard from the Tabbed Application template

I already had to zoom it out to fit the whole thing on my screen. Storyboards are great but they sure take up a lot of space!

In this section you'll be working with the first tab only. In the second half of the tutorial you'll create the screen for the second tab and add a third tab as well.

There is one change you should make at this point: disabling Auto Layout. You've seen a glimpse of Auto Layout in the Bull's Eye chapter where you used it to make the app fit on both 3.5 and 4-inch devices. The Checklists app also had Auto Layout enabled but didn't really use it. Auto Layout is a great technology for making complex user interfaces but it can also be quite confusing to use – it can be more trouble than it's worth. For this tutorial you disable Auto Layout altogether.

▸ Go to the storyboard's **File inspector** and uncheck the **Use Autolayout** option.



Disabling Auto Layout in the File inspector for the storyboard

Let's give `FirstViewController` a better name.

▸ Use Xcode's Refactor tool (**Edit ☐ Refactor ☐ Rename**) to rename `FirstViewController` to `CurrentLocationViewController`. You've done this a couple of times now so I'm sure you can manage on your own.

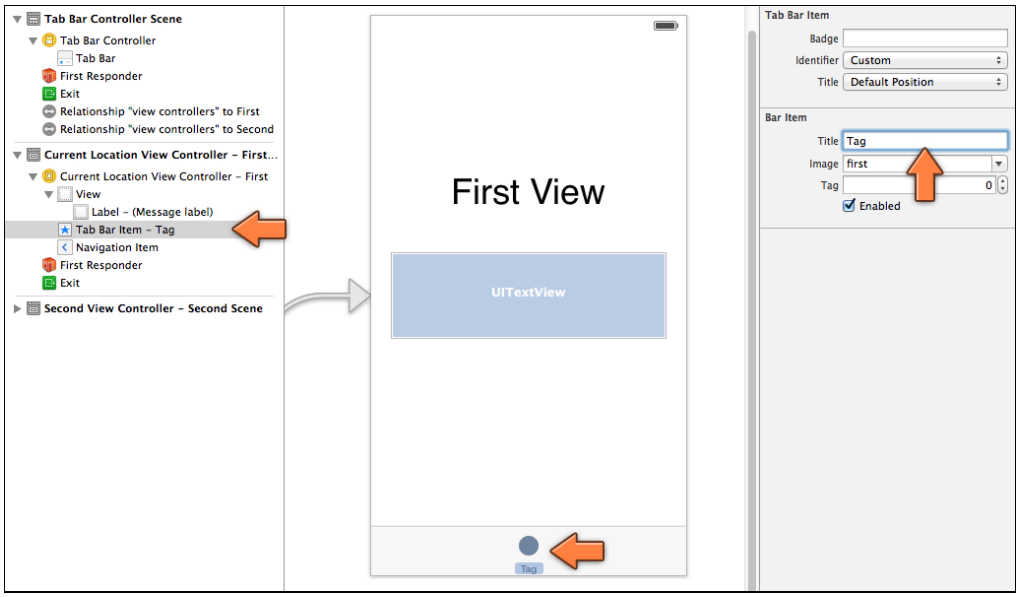
▸ Verify that Xcode also changed the class of this view controller in the storyboard (the one connected to the first tab). You can see this in the **Identity inspector**. Xcode should have automatically done this, but it's good to double-check.

▸ Go into the **Project Settings** screen and de-select the Landscape Left and Landscape Right settings under **Deployment Info**, **Device Orientation**. Now the app is portrait-only.

▸ Run the app again just to make sure everything still works. Whenever I change how things are hooked up in the storyboard, I find it useful to run the app and verify that the change was successful – it's way too easy to forget a step and you want to catch such mistakes right away.

As you've seen, a view controller that sits inside a navigation controller has a Navigation Item object that allows it to configure the navigation bar. Tab bars work the same way. Each view controller that represents a tab has a Tab Bar Item object.

▸ Select the Current Location view controller's Tab Bar Item object and go to the **Attributes inspector**. Change the Title to **Tag**. Later on you'll also set an image (it currently uses the default image of a circle from the template).



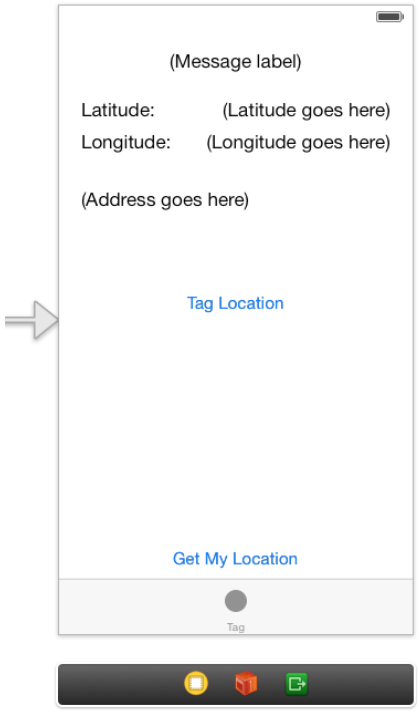
Changing the title of the Tab Bar Item

You will now design the screen for this first tab. It gets two buttons and a few labels that will hold the GPS coordinates and the street address. To save you some time, you'll add all the outlet properties in one go.

➤ Add the following to **CurrentLocationViewController.h**:

```
@property (nonatomic, weak) IBOutlet UILabel *messageLabel;
@property (nonatomic, weak) IBOutlet UILabel *latitudeLabel;
@property (nonatomic, weak) IBOutlet UILabel *longitudeLabel;
@property (nonatomic, weak) IBOutlet UILabel *addressLabel;
@property (nonatomic, weak) IBOutlet UIButton *tagButton;
@property (nonatomic, weak) IBOutlet UIButton *getButton;

- (IBAction)getLocation:(id)sender;
```

➤ Design the UI to look as follows:



The design of the Current Location screen

› The **(Message label)** at the top should span the whole width of the screen. You'll use this label for status messages while the app is obtaining the GPS coordinates. Set the **Alignment** attribute to centered and connect the label to the `messageLabel` outlet.

› Make the **(Latitude goes here)** and **(Longitude goes here)** labels right-aligned and connect them to the `latitudeLabel` and `longitudeLabel` outlets respectively.

› The **(Address goes here)** label also spans the whole width of the screen and is 50 points high so it can fit two lines of text. Set its **Lines** attribute to 0 (that means it can fit a variable number of lines). Connect this label to the `addressLabel` outlet.

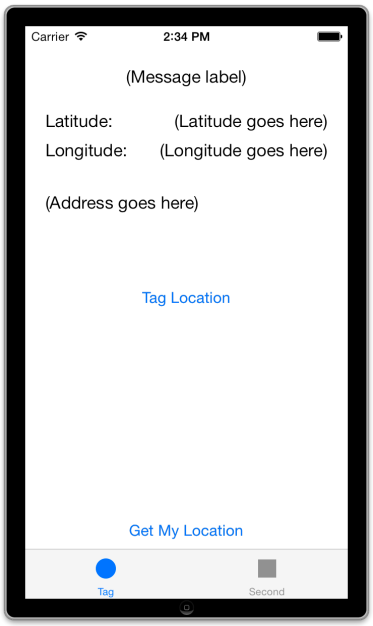
› The **Tag Location** button doesn't do anything yet but should be connected to the `tagButton` outlet.

› Connect the **Get My Location** button to the `getButton` outlet, and its Touch Up Inside event to the `getLocation:` action.

› To complete these changes, and to prevent the app from crashing when you tap the button, add the following to **CurrentLocationViewController.m**:

```
- (IBAction)getLocation:(id)sender
{
  // do nothing yet
}
```

➤ Run the app to see the new design in action:



The app with the new screen design on the 4-inch Simulator

> **Note:** If you're using the 3.5-inch Simulator, then switch to **iPhone Retina (4-inch)**. In this tutorial you will first design the app to work on 4" devices. Later on, you will also make it work on devices with smaller screens.

So far, nothing special. With the exception of the tab bar this is stuff you've seen and done before. Time to add something new: let's play with Core Location!

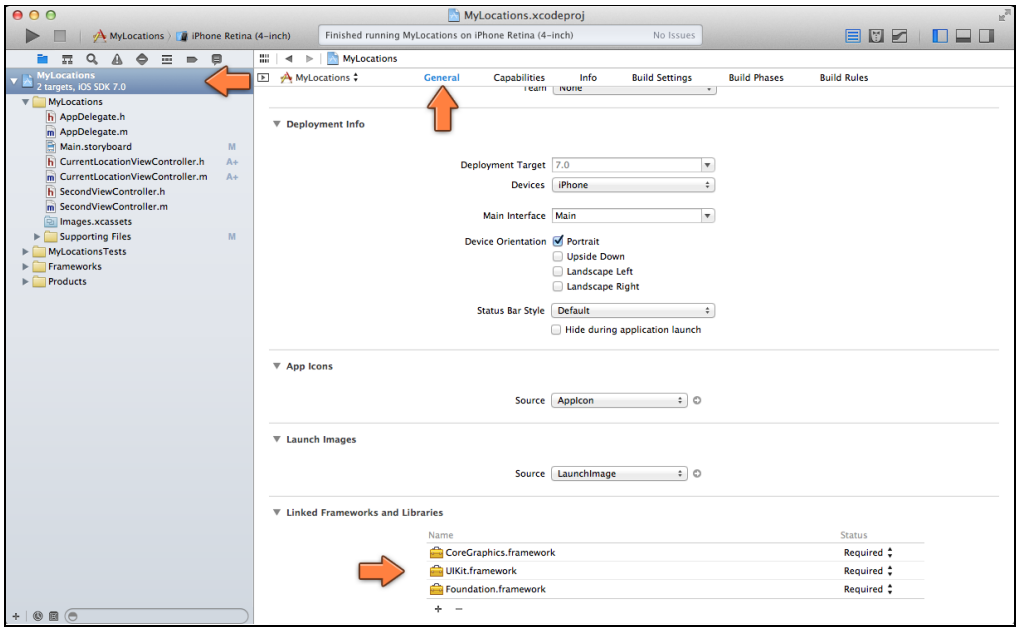
Core Location

Most iOS-enabled devices have a way to let you know exactly where you are on the globe, either through communication with GPS satellites, or Wi-Fi and cell tower triangulation. The Core Location framework puts that power into your own hands.

An app can ask Core Location for the user's current latitude and longitude, and for devices with a compass also the heading (you won't be using that in this tutorial). Core Location can also give you continuous location updates while you're on the move.

Getting a location from Core Location is pretty easy but there are some pitfalls that you need to avoid. Let's start simple and add Core Location to the project and just ask it for the current coordinates and see what happens.

➤ Go to the Project Settings screen and scroll down to the section **Linked Frameworks and Libraries**. Click the **+** button and choose **CoreLocation.framework** from the list.



Adding CoreLocation.framework to the project

▸ Change the following in **CurrentLocationViewController.h**:

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

@interface CurrentLocationViewController : UIViewController
                                    <CLLocationManagerDelegate>
```

Core Location, as so many other parts of the iOS SDK, works with a delegate, so you've made the view controller conform to the `CLLocationManagerDelegate` protocol. In order to let your code know about this protocol, you have to `#import` the Core Location header.

▸ In **CurrentLocationViewController.m**, add an instance variable:

```
@implementation CurrentLocationViewController
{
  CLLocationManager *_locationManager;
}
```

The `CLLocationManager` is the object that will give you the GPS coordinates. You'll create this object in the view controller's `initWithCoder:` method.

▸ Add the `initWithCoder:` method to **CurrentLocationViewController.m**:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    _locationManager = [[CLLocationManager alloc] init];
```

```
  }
  return self;
}
```

The new `CLLocationManager` object doesn't give out GPS coordinates right away. To begin receiving coordinates, you have to call the `startUpdatingLocation` method first.

Unless you're doing turn-by-turn navigation, you don't want your app to continuously receive GPS coordinates. That requires a lot of power and will quickly drain the battery. For this app, you only turn on the location manager when you want a location fix and turn it off again when you've received a usable location.

You'll implement that logic in a minute (it's more complex than you think it would be). For now, you're just interested in receiving something from Core Location, just so you know that it works.

➤ Change the `getLocation:` method to the following:

```
- (IBAction)getLocation:(id)sender
{
  _locationManager.delegate = self;
  _locationManager.desiredAccuracy =
                            kCLLocationAccuracyNearestTenMeters;
  [_locationManager startUpdatingLocation];
}
```

This method is hooked up to the Get My Location button. It tells the location manager that the view controller is its delegate and that you want locations with an accuracy of up to ten meters. Then you start the location manager and from that moment on it will send location updates to the delegate, i.e. the view controller.

➤ Speaking of the delegate, add the following code to the bottom of the file:

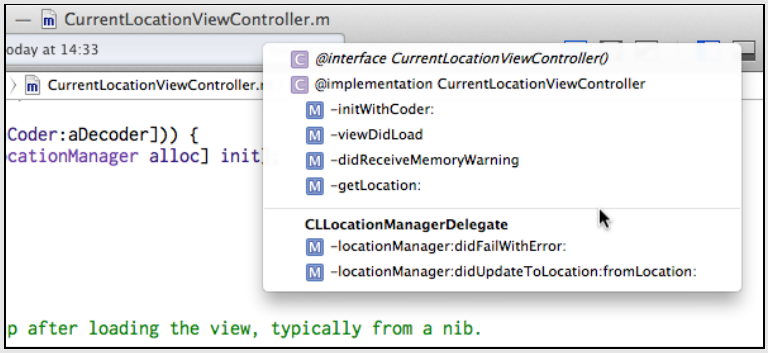
```
#pragma mark - CLLocationManagerDelegate

- (void)locationManager:(CLLocationManager *)manager
       didFailWithError:(NSError *)error
{
  NSLog(@"didFailWithError %@", error);
}

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

  NSLog(@"didUpdateLocations %@", newLocation);
```

```
}
```

These are the delegate methods for the location manager. For the time being, you'll simply write an `NSLog()` message to the debug area.

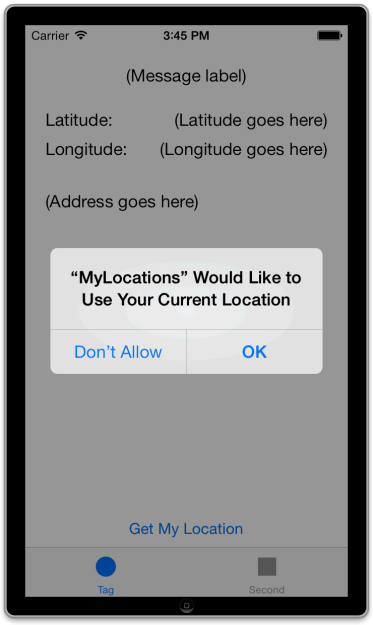
Pragma mark

In the code above there is a line that starts with `#pragma mark`. You may have seen lines like these before in your source code. A **pragma** is a special instruction to the compiler on how it should treat your source file. They're not often used, except for `#pragma mark`. This gives a hint to Xcode that you have organized your source file into neat sections. You can see this from the Jump Bar:



➤ Run the app in the Simulator and press the Get My Location button.

The first time you run an app that requests access to the location manager, Core Location will pop up the following alert:



Users have to allow your app to use their location

If a user denies the request with the Don't Allow button, then Core Location will never give your app location coordinates.

› Press the **Don't Allow** button.

Xcode's debug area should now show the following message:

```
MyLocations[3063:207] didFailWithError Error Domain=kCLErrorDomain
Code=1 "The operation couldn't be completed. (kCLErrorDomain error 1.)"
```

This comes from the `locationManager:didFailWithError:` delegate method. It's telling you that the location manager wasn't able to obtain a location. The reason why is described by an `NSError` object, which is the standard object that the iOS SDK uses to convey error information. You'll see it in many other places in the SDK (there are plenty of places where things can go wrong!).

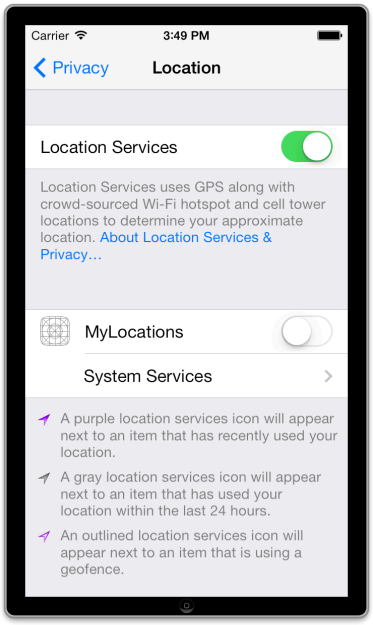
An `NSError` has a "domain" and a "code". The domain in this case is `kCLErrorDomain` meaning the error came from Core Location (CL). The code is 1 (also known by the symbolic name `kCLErrorDenied`), which means the user did not allow the app to obtain location information.

› Stop the app from within Xcode and run it again.

When you now press the Get My Location button, the app does not show the alert but immediately gives you the same error message.

Fortunately, users can change their minds and enable location services for your app again. This is done from the iPhone's Settings app.

› Go to the **Settings** app in the Simulator go to **Privacy** → **Location**.



Location Services in the Settings app

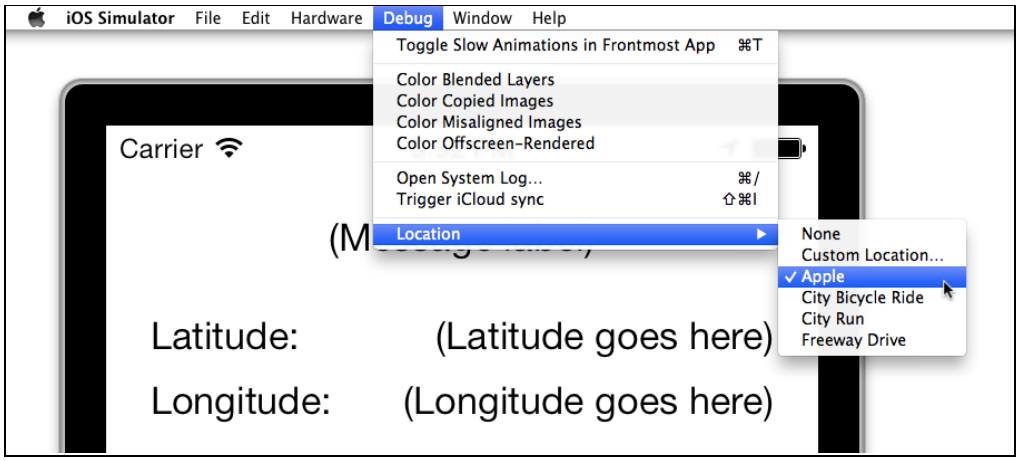
› Click the switch next to MyLocations to enable location services again and switch back to the app (or run it again from within Xcode). Press the Get My Location button.

When I tried this, the following message appeared in Xcode's debug area:

```
MyLocations[3016:207] didFailWithError Error Domain=kCLErrorDomain
Code=0 "The operation couldn't be completed. (kCLErrorDomain error 0.)"
```

Again there is an error message but with a different code, 0. This is "location unknown" which means Core Location was unable to obtain a location for some reason. That is not so strange, as you're running this from the Simulator, which obviously does not have a real GPS. Your Mac may have a way to obtain location information through Wi-Fi but this is not built into the Simulator. Fortunately, there is a way to fake it!

› With the app running, from the Simulator's menu bar at the top of the screen, choose **Debug** → **Location** → **Apple**.



The Simulator's Location menu

You now should see messages like these in the debug area:

```
MyLocations[3289:207] didUpdateLocations <+37.33259552,-122.03031802>
+/- 500.00m (speed -1.00 mps / course -1.00) @ 8/10/11 4:03:52 PM
Central European Summer Time

MyLocations[3289:207] didUpdateLocations <+37.33241023,-122.03051088>
+/- 65.00m (speed -1.00 mps / course -1.00) @ 8/10/11 4:03:54 PM Central
European Summer Time

MyLocations[3289:207] didUpdateLocations <+37.33233141,-122.03121860>
+/- 50.00m (speed -1.00 mps / course -1.00) @ 8/10/11 4:04:01 PM Central
European Summer Time

MyLocations[3289:207] didUpdateLocations <+37.33233141,-122.03121860>
+/- 30.00m (speed 0.00 mps / course -1.00) @ 8/10/11 4:04:03 PM Central
European Summer Time

MyLocations[3289:207] didUpdateLocations <+37.33233141,-122.03121860>
+/- 10.00m (speed 0.00 mps / course -1.00) @ 8/10/11 4:04:05 PM Central
European Summer Time
```

It keeps going on and on, giving the app a new location every second or so, although after a short while the latitude and longitude readings do not change anymore. These particular coordinates point at the Apple headquarters in Cupertino, California.

Look carefully at the coordinates the app is receiving. The first one says "+/- 500.00m", the second one "+/- 65.00m", the third "+/- 50.00m". This number keeps getting smaller and smaller until it stops at about "+/- 5.00m". This is the accuracy of the measurement, expressed in meters. What you see is the Simulator imitating what happens when you ask for a location on a real device.

If you go out with an iPhone and try to obtain location information, the iPhone uses three different ways to get your coordinates. It has onboard cell, Wi-Fi and GPS radios that each give it location information in more detail:

- Cell tower triangulation will always work if there is a signal but it's not very precise.
- Wi-Fi positioning works better but that is only available if there are known Wi-Fi routers nearby. This system uses a big database that contains the locations of wireless networking equipment.
- You get the best results from the GPS (Global Positioning System) but that attempts a satellite communication and is therefore the slowest of the three. It also won't work very well indoors.

So your device has several ways of obtaining location data, ranging from fast but inaccurate (cell towers, Wi-Fi) to accurate but slow (GPS). And none of these are guaranteed to work. Some devices don't even have a GPS or cell radio at all and have to rely on just the Wi-Fi. Suddenly obtaining a location seems a lot trickier.

Fortunately for us, Core Location does all of the hard work of turning the location readings from its various sources into a useful number. Instead of making you wait for the definitive results from the GPS (which may never come), Core Location sends location data to the app as soon as it gets it, and then follows up with more and more accurate readings.

Exercise. If you have an iPhone, iPod touch or iPad nearby, try the app on your device and see what kind of readings it gives you. If you have more than one device, try the app on all of them and note the differences. □

Asynchronous operations

Obtaining a location is an example of an **asynchronous** process.

Sometimes apps need to do things that may take a while. You start an operation and then you have to wait until it gives you the results. If you're unlucky, those results may never come at all! In the case of Core Location, it may take a second or two before you get the first location reading and then quite a few seconds more to get coordinates that are accurate enough for your app to use.

Asynchronous means that after you start such an operation, your app will continue on its merry way. The user interface is still responsive, new events are being sent and handled, and the user can still tap on things. The asynchronous process is said to be operating "in the background". As soon as the operation is done, the app is notified through a delegate so that it can process the results.

The opposite is **synchronous** (without the a). If you start a synchronous operation, the app won't continue until that operation is done. In the case of

`CLLocationManager` that would cause a big problem: your app would be totally unresponsive for the couple of seconds that it takes to get a location fix. Those kinds of "blocking" operations are often a bad experience for the user.

For example, MyLocations has a tab bar at the bottom. If the app would block while getting the location, then tapping on a tab during that time would have no effect. The user expects to always be able to switch between tabs but now it appears that the app is frozen, or worse, has crashed. The designers of iOS decided that such behavior is unacceptable and therefore operations that take longer than a fraction of a second should be performed in an asynchronous manner.

In the next tutorial you'll see more asynchronous processing in action when we talk about network connections and downloading stuff from the internet.

By the way, iOS has something called the "watchdog timer". If your app is unresponsive for too long, then under certain circumstances the watchdog timer may kill your app, so don't do that! Any operation that takes long enough to be noticeable by the user – even when you're just performing calculations – should be done asynchronously, in the background.

Putting the coordinates on the screen

The `didUpdateLocations` delegate method gives you an array of `CLLocation` objects that contain the current latitude and longitude coordinates of the user. (These objects also have some additional information, such as the altitude and speed, but you don't use those in this app.)

You'll take the last `CLLocation` object from the array – because that is the most recent update – and display its coordinates in the labels that you added to the screen earlier.

➤ Add a new instance variable, `_location`, to **CurrentLocationViewController.m**:

```
@implementation CurrentLocationViewController
{
  CLLocationManager *_locationManager;
  CLLocation *_location;
}
```

You store the user's current location in this variable.

➤ Change the `didUpdateLocations` method to:

```
- (void)locationManager:(CLLocationManager *)manager
     didUpdateLocations:(NSArray *)locations
{
  CLLocation *newLocation = [locations lastObject];
```

```
  NSLog(@"didUpdateLocations %@", newLocation);

  _location = newLocation;
  [self updateLabels];
}
```

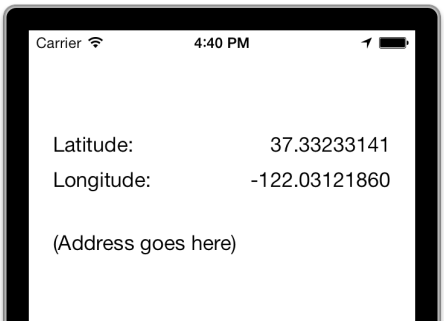
Keep the `NSLog()` in there because it's handy for debugging. You store the location object that you get from the location manager into the instance variable and call the new `updateLabels` method.

➤ Add the `updateLabels` method below:

```
- (void)updateLabels
{
  if (_location != nil) {
    self.latitudeLabel.text = [NSString stringWithFormat:
                        @"%.8f", _location.coordinate.latitude];
    self.longitudeLabel.text = [NSString stringWithFormat:
                       @"%.8f", _location.coordinate.longitude];
    self.tagButton.hidden = NO;
    self.messageLabel.text = @"";

  } else {
    self.latitudeLabel.text = @"";
    self.longitudeLabel.text = @"";
    self.addressLabel.text = @"";
    self.tagButton.hidden = YES;
    self.messageLabel.text = @"Press the Button to Start";
  }
}
```

If there is a location object (`_location` is not `nil`) then this converts the latitude and longitude, which are values with datatype `double`, into strings and put them into the labels. The `%.8f` format specifier in `stringWithFormat` does the same thing as the `%f` that you've seen earlier: it takes a decimal number and puts it in the string. The `.8` part means that there should always be 8 digits behind the decimal point.

➤ Run the app, select a location to simulate from the Simulator's **Debug** menu and tap the Get My Location button. You'll now see the latitude and longitude appear on the screen.



The app shows the GPS coordinates

When the app starts up it has no location object and therefore should show the "Press the Button to Start" message, but it doesn't do that yet. That's because you don't call `updateLabels` until the app receives coordinates. You should also call this method from `viewDidLoad`.

▸ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  [self updateLabels];
}
```

▸ Run the app. Initially, the screen should now say, "Press the Button to Start" and the latitude and longitude labels are empty.

Handling errors

Getting GPS coordinates is error-prone. You may be somewhere where there is no clear line-of-sight to the sky (such as inside or in an area with lots of tall buildings), blocking your GPS signal. There may not be many Wi-Fi routers around you, or they haven't been catalogued yet, so the Wi-Fi radio isn't much help getting a location fix either. And of course your cell signal is so weak that triangulating your position doesn't offer particularly good results either.

All of that is assuming your device actually has a GPS or cell radio. I just went out with my iPod touch to capture coordinates and make some pictures to show in this app. In the city center it was unable to obtain a location fix. My iPhone did better but it still wasn't ideal.

The moral of this story is that your location-aware apps better know how to deal with errors and bad readings. There are no guarantees that you'll be able to get a location fix, and if you do then it might still take a few seconds. This is where software meets the real world. You should add some error handling code to the app to let users know about problems getting those coordinates.

▸ Change the `didFailWithError` delegate method to the following:

```
- (void)locationManager:(CLLocationManager *)manager
       didFailWithError:(NSError *)error
{
  NSLog(@"didFailWithError %@", error);

  if (error.code == kCLErrorLocationUnknown) {
    return;
  }

  [self stopLocationManager];
  _lastLocationError = error;

  [self updateLabels];
}
```

The location manager may report a variety of errors. You can look at the `code` property of the `NSError` object to find out what type of error you're dealing with.

Some of the Core Location errors:

- `kCLErrorLocationUnknown` - The location is currently unknown, but Core Location will keep trying.
- `kCLErrorDenied` - The user declined the app to use location services.
- `kCLErrorNetwork` - There was a network-related error.

There are more (having to do with the compass and geocoding), but you get the point.

> What's with the weird `kCLErrorXXX` notation? These error codes are simple integer values but rather than using the values 0, 1, 2 and so on in your program, Core Location has given them symbolic names. That makes these codes easier to understand and you're less likely to pick the wrong one.
>
> The `k` prefix is often used to signify that a name represents a **constant** value (I guess whoever came up with this prefix thought it was spelled "konstant"). `CL`, of course, stands for Core Location.

In your updated `didFailWithError`, you do:

```
if (error.code == kCLErrorLocationUnknown) {
  return;
}
```

The `kCLErrorLocationUnknown` error means the location manager was unable to obtain a location right now, but that doesn't mean all is lost. It might just need another second or so to get an uplink to the GPS satellite. In the mean time it's

letting you know that for now it could not get any location information. When you get this error, you will simply keep trying until you do find a location or receive a more serious error.

In the case of such a more serious error, the method does the following:

```
  [self stopLocationManager];
  _lastLocationError = error;

  [self updateLabels];
```

You've seen `updateLabels`. You'll be extending that method in a second to show the error to the user because you don't want to leave them in the dark about such things. To this end, you store the error object into a new instance variable, `_lastLocationError`. That way, you can look up later what kind of error you were dealing with.

The `stopLocationManager` method is new. To conserve battery power the app really should power down the iPhone's radios as soon as it doesn't need them anymore. If obtaining a location appears to be impossible for wherever the user currently is, then you'll tell the location manager to stop.

If this was a turn-by-turn navigation app, you'd keep the location manager running even in the case of a network error, because who knows, a couple of meters ahead you might get a valid location. For this app the user will simply have to press the Get My Location button again if they want to try in another spot.

➤ Add the `stopLocationManager` method below `updateLabels`:

```
- (void)stopLocationManager
{
  if (_updatingLocation) {
    [_locationManager stopUpdatingLocation];
    _locationManager.delegate = nil;
    _updatingLocation = NO;
  }
}
```

There's an if-statement in here that checks whether the boolean instance variable `_updatingLocation` is `YES` or `NO`. If it is `NO`, then the location manager wasn't currently active and there's no need to stop it. The reason for having this `_updatingLocation` variable is that you are going to change the appearance of the Get My Location button and the status message label when the app is trying to obtain a location fix, to let the user know the app is working on it.

Let's add the missing bits and then try the app again.

➤ Add these two instance variables:

```
BOOL _updatingLocation;
NSError *_lastLocationError;
```

▸ And put some extra code in `updateLabels`:

```
- (void)updateLabels
{
  if (_location != nil) {
    . . .

  } else {
    self.latitudeLabel.text = @"";
    self.longitudeLabel.text = @"";
    self.addressLabel.text = @"";
    self.tagButton.hidden = YES;

    NSString *statusMessage;
    if (_lastLocationError != nil) {
      if ([_lastLocationError.domain isEqualToString:
                                              kCLErrorDomain] &&
          _lastLocationError.code == kCLErrorDenied) {
        statusMessage = @"Location Services Disabled";
      } else {
        statusMessage = @"Error Getting Location";
      }
    } else if (![CLLocationManager locationServicesEnabled]) {
      statusMessage = @"Location Services Disabled";
    } else if (_updatingLocation) {
      statusMessage = @"Searching...";
    } else {
      statusMessage = @"Press the Button to Start";
    }

    self.messageLabel.text = statusMessage;
  }
}
```

The new bit of code determines what to put in the `messageLabel` at the top of the screen. It uses a bunch of if-statements to figure out what the current status of the app is.

If the location manager gave an error, the label will show an error message. The first error it checks for is `kCLErrorDenied` (in the error domain `kCLErrorDomain`, which means Core Location errors). In that case the user has not given this app permission to use the location services. That sort of defeats the purpose of this app but it can happen and you have to check for it anyway. If the error code is

something else then you simply say "Error Getting Location" as this usually means there was no way of obtaining a location fix.

Even if there was no error then it might still be impossible to get location coordinates if the user disabled Location Services completely on his device (instead of just for this app). You check for that situation with the `locationServicesEnabled` method of `CLLocationManager`.

Suppose there were no errors and everything works fine, then the status label will say "Searching..." before the first location object has been received. If your device can obtain the location fix quickly then this text will be visible only for a fraction of a second, but often it might take a short while to get that first location fix so it's nice to let the user know that the app is actively looking up his location. That is what you're using the `_updatingLocation` boolean for.

You put all this logic into a single method because that makes it easy to change the screen when something has changed. Received a location? Simply call `updateLabels` to refresh the contents of the screen. Received an error? Let `updateLabels` sort it out...

➤ Add the `startLocationManager` method right above `stopLocationManager`:

```
- (void)startLocationManager
{
  if ([CLLocationManager locationServicesEnabled]) {
    _locationManager.delegate = self;
    _locationManager.desiredAccuracy =
                            kCLLocationAccuracyNearestTenMeters;
    [_locationManager startUpdatingLocation];
    _updatingLocation = YES;
  }
}
```

Starting the location manager used to happen in the `getLocation` action. Because you now have a `stopLocationManager` method, it makes sense to move that code into a method of its own, `startLocationManager`, just to keep things symmetrical. The only difference with before is that this checks whether the location services are enabled. You also set the variable `_updatingLocation` to `YES`.

➤ Change the `getLocation` method to:

```
- (IBAction)getLocation:(id)sender
{
  [self startLocationManager];
  [self updateLabels];
}
```

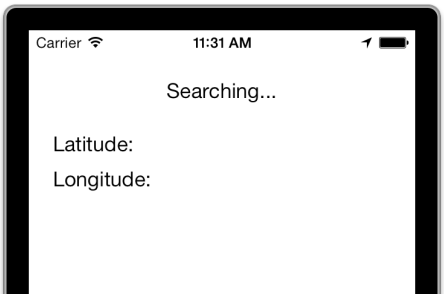
There is one more small change to make. Suppose there was an error and no location could be obtained, but then you walk around for a bit and a valid location comes in. In that case it's a good idea to wipe the old error code.

➤ Change `didUpdateLocations` to:

```
- (void)locationManager:(CLLocationManager *)manager
     didUpdateLocations:(NSArray *)locations
{
  CLLocation *newLocation = [locations lastObject];

  NSLog(@"didUpdateLocations %@", newLocation);

  _lastLocationError = nil;
  _location = newLocation;
  [self updateLabels];
}
```

The only new line here is `_lastLocationError = nil;` to clear out the old error state. If you receive a valid coordinate, then whatever previous error you may have encountered is no longer applicable.

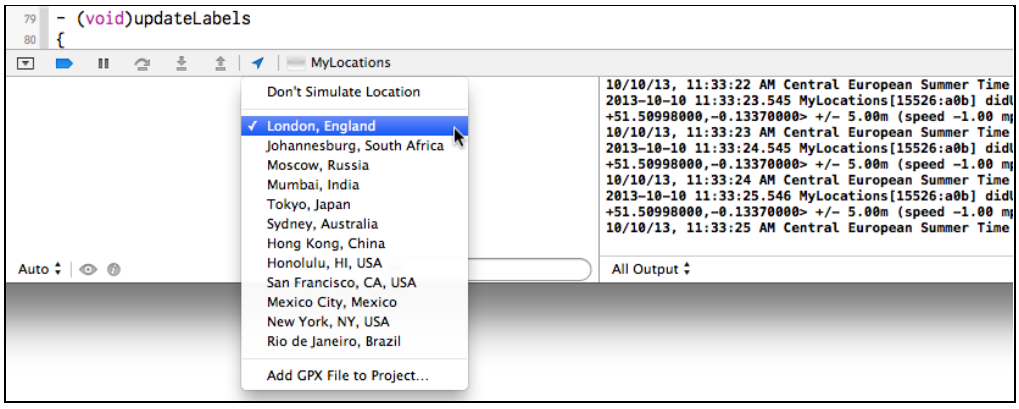
➤ Run the app. While the app is waiting for incoming coordinates, the label at the top should say "Searching..." until it finds a valid coordinate or encounters a fatal error.



The app is waiting to receive GPS coordinates

Play with the Simulator's location settings. Note that setting the Simulator's location to None isn't actually an error anymore. This still returns the `kCLErrorLocationUnknown` error code but you ignore that because it's not a fatal error.

Tip: You can also simulate locations from within Xcode. If your app uses Core Location, the bar at the top of the debug area gets an arrow icon. Click on that icon to change the simulated location:



Simulating locations from within the Xcode debugger

Ideally you should not just test in the Simulator but also on your device, as you're more likely to encounter real errors that way.

Improving the results

Cool, you know how to obtain a `CLLocation` object from Core Location and you're able to handle errors. Now what?

Well, here's the thing: you saw in the Simulator that Core Location keeps giving you new location objects over and over, even though the coordinates may not have changed. That's because the user could be moving and then their coordinates do change. However, you're not building a navigation app so for the purposes of this app you just want to get a location that is accurate enough and then you can tell the location manager to stop sending updates.

This is important because getting location updates costs a lot of battery power as the device needs to keep its GPS/Wi-Fi/cell radios powered up for this. This app doesn't need to ask for GPS coordinates all the time, so it should stop when the location is accurate enough.

The problem is that you can't always get the accuracy you want, so you have to detect this. When the last couple of coordinates you received aren't increasing in accuracy then this is probably as good as it's going to get and you should let the radio power down.

▸ Change `didUpdateLocations` to the following:

```
- (void)locationManager:(CLLocationManager *)manager
     didUpdateLocations:(NSArray *)locations
{
  CLLocation *newLocation = [locations lastObject];

  NSLog(@"didUpdateLocations %@", newLocation);

  if ([newLocation.timestamp timeIntervalSinceNow] < -5.0) {
```

```
    return;
  }

  if (newLocation.horizontalAccuracy < 0) {
    return;
  }

  if (_location == nil || _location.horizontalAccuracy >
                              newLocation.horizontalAccuracy) {

    _lastLocationError = nil;
    _location = newLocation;
    [self updateLabels];

    if (newLocation.horizontalAccuracy <=
                            _locationManager.desiredAccuracy) {
      NSLog(@"*** We're done!");
      [self stopLocationManager];
    }
  }
}
```

Let's take these changes one-by-one:

```
if ([newLocation.timestamp timeIntervalSinceNow] < -5.0) {
  return;
}
```

If the time at which the location object was determined is too long ago (5 seconds in this case), then this is a so-called *cached* result. Instead of returning a new location fix, the location manager may initially give you the most recently found location under the assumption that you might not have moved much since last time (obviously this does not take into consideration people with jet packs). You'll simply ignore these cached locations if they are too old.

```
if (newLocation.horizontalAccuracy < 0) {
  return;
}
```

You're going to be using the `horizontalAccuracy` property of the location to determine whether new readings are more accurate than previous ones. However, sometimes locations may have a `horizontalAccuracy` that is less than 0, in which case these measurements are invalid and you should ignore them.

```
if (_location == nil || _location.horizontalAccuracy >
                              newLocation.horizontalAccuracy) {
```

This is where you determine if the new reading is more useful than the previous one. Generally speaking, Core Location starts out with a fairly inaccurate reading and then gives you more and more accurate ones as time passes. However, there are no guarantees here so you cannot assume that the next reading truly is always more accurate.

Note that a larger accuracy value actually means *less* accurate – after all, accurate up to 100 meters is worse than accurate up to 10 meters. That's why you check whether the previous reading, `_location.horizontalAccuracy`, is greater than the new reading, `newLocation.horizontalAccuracy`.

You also check for `_location == nil`. Recall that `_location` is the instance variable that stores the `CLLocation` object that you obtained in a previous call to `didUpdateLocations`. If `_location` is `nil` then that means this is the very first location update you're receiving and in that case you should also continue. The `||` operator (logical or) tests whether either of these conditions is true. If so, you go on:

```
    _lastLocationError = nil;
    _location = newLocation;
    [self updateLabels];
```

You've seen this before. It clears out any previous error if there was one and stores the new location object into the `_location` variable.

```
if (newLocation.horizontalAccuracy <=
                              _locationManager.desiredAccuracy) {
  NSLog(@"*** We're done!");
  [self stopLocationManager];
}
```

If the new location's accuracy is equal to or better than the desired accuracy then you call it a day and you stop asking the location manager for updates. When you started the location manager in `startLocationManager`, you set the desired accuracy to 10 meters (`kCLLocationAccuracyNearestTenMeters`), which is good enough for this app.

➤ Run the app. First set the Simulator's location to None, then press Get My Location. The screen now says "Searching..." Switch to location Apple. After a brief moment, the screen is updated with GPS coordinates as they come in. If you follow along in the debug area, you'll get about 10 location updates before it says "*** We're done!" and the location updates stop.

You as the developer can tell from the debug area when the location updates stop, but obviously the user won't see this. The Tag Location button becomes visible as soon as the first location is received so the user can start tagging right away, but this location may not be accurate enough. So it's nice to show the user when the app has found the most accurate location.

To make this clearer, you are going to toggle the Get My Location button to say "Stop" when the location grabbing is active and go back to "Get My Location" when it's done. That gives a nice visual clue to the user. Later in this lesson you'll also show an animated activity spinner that makes this even more obvious.

To change the state of this button, you'll add a `configureGetButton` method.

▸ Add the following method below `updateLabels`:

```
- (void)configureGetButton
{
  if (_updatingLocation) {
    [self.getButton setTitle:@"Stop"
                    forState:UIControlStateNormal];
  } else {
    [self.getButton setTitle:@"Get My Location"
                    forState:UIControlStateNormal];
  }
}
```

It's quite simple: if the app is currently updating the location then the button's title becomes Stop, otherwise it is Get My Location.

▸ Call the `configureGetButton` method from the following places:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  [self updateLabels];
  [self configureGetButton];
}
```

```
- (IBAction)getLocation:(id)sender
{
  [self startLocationManager];
  [self updateLabels];
  [self configureGetButton];
}
```

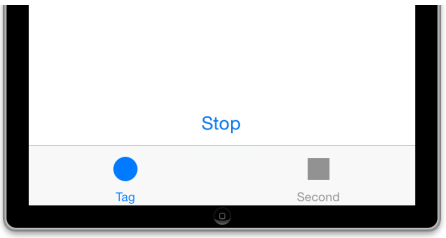
```
- (void)locationManager:(CLLocationManager *)manager
       didFailWithError:(NSError *)error
{
  . . .
```

```
  [self updateLabels];
  [self configureGetButton];
}
```

```
- (void)locationManager:(CLLocationManager *)manager
     didUpdateLocations:(NSArray *)locations
{
    . . .

    if (newLocation.horizontalAccuracy <=
                              _locationManager.desiredAccuracy) {
      NSLog(@"*** We're done!");
      [self stopLocationManager];
      [self configureGetButton];
    }
  }
}
```

Basically anywhere you did `updateLabels`, you're now also calling `configureGetButton`. In `didUpdateLocations` you only call it when you're done.

➤ Run the app again and perform the same test. The button changes to Stop when you press it and when there are no more location updates, it switches back.



The stop button

When a button says Stop you expect to be able to press it so you can interrupt the location updates. This is especially so when you're not getting any coordinates. Eventually Core Location may give an error but as a user you may not want to wait for that. Currently, however, pressing Stop doesn't stop anything. You have to change `getLocation` for this as taps on the button call this method.

➤ Change `getLocation` to the following:

```
- (IBAction)getLocation:(id)sender
{
  if (_updatingLocation) {
```

```
    [self stopLocationManager];
  } else {
    _location = nil;
    _lastLocationError = nil;

    [self startLocationManager];
  }

  [self updateLabels];
  [self configureGetButton];
}
```

Again you're using the `_updatingLocation` flag to determine what state the app is in. If the button is pressed while the app is already doing the location fetching, you stop the location manager. Note that you also clear out the old location and error objects before you start looking for a new location.

➤ Run the app. Now pressing the Stop button will put an end to the location updates. You should see no more updates in the debug area after you press Stop.

Reverse geocoding

The GPS coordinates you've dealt with so far are just numbers. The coordinates 37.33240904, -122.03051218 don't really mean that much, but the address 1 Infinite Loop in Cupertino, California does.

Using a process known as **reverse geocoding**, you can turn that set of coordinates into a human-readable address. (Of course, regular or "forward" geocoding does the opposite: it turns an address into GPS coordinates. You can do both with the iOS SDK, but in this tutorial you only do the reverse one.)

> **Note:** Previously, reverse geocoding was done with the `MKReverseGeocoder` object but as of iOS 5 that is *deprecated*, which means it will still work but you really shouldn't use it anymore in new projects. When old objects are replaced by newer ones, they're usually kept around for a few OS releases just so that older apps can still continue to function. New apps should really move on to the new API, though. That new API is `CLGeocoder`.

You'll use the `CLGeocoder` object to turn the location data into a human-readable address and then display that address on the screen in the `addressLabel` outlet. It's quite easy to do this but there are some rules. You're not supposed to send out a ton of these reverse geocoding requests at the same time. The process of reverse geocoding takes place on a server hosted by Apple and it costs them bandwidth and processor time to handle these requests. If you flood these servers with requests, Apple won't be happy.

The MyLocations app is only supposed to be used occasionally, so its users won't be spamming the Apple servers but you should still limit the geocoding requests to one at a time, and once for every unique location. It makes no sense to reverse geocode the same set of coordinates over and over. Reverse geocoding needs an internet connection and anything you can do to prevent unnecessary use of the iPhone's radios is a good thing for your users.

▸ Add the following instance variables to **CurrentLocationViewController.m**:

```
CLGeocoder *_geocoder;
CLPlacemark *_placemark;
BOOL _performingReverseGeocoding;
NSError *_lastGeocodingError;
```

These mirror what you did for the location manager. `CLGeocoder` is the object that will perform the geocoding and `CLPlacemark` is the object that contains the address results. You set `_performingReverseGeocoding` to `YES` when a geocoding operation is taking place, and `_lastGeocodingError` will contain an `NSError` object if something went wrong.

▸ Create the geocoder object in `initWithCoder:`

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    _locationManager = [[CLLocationManager alloc] init];
    _geocoder = [[CLGeocoder alloc] init];
  }
  return self;
}
```

▸ Put the geocoder to work in `didUpdateLocations`:

```
- (void)locationManager:(CLLocationManager *)manager
     didUpdateLocations:(NSArray *)locations
{
  . . .

  if (_location == nil || _location.horizontalAccuracy >
                                newLocation.horizontalAccuracy) {

    if (newLocation.horizontalAccuracy <=
                              _locationManager.desiredAccuracy) {
      . . .
    }

    if (!_performingReverseGeocoding) {
```

```
      NSLog(@"*** Going to geocode");

      _performingReverseGeocoding = YES;

      [_geocoder reverseGeocodeLocation:_location
                      completionHandler:
        ^(NSArray *placemarks, NSError *error) {

          NSLog(@"*** Found placemarks: %@, error: %@",
                                              placemarks, error);

          _lastGeocodingError = error;
          if (error == nil && [placemarks count] > 0) {
            _placemark = [placemarks lastObject];
          } else {
            _placemark = nil;
          }

          _performingReverseGeocoding = NO;
          [self updateLabels];
        }];
    }
  }
}
```

Whoa, what is that!? Unlike the location manager, `CLGeocoder` does not use a delegate to tell you about the result, but something called a **block**. Blocks are a relatively new addition to the Objective-C language (since iOS 4) and they make it a lot easier to write certain bits of code. Many of the new APIs that were introduced after iOS 4 use blocks instead of delegates.

The problem with using a delegate to provide feedback is that you need to write one or more separate methods. For `CLLocationManager` those are the `didUpdateLocations` and `didFailWithError` methods. By making different methods you move the code that deals with the response away from the code that makes the request. With blocks, on the other hand, you can put that handling code inline.

This is what it looks like without the code:

```
[_geocoder reverseGeocodeLocation:_location
    completionHandler:^(NSArray *placemarks, NSError *error) {

    // put your statements here

  }];
```

This tells the `CLGeocoder` object that you want to reverse geocode the location, and that the code in the block following `completionHandler` should be executed as soon as the geocoding is completed.

The block itself is:

```
^(NSArray *placemarks, NSError *error) {

    // put your statements here

  }];
```

You can tell this is a block because it starts with a caret ^ symbol. The variables between the `( )` parentheses are the parameters for this block and they work just like parameters for a method or a function. When the geocoder finds a result for the location object that you gave it (or encounters an error), it invokes the block and executes the statements within. The `placemarks` variable will contain an array of `CLPlacemark` objects that describe the address information, and the `error` variable contains an error message in case something went wrong.

That means the statements in the block are *not* executed right away when `didUpdateLocations` is called. Instead, the block and everything inside it is given to `CLGeocoder`, which keeps it until later when it has performed the reverse geocoding operation. Only then will it execute the code from the block. In a sense you're defining a delegate method inline.

It's OK if blocks don't make any sense to you right now. If you're coming from another language you may recognize blocks as being *closures*. In the next tutorial you'll use blocks a couple more times.

Back to the reverse geocoder. I said that the app should only perform a single request at a time, so first you check whether it is not busy yet:

```
if (!_performingReverseGeocoding) {
  NSLog(@"*** Going to geocode");
```

Then you start the geocoder and give it the block. Inside the block, the first thing you do is an `NSLog()` just so you can see what is going on.

```
  _performingReverseGeocoding = YES;

  [_geocoder reverseGeocodeLocation:_location
      completionHandler:^(NSArray *placemarks, NSError *error) {

      NSLog(@"*** Found placemarks: %@, error: %@",
                                          placemarks, error);
```

Just as with the location manager, you store the error object so you can refer to it later, although you use a different instance variable for this.

```
      _lastGeocodingError = error;
      if (error == nil && [placemarks count] > 0) {
        _placemark = [placemarks lastObject];
      } else {
        _placemark = nil;
      }

      _performingReverseGeocoding = NO;
      [self updateLabels];
    }];
```

If there is no error and there are objects inside the `placemarks` array, then you take the last one. Usually there will be only one `CLPlacemark` object in the array but there is the odd situation where one location coordinate may refer to more than one address. This app can only handle one address, so you'll just pick the last one (which usually is the only one).

Note that you're doing a bit of **defensive programming** here: you specifically check first whether the array has any objects in it. If there is no error then it should, but you're not going to trust that it always will.

If there was an error, you set `_placemark` to `nil`. Note that you did not do that for the locations. If there was an error there, you kept the previous location object because it may actually be correct (or good enough) and it's better than nothing. But for the address that makes less sense. You don't want to show an old address, only the address that corresponds to the current location or no address at all.

In mobile development, nothing is guaranteed. You may get coordinates back or you may not, and if you do, they may not be very accurate. The reverse geocoding will probably succeed if there is some type of network connection available, but you also need to be prepared to handle the case where there is none. And not all GPS coordinates correspond to actual street addresses (there is no corner of 52nd and Broadway in the Sahara desert).

➤ Run the app and pick a location. As soon as the first location is found, you can see in the debug area that the reverse geocoder kicks in. If you choose the Apple location you'll see that some location readings are duplicates; the geocoder only does the first of those. Only when the accuracy of the reading improves does the app reverse geocode again.

```
MyLocations[16278:a0b] didUpdateLocations <+37.33233141,-122.03121860>
+/- 50.00m (speed 0.00 mps / course -1.00) @ 10/10/13, 1:31:37 PM
Central European Summer Time

MyLocations[16278:a0b] didUpdateLocations <+37.33233141,-122.03121860>
```

```
+/- 30.00m (speed 0.00 mps / course -1.00) @ 10/10/13, 1:31:38 PM
Central European Summer Time

MyLocations[16278:a0b] *** Going to geocode

MyLocations[16278:a0b] *** Found placemarks: (
    "Apple Inc., Apple Inc., 1\U20135 Infinite Loop, Cupertino, CA
95014-2083, United States @ <+37.33231290,-122.03073410> +/- 100.00m"
), error: (null)
```

Let's make the address visible to the user as well.

➤ Change `updateLabels` to:

```
- (void)updateLabels
{
  if (_location != nil) {
    . . .

    if (_placemark != nil) {
      self.addressLabel.text = [self stringFromPlacemark:
                                                    _placemark];
    } else if (_performingReverseGeocoding) {
      self.addressLabel.text = @"Searching for Address...";
    } else if (_lastGeocodingError != nil) {
      self.addressLabel.text = @"Error Finding Address";
    } else {
      self.addressLabel.text = @"No Address Found";
    }

  } else {
    . . .
  }
}
```

Because you only do the address lookup once the app has a location, you just have to change the code inside the first `if`. If you've found an address, you show that to the user, otherwise you show a status message. The code to format the `CLPlacemark` object into a string is placed in its own method, just to keep the code readable.

➤ Add the `stringFromPlacemark:` method above `updateLabels`:

```
- (NSString *)stringFromPlacemark:(CLPlacemark *)thePlacemark
{
  return [NSString stringWithFormat:@"%@ %@\n%@ %@ %@",
    thePlacemark.subThoroughfare, thePlacemark.thoroughfare,
```

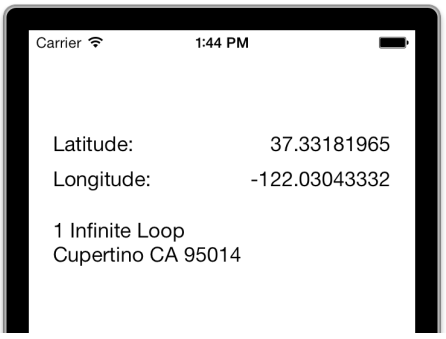
```
    thePlacemark.locality, thePlacemark.administrativeArea,
    thePlacemark.postalCode];
}
```

Just so you know, `subThoroughfare` is the house number, `thoroughfare` is the street name, `locality` is the city, `administrativeArea` is the state or province, and `postalCode` is the zip code or postal code.

▸ In `getLocation`, clear out the `_placemark` and `_lastGeocodingError` variables in order to start with a clean slate:

```
- (IBAction)getLocation:(id)sender
{
  if (_updatingLocation) {
    [self stopLocationManager];
  } else {
    _location = nil;
    _lastLocationError = nil;
    _placemark = nil;
    _lastGeocodingError = nil;

    [self startLocationManager];
  }

  [self updateLabels];
  [self configureGetButton];
}
```

▸ Run the app again. Now you'll see that seconds after a location is found, the address label is filled in as well.



Reverse geocoding finds the address for the GPS coordinates

Note: For some locations the address label can say "(null)". This is because the `CLPlacemark` object may contain incomplete information, for example a

missing street number. Later in this tutorial you'll improve how addresses are displayed.

Exercise. If you pick the City Bicycle Ride or City Run locations from the Simulator's Debug menu, you should see in the debug area that the app jumps through a whole bunch of different coordinates (it simulates someone moving from one place to another). However, the coordinates on the screen and the address label don't change nearly as often. Why is that? □

Answer: The logic in the MyLocations app was designed to find the most accurate set of coordinates for a stationary position. You only update the `_location` variable when a new set of coordinates comes in that is more accurate than previous readings. Any new readings with a higher – or the same – `horizontalAccuracy` value are simply ignored, regardless of what the actual coordinates are.

With the City Bicycle Ride and City Run options the app doesn't receive the same coordinates with increasing accuracy but a series of completely different coordinates. That means this app doesn't work very well when you're on the move (unless you press Stop and try again), but that's also not what it was intended for.

Testing in practice

When I first wrote this source code I had only tested it out on the Simulator and there it worked fine. Then I put it on my iPod touch and guess what, not so good. The problem with the iPod touch is that it doesn't have a GPS so it relies on Wi-Fi only to determine the location. But Wi-Fi might not be able to give you accuracy up to ten meters; I got +/- 100 meters at best.

Right now, you only stop the location updates when the accuracy of the reading falls within the `desiredAccuracy` setting – something that will never happen on this iPod. That goes to show that you can't always rely on the Simulator to test your apps. You need to put them on your device and test them in the wild, especially when using location-based APIs. If you have more than one device, then test on all of them.

In order to deal with this situation, you will improve the `didUpdateLocations` method some more.

▸ Change `didUpdateLocations` to:

```
- (void)locationManager:(CLLocationManager *)manager
     didUpdateLocations:(NSArray *)locations
{
  CLLocation *newLocation = [locations lastObject];

  NSLog(@"didUpdateLocations %@", newLocation);

  if ([newLocation.timestamp timeIntervalSinceNow] < -5.0) {
```

```
    return;
  }

  if (newLocation.horizontalAccuracy < 0) {
    return;
  }

  CLLocationDistance distance = MAXFLOAT;
  if (_location != nil) {
    distance = [newLocation distanceFromLocation:_location];
  }

  if (_location == nil || _location.horizontalAccuracy >
                                newLocation.horizontalAccuracy) {

    _lastLocationError = nil;
    _location = newLocation;
    [self updateLabels];

    if (newLocation.horizontalAccuracy <=
                              _locationManager.desiredAccuracy) {
      NSLog(@"*** We're done!");
      [self stopLocationManager];
      [self configureGetButton];

      if (distance > 0) {
        _performingReverseGeocoding = NO;
      }
    }

    if (!_performingReverseGeocoding) {
      NSLog(@"*** Going to geocode");

      _performingReverseGeocoding = YES;
      [_geocoder reverseGeocodeLocation:_location
                  completionHandler:
                        ^(NSArray *placemarks, NSError *error) {

        NSLog(@"*** Found placemarks: %@, error: %@",
                                             placemarks, error);

        _lastGeocodingError = error;
        if (error == nil && [placemarks count] > 0) {
          _placemark = [placemarks lastObject];
        } else {
```

```
            _placemark = nil;
          }

          _performingReverseGeocoding = NO;
          [self updateLabels];
        }];
      }

  } else if (distance < 1.0) {
    NSTimeInterval timeInterval = [newLocation.timestamp
                    timeIntervalSinceDate:_location.timestamp];
    if (timeInterval > 10) {
      NSLog(@"*** Force done!");
      [self stopLocationManager];
      [self updateLabels];
      [self configureGetButton];
    }
  }
}
```

It's a pretty long method now, but only the three highlighted bits were added. This is the first one:

```
CLLocationDistance distance = MAXFLOAT;
if (_location != nil) {
  distance = [newLocation distanceFromLocation:_location];
}
```

This calculates the distance between the new reading and the previous reading, if there was one. If there was no previous reading, then the distance is `MAXFLOAT`. That is a built-in constant that represents the maximum value that a floating-point number can have. This little trick gives it a gigantic distance if this is the very first reading. You're doing that so any of the following calculations still work even if you weren't able to calculate a true distance yet.

You also added an `if` in the part where you determine whether the desired accuracy has been reached. This happens after you stop the location manager:

```
if (distance > 0) {
  _performingReverseGeocoding = NO;
}
```

This forces a reverse geocoding even if the app is already currently performing another geocoding request. Of course, if `distance` is 0, then this location is the same as the location from a previous reading and you don't need to reverse geocode it anymore.

This is done because you absolutely want the address for that final location, as that is the most accurate location you've found. But if some previous location was still being reverse geocoded, that step would normally be skipped. Simply by setting `_performingReverseGeocoding` to `NO`, you always force the geocoding to be done for this final coordinate.

The real improvement is found at the bottom of the method:

```
} else if (distance < 1.0) {
  NSTimeInterval timeInterval = [newLocation.timestamp
                    timeIntervalSinceDate:_location.timestamp];
  if (timeInterval > 10) {
    NSLog(@"*** Force done!");
    [self stopLocationManager];
    [self updateLabels];
    [self configureGetButton];
  }
}
```

If the coordinate from this reading is not significantly different from the previous reading and it has been more than 10 seconds since you've received that original reading, then it's a good point to hang up your hat and stop. It's safe to assume you're not going to get a better coordinate than this and you stop fetching the location.

This is the improvement that was necessary to make my iPod touch stop. It wouldn't give me a location with better accuracy than +/- 100 meters but it kept repeating the same one over and over. I picked a time limit of 10 seconds because that seemed to give good results.

Note that you don't just say:

```
} else if (distance = 0) {
```

The distance between subsequent readings is never exactly 0. It may be something like 0.0017632. Rather than checking for equals to 0, it's better to check for less than a certain distance, in this case one meter.

➤ Run the app and test that everything still works. It may be hard to recreate this situation on the Simulator but try it on your iPod touch or iPhone inside the house and see what the `NSLog`s output to the debug area.

There is another improvement you can make to increase the robustness of this logic and that is to set a time-out on the whole thing. You can tell iOS to perform a method one minute from now. If by that time the app hasn't found a location yet, you stop the location manager and show an error message.

➤ Change `startLocationManager` to:

```
- (void)startLocationManager
{
  if ([CLLocationManager locationServicesEnabled]) {
    _locationManager.delegate = self;
    _locationManager.desiredAccuracy =
                          kCLLocationAccuracyNearestTenMeters;
    [_locationManager startUpdatingLocation];
    _updatingLocation = YES;

    [self performSelector:@selector(didTimeOut:)
               withObject:nil afterDelay:60];
  }
}
```

The new line schedules the operating system to send the `didTimeOut:` message to `self` after 60 seconds. `didTimeOut:` is of course the name of a method that you have to provide. A **selector** is the name of a method and the `@selector()` syntax is how you refer to a method by name in Objective-C.

▸ Change `stopLocationManager` to:

```
- (void)stopLocationManager
{
  if (_updatingLocation) {
    [NSObject cancelPreviousPerformRequestsWithTarget:self
              selector:@selector(didTimeOut:) object:nil];

    [_locationManager stopUpdatingLocation];
    _locationManager.delegate = nil;
    _updatingLocation = NO;
  }
}
```

Just as you scheduled the call to `didTimeOut:` from `startLocationManager`, you have to cancel this call from `stopLocationManager` just in case the location manager is stopped before the time-out fires (i.e. within one minute after starting).

▸ Add the `didTimeOut` method below `stopLocationManager`:

```
- (void)didTimeOut:(id)obj
{
  NSLog(@"*** Time out");

  if (_location == nil) {
    [self stopLocationManager];

    _lastLocationError = [NSError errorWithDomain:
```

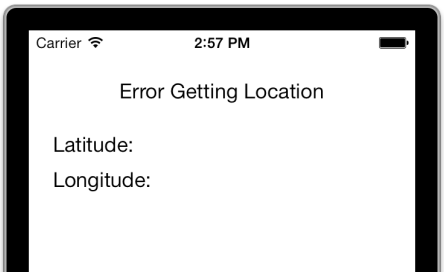
```
                    @"MyLocationsErrorDomain" code:1 userInfo:nil];

      [self updateLabels];
      [self configureGetButton];
    }
  }
```

The `didTimeOut:` method is always called after one minute, whether you've obtained a valid location or not (unless `stopLocationManager` cancels the timer first). If after that one minute there still is no valid location, you stop the location manager, create your own error code, and update the screen. By creating your own `NSError` object and putting it into the `_lastLocationError` instance variable, you don't have to change any of the logic in `updateLabels`.

However, you do have to make sure that the error's domain is not `kCLErrorDomain` because this error object does not come from Core Location but from within your own app. A domain is simply a string, so `@"MyLocationsErrorDomain"` will do. For the code I picked `1`. The value of `code` doesn't really matter at this point because you only have one custom error, but you can imagine that when the app becomes bigger you'd have the need for multiple error codes.

Note that you don't always have to use an `NSError` object; there are other ways to let the rest of your code know that an error occurred. But in this case `updateLabels` was already using an `NSError` anyway, so making your own error object just made sense.

➤ Run the app. Set the Simulator location to None and press Get My Location. After a minute, the debug area should say "*** Time out" and the Stop button reverts to Get My Location. There should be an error message in the screen.



The error after a time out

Just getting a simple location from Core Location and finding the corresponding street address turned out to be a lot more hassle than it initially appeared. There are many different situations to handle. Nothing is guaranteed and everything can go wrong.

To recap, the app either:

• Finds a location with the desired accuracy

- Finds a location that is not as accurate as you'd like and you don't get any more accurate readings
- Doesn't find a location at all
- Takes too long finding a location

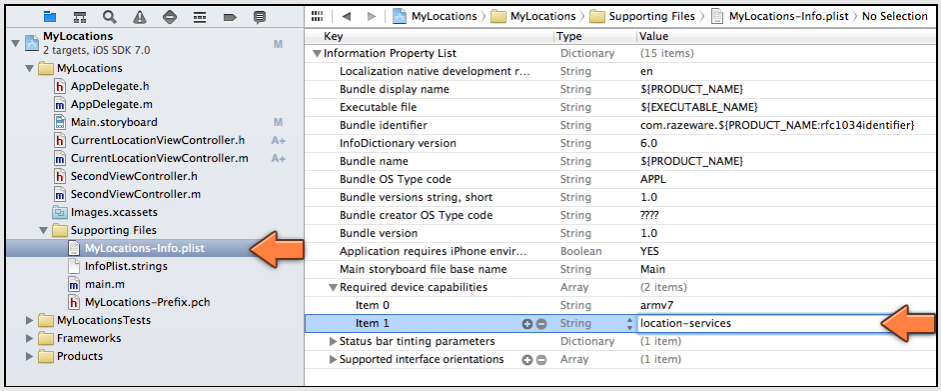
The code now handles all these situations, but I'm sure it's not perfect yet. No doubt the logic could be tweaked more but it will do for the purposes of this tutorial. I hope it's clear that if you're releasing a location-based app, you need to do a lot of field testing!

Required Device Capabilities

The Info.plist file has a field, **Required device capabilities**, that lists the hardware that your app needs in order to run. This is the key that the App Store uses to determine whether a user can install your app on their device.

The default value is **armv7**, which is the CPU architecture of the iPhone 3GS and later models. If your app requires additional features, such as Core Location to retrieve the user's location, you should list them here.

➤ Add an item with the value `location-services` to **MyLocations-Info.plist**:



You can also add the item `gps`, in which case the app requires a GPS receiver. When that item is present, users cannot install the app on iPod touch hardware and certain iPads. For the full list of possible device capabilities, see the *iOS App Programming Guide* on the Apple Developer website.

P.S. You can now take the `NSLog()` statements out of the app (or simply comment them out). Personally, I keep them in there as they're handy for debugging but in an app that you plan to upload to the App Store you'll want to remove the `NSLog()` statements.

You can find the project files for this first part of the app under **01 - GPS Coordinates** in the tutorial's Source Code folder.

Objects vs. classes

Time for something new. Up until now I've been calling almost everything an object. However, to use proper object-oriented programming vernacular, we have to make a distinction between an object and its **class**.

When you do this,

```
@interface ChecklistItem : NSObject
. . .
@end
```

You're really defining a class named `ChecklistItem`, not an object. An object is what you get when you **instantiate** a class:

```
ChecklistItem *item = [[ChecklistItem alloc] init];
```

The `item` variable now contains an object of the class `ChecklistItem`. It's an object because of the `*`. You can also say: the `item` variable contains an **instance** of the class `ChecklistItem`. The terms object and instance mean the same thing.

In other words, "object of class `ChecklistItem`" is the **datatype** of this `item` variable. The Objective-C language already comes with a lot of datatypes built-in but you can also add datatypes of your own by making classes.

Let's use an example to illustrate the difference between a class and an instance/object. You and me are both hungry, so we decide to eat some ice cream (my favorite subject next to programming!). Ice cream is the class of food that we're going to eat.

The ice cream class looks like this:

```
@interface IceCream : NSObject

@property (nonatomic, strong) NSString *flavor;
@property (nonatomic, assign) int scoops;

- (void)eatIt;

@end
```

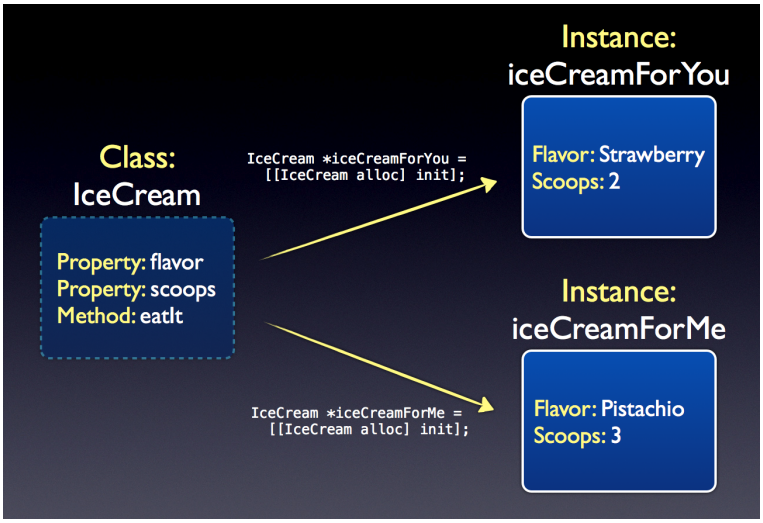
You and I go on over to the ice cream stand and ask for two cones:

```
// one for you
IceCream *iceCreamForYou = [[IceCream alloc] init];
iceCreamForYou.flavor = @"Strawberry";
iceCreamForYou.scoops = 2;
```

```
// and one for me
IceCream *iceCreamForMe = [[IceCream alloc] init];
iceCreamForMe.flavor = @"Pistachio";
iceCreamForMe.scoops = 3;
```

Yep, I get more scoops but that's because I'm hungry from all this explaining. ;-)

Now the app has two instances of `IceCream`, one for you and one for me. There is just one class that describes what sort of food we're eating – ice cream – but there are two distinct objects. Your object has strawberry flavor, mine pistachio.



The class is a template for making new instances

The `IceCream` class is like a template that says, objects of this type have two properties: `flavor` and `scoops` (whose values will be stored in backing instance variables with the same names) and a method named `eatIt`. Any new instance that is made from this template will have those properties, instance variables, and methods, but it lives in its own section of computer memory and has its own values.

If you're more into architecture than food, then you can also think of a class as a blueprint for a building. It is the design of the building, but not the building itself. One blueprint can make many buildings, and you could paint each one – each instance – a different color if you wanted to.

Inheritance

No, this is not where I tell you that you've inherited a fortune. We're talking about **class inheritance** here, one of the main principles of object-oriented programming.

Inheritance is a powerful feature that allows a class to be built on top of another class. The new class takes over all the data and functionality from that other class and adds its own specializations to it.

Take the `IceCream` class from the previous example. It is built on `NSObject`, the fundamental class in Objective-C. You can see that in the `@interface` line:

```
@interface IceCream : NSObject
```

This means the `IceCream` class is actually the `NSObject` class with a few additions of its own, namely the `flavor` and `scoops` properties and the `eatIt` method.

`NSObject` is the **base class** for almost all other classes in Objective-C. Any classes that you create either directly inherit from `NSObject` or from another class that is ultimately based on `NSObject`. You can't escape it!

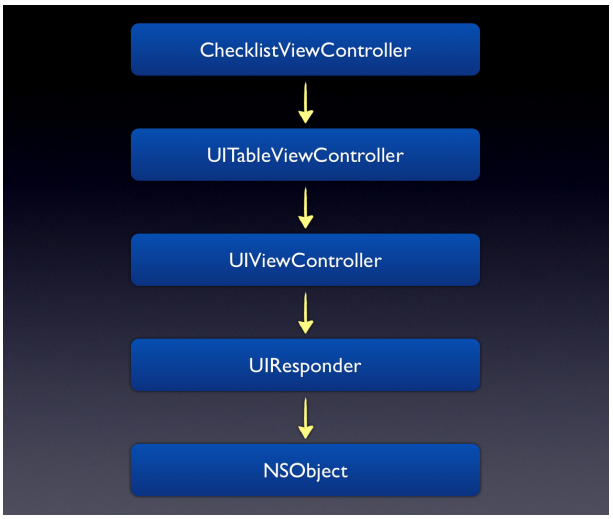
You've also seen interface lines that look like this:

```
@interface ChecklistViewController : UITableViewController
```

The `ChecklistViewController` class is really a `UITableViewController` class with your own additions. It does everything a `UITableViewController` can, plus whatever new data and functionality you've given it.

This inheritance thing is very handy because `UITableViewController` already does a lot of work for you behind the scenes. It has a table view, knows how to deal with prototype cells and static cells, and handles things like scrolling and a ton of other stuff. All you have to do is add your own customizations and you're ready to go. If you would have to program each screen totally from scratch, you'd still be working on lesson 1! Inheritance lets you re-use a lot of existing code with minimal effort.

`UITableViewController` itself is built on top of `UIViewController`, which is built on top of something called `UIResponder`, and ultimately that class is built on `NSObject`. This is called the inheritance tree.



All classes stand on the shoulders of NSObject

The big idea here is that each object that is higher up performs a more specialized task than the one below it.

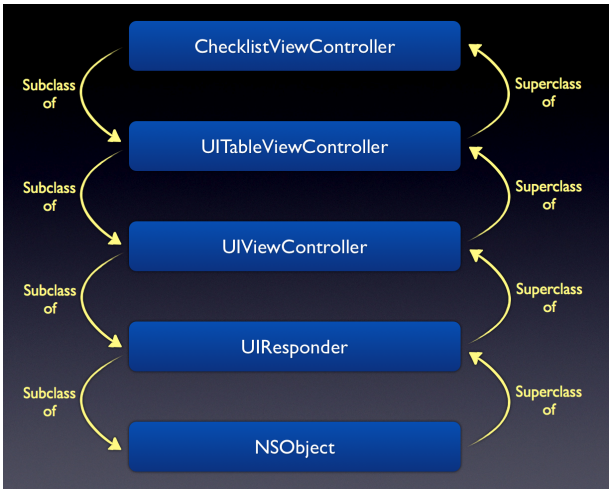
`NSObject`, the base class, only provides a few basic functions that are needed for all objects. For example, it contains the `alloc` method that is used to reserve memory space for the object's instance variables, and the basic `init` method.

`UIViewController` is the base class for all view controllers. If you want to make your own view controller, you extend `UIViewController`. To **extend** means that you make a class that inherits from this one. Other commonly used terms are **to derive from** or **to base on**. These phrases all mean the same thing.

You don't want to write all your own screen and view handling code. That stuff has been taken care of for you by very smart people working at Apple and they've bundled it into `UIViewController`. You simply make a class that inherits from `UIViewController` and you get all that functionality for free. You just add your own data and logic to that class and off you go!

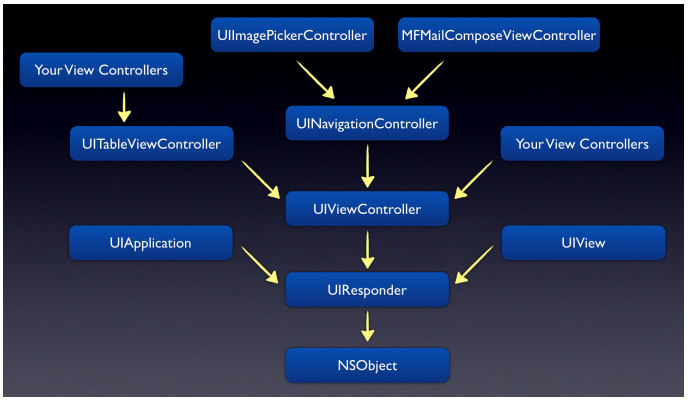
If your screen primarily deals with a table view then you'd make your class inherit from `UITableViewController` instead. This class does everything `UIViewController` does – because it inherits from it – but is more specialized for dealing with table views. You could write all that code by yourself, but why would you when it's already available in a convenient package? Class inheritance can save you a lot of time!

When programmers talk about inheritance, they'll often throw around the terms **superclass** and **subclass**. In the example above, `UITableViewController` is the immediate superclass of `ChecklistViewController`, and conversely `ChecklistViewController` is a subclass of `UITableViewController`. The superclass is the class you derived from, while a subclass derives from your class.



Superclass and subclass

A class in Objective-C can have many subclasses but only one immediate superclass. Of course, that superclass can have a superclass of its own. There are many different classes that inherit from `UIViewController`, for example:



A small portion of the UIKit inheritance tree

Because nearly all classes extend from `NSObject`, they form a big hierarchy. It is important that you understand this class hierarchy so can you make your own objects inherit from the proper superclasses. As you'll see later in this tutorial, there are many other types of hierarchies in programming. For some reason programmers seem to like them.

By the way, you may have noticed that protocols do something similar:

```
@protocol MyProtocol <NSObject>
. . .
@end
```

When an `@interface` line has `< >` brackets, it declares that class to conform to one or more protocols. Protocol definitions themselves can also do this. In the above example `MyProtocol` says it conforms to the `NSObject` protocol.

This isn't true inheritance the way classes do it but the principle is similar. It establishes that any class conforming to `MyProtocol` should also implement the methods from the `NSObject` protocol.

Note that there is an `NSObject` class (the base class for almost all other classes in iOS) but also an `NSObject` protocol (the base protocol that all other protocols should conform to). I hope this isn't too confusing. They are both called `NSObject` but one is a class and one is a protocol.

Just remember that all your classes should at least inherit from the `NSObject` class and that all your own protocols should at least conform to the `NSObject` protocol, and you should be all set.

Overriding methods

Inheriting from a class means your own class gets to use the methods from its superclass.

If you create a new class,

```
@interface MyClass : NSObject
@end
```

then elsewhere in your code you can do:

```
MyClass *myObject = [[MyClass alloc] init];
```

This works even though `MyClass` did not explicitly declare an `alloc` method or an `init` method. But `NSObject` does! Because `MyClass` inherits from `NSObject`, it automatically gets these methods for free.

`MyClass` can also provide its own `init` method:

```
@implementation MyClass

- (id)init
{
  if ((self = [super init])) {
    // do stuff
  }
  return self;
}

@end
```

Now when someone calls `[[MyClass alloc] init]`, this new version of the `init` method is invoked. You still need to call `NSObject`'s version of `init` so that the superclass can initialize itself properly, which explains why the first thing your own `init` methods should do is `[super init]`. That is the reason you've been calling `super` in various places in your code, to let any superclasses do their thing.

You don't need to put a declaration for `init` in your interface section:

```
@interface MyClass : NSObject

- (id)init;  // not needed

@end
```

This won't harm any but it's a little superfluous because `NSObject`'s interface already does it. Since you're inheriting from `NSObject`, it is already known that there is a method named `init`.

However, if you create an initializer method that has a different name, then you do need to put it in the interface otherwise no one will know about it:

```
@interface MyClass : NSObject

- (id)initWithText:(NSString *)text;

@end
```

Methods can be used for communicating between a class and its subclasses, so that the subclass can perform specific behavior in certain circumstances. That is what methods such as `viewDidLoad` and `viewWillAppear:` are for. These methods are defined and implemented by `UIViewController` but your own view controller subclass can **override** them.

For example, when its screen is about to become visible, the `UIViewController` class will call `[self viewWillAppear:YES]`. Normally this will invoke the `viewWillAppear:` method from `UIViewController` itself, but if you've provided your own version of this method in your subclass, then yours will be invoked instead. By overriding `viewWillAppear:`, you get a chance to handle this event before the superclass does:

```
@implementation MyViewController

- (void)viewWillAppear:(BOOL)animated
{
  // don't forget to call this!
  [super viewWillAppear:animated];

  // do your own stuff here
}
```

```
@end
```

That's how you can tap into the power of your superclass. A well-designed superclass provides such "hooks" that allow you to react to certain events. Don't forget to call `super`'s version of the method, though. If you neglect this, the superclass will not get its own notification and weird things may happen.

Does a subclass get to use *all* the methods from its superclass? Not quite. `UIViewController` has a lot more methods hidden away in its implementation than are visible in its interface. A subclass really only gets access to the publicly visible methods of its superclass. It is in theory possible to call hidden methods if you know their names, but this is not recommended (and may even get your app rejected from the App Store).

Protected instance variables

Subclasses also inherit all properties and instance variables from their superclasses. Inside a subclass you can use all of the parent's properties (at least the ones that are declared in the interface section) but it is unlikely that you can access the instance variables of a superclass.

We've already discussed that everything in an object's interface is publicly visible to other objects and everything in its implementation is hidden. But what if those other objects are subclasses? Well, there is something in between public and private and that is **protected**. Data that is private cannot be used by subclasses but anything that is marked as protected can.

If you declare instance variables, you can use the special keyword `@protected` to make them available to subclasses:

```
@implementation IceCream
{
@protected
  float _percentageSugar;
}

@end
```

This variable `_percentageSugar` can now be used by subclasses of `IceCream`:

```
@interface FrozenYoghurt : IceCream
@end

@implementation FrozenYoghurt

- (void)eatIt
```

```
{
  NSLog(@"Sugar: %f", _percentageSugar);
}

@end
```

In other languages the use of protected instance variables is more common but I have only had to do this once or twice in my Objective-C programs. Usually putting publicly available data in properties is sufficient, and you can keep your instance variables hidden.

Casts

`NSObject` is the base class for almost all other classes in your apps, so can you refer to your objects as if they were instances of `NSObject`:

```
NSString *text = @"Hello, world";
NSObject *o = text;
NSLog(@"The text is: %@", o);
```

This works because `NSString` is in essence an `NSObject` with some additions. You can even do it like this:

```
NSObject *o = @"Hello, world";
NSLog(@"The text is: %@", o);
```

However, the following won't work:

```
NSNumber *n = @"Hello, world";
```

`NSString` is not a superclass of `NSNumber`, so you cannot make it act like a string.

Why would you ever want to treat an object as if it was one of its superclasses? The app you're writing in this tutorial actually has a good example of that. It has a `UITabBarController` with three tabs, each of which is represented by a view controller. The view controller for the first tab is `CurrentLocationViewController`. Later in this chapter you'll add two others, `LocationsViewController` for the second tab and `MapViewController` for the third.

The designers of the tab bar component obviously didn't know anything about those three view controllers when they created `UITabBarController`. The only thing the tab bar controller can reliably depend on is that each tab has a view controller that inherits from `UIViewController`.

So instead of talking to the `CurrentLocationViewController` class, the tab bar controller only sees its superclass part, `UIViewController`. As far as the tab bar controller is concerned it has three `UIViewController` instances and it doesn't know or care about the additions that you've made to them.

The same thing goes for `UINavigationController`. To the navigation controller any new view controllers you push on the navigation stack are all instances of `UIViewController`, nothing more, nothing less.

Sometimes that can be a little annoying. When you ask the navigation controller for one of the view controllers on its stack, it necessarily returns a pointer to a `UIViewController` object, even though that is not the full datatype of that object. If you want to treat that object as your own view controller subclass instead, you need to **cast** it to the proper type.

For example, in the previous tutorial you did the following in `prepareForSegue`:

```
UINavigationController *navigationController =
                                segue.destinationViewController;

ItemDetailViewController *controller =
            (ItemDetailViewController *)
                        navigationController.topViewController;

controller.delegate = self;
```

Here you wanted to get the top view controller from the navigation stack, which is an instance of `ItemDetailViewController`, and set its `delegate` property.

However, `UINavigationController`'s `topViewController` property won't give you an object of type `ItemDetailViewController` but just a plain `UIViewController`, which naturally doesn't have your `delegate` property.

If you were to do,

```
ItemDetailViewController *controller =
                        navigationController.topViewController;
```

then Xcode gives the warning: "Incompatible pointer types".

You can't just put any `UIViewController` object into an `ItemDetailViewController` variable. Even though all `ItemDetailViewControllers` are `UIViewControllers`, not all `UIViewControllers` are `ItemDetailViewControllers`! Just because your friend Willy has no hair, that doesn't mean all bald guys are named Willy.

To solve this problem, you have to **cast** the object to the proper type. You know this object is an `ItemDetailViewController`, so you use the cast operator `()` to tell the compiler, "I want to treat this object as an `ItemDetailViewController`."

With the cast, the code looks like this:

```
ItemDetailViewController *controller =
            (ItemDetailViewController *)
                        navigationController.topViewController;
```

(You would put this all on a single line in Xcode. Having long descriptive names is great for making the source code more readable, but it also necessitates clumsy line wrapping to make it fit in the book.)

Now you can treat the value from `topViewController` as an `ItemDetailViewController` object. The compiler doesn't check whether the thing you're casting really is that kind of object, so if you're wrong and it's not, your app will most likely crash.

A cast doesn't magically convert one type into another. You can't cast an `NSNumber` into an `NSString`, for example. You only use a cast to make a type more specific, and the two types have to be compatible for this to work.

The "id" type

When you ask an `NSArray` for an object using the `[ ]` brackets or the `objectAtIndex:` method, you're given an object with type `id`.

`NSArray` needs to be able to store any kind of object, but it can't really make any assumptions based on those objects other than that they probably extend `NSObject`. In fact, it can't even make that assumption because there are some rare objects that do not inherit from `NSObject` at all. The Objective-C language has a provision for this: the `id` type.

`id` is a special Objective-C keyword that means "any object". It's similar to `NSObject` except that it doesn't make any assumptions at all about what kind of object it is – `id` describes even less about an object than `NSObject` does. `id` doesn't have any methods, properties or instance variables, it's a completely naked object.

```
id o = @"Hello, world";
NSLog(@"The text is: %@", o);
```

You'll often use `id` to describe an object that conforms to a specific protocol, without having to know anything about its class. That's what you've been doing for your delegates all along:

```
@property (nonatomic, weak) id
                    <ItemDetailViewControllerDelegate> delegate;
```

The notation `id <ItemDetailViewControllerDelegate>` means: "I only care that this object implements the `ItemDetailViewControllerDelegate` protocol but its actual class is not important to me."

You could also do it as follows:

```
@property (nonatomic, weak) NSObject
                   <ItemDetailViewControllerDelegate> *delegate;
```

That certainly works but it's customary to use `id` instead.

Note that you don't use an * asterisk when you use `id`; any variable of type `id` is implied to be a pointer.

`id` has other special rules as well. You can send any message you want to `id` and the compiler won't give you a warning. That makes it a very powerful tool, but also very dangerous. Without being strict about the types of objects, it's easy to send the wrong message to the wrong object and iOS dislikes that so much that it will crash your app.

You also don't need to cast `id` objects:

```
NSString *s = [someArray objectAtIndex:3];
  // or
NSString *s = someArray[3];
```

If this had returned an `NSObject *` instead of `id`, then you would have had to write:

```
NSString *s = (NSString *)[someArray objectAtIndex:3];
  // or
NSString *s = (NSString *)someArray[3];
```

But because it returns `id`, you can leave out the cast, making the code much simpler.

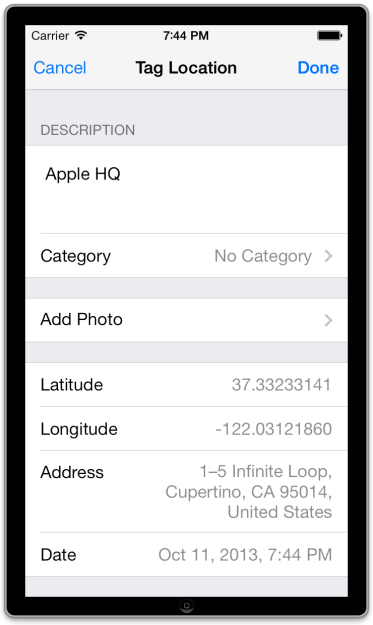
OK, that's enough theory for now. I hope this gives you a little more insight into objects and classes. The only way to really get the hang of this object-oriented programming thing is do it, so let's continue with some coding.

The Tag Location screen

There is a big button on the main screen of the app that says Tag Location. It only becomes active when GPS coordinates have been captured, and you use it to add a description and a photo to that location.

In this section you'll build the Tag Location screen but you won't save the location object anywhere yet, that's the topic of the next section. The Tag Location screen is a regular table view controller with static cells, so this is going to be very similar to what you did a few times already in the previous tutorial.

The finished Tag Location screen will look like this:



The Tag Location screen

The description cell at the top contains a `UITextView` for text. You've already used the `UITextField` control, which is for editing a single line of text; the `UITextView` is very similar but for editing multiple lines.

Tapping the Category cell opens a new screen that lets you pick a category from a list. This is very similar to the icon picker from the last tutorial, so no big surprises there either.

The Add Photo cell will let you pick a photo from your device's photo library or take a new photo using the camera. You'll skip this feature for now and build that later in this tutorial. Let's not get ahead of ourselves and try too much at once!

The other cells are read-only and contain the latitude, longitude and address information that you just captured, and the current date so you'll know when it was that you tagged this location. (You can't see the Date field in the above screenshot but it's below the Address cell.)

Exercise. Try to implement this screen by yourself using the description I just gave you. You don't have to make the Category and Add Photo buttons work yet. Yikes, that seems like a big job! It sure is, but you should be able to pull this off. This screen doesn't do anything you haven't done in the previous tutorial. So if you feel brave, go ahead!

Adding the new view controller

➤ Add a new file to project. Make it a subclass of `UITableViewController` and name the file **LocationDetailsViewController**. The "Targeted for iPad" and "With XIB for user interface" options should be unchecked.

You know what's next: create outlets, connect them to the controls on the storyboard, and so on. In the interest of saving time, I'll just give you the code that you're going to end up with.

▸ Replace the contents of **LocationDetailsViewController.m** with the following:

```
#import "LocationDetailsViewController.h"

@interface LocationDetailsViewController ()

@property (nonatomic, weak) IBOutlet UITextView
                                        *descriptionTextView;
@property (nonatomic, weak) IBOutlet UILabel *categoryLabel;
@property (nonatomic, weak) IBOutlet UILabel *latitudeLabel;
@property (nonatomic, weak) IBOutlet UILabel *longitudeLabel;
@property (nonatomic, weak) IBOutlet UILabel *addressLabel;
@property (nonatomic, weak) IBOutlet UILabel *dateLabel;

@end

@implementation LocationDetailsViewController

- (IBAction)done:(id)sender
{
}

- (IBAction)cancel:(id)sender
{
}

@end
```

Nothing special here, just a bunch of outlet properties and two action methods. Except… why are these properties declared inside the .m file and not in the .h file?

You may have noticed before that Xcode already puts an `@interface` line at the top of new .m files, just like in .h files:

```
@interface LocationDetailsViewController ()
```

However, this is not a regular interface declaration, thanks to the `()` at the end. The presence of those parentheses makes this a so-called **class extension**. A class extension is an addition to the `@interface` section of the class, but one that you keep hidden. It's a special form of another Objective-C feature, **categories**, that you'll encounter later in this tutorial.

The cool thing about a class extension is that it allows you to move your outlet properties into the .m file, to hide them from the other objects in your app. There is

no reason why any of those other objects should have access to the `descriptionTextView` or the `categoryLabel`, for instance. Those things are private to this view controller and therefore only the `LocationDetailsViewController` should be allowed to do stuff with them. By placing these outlets in the .m file inside the class extension, they become invisible to any other view controllers.

The same thing goes for the two action methods, `done:` and `cancel:`. If you don't declare them inside the .h file, they will become private. In this case, there is no reason for any object outside this view controller to ever call these methods, so you might as well keep them hidden. As long as they are declared `IBAction`, Interface Builder will still let you connect the action methods to buttons inside this view controller.

Because everything is now inside the .m file, you don't need to make any changes to **LocationDetailsViewController.h**. The less there is inside your .h files, the better.

➤ In the storyboard, drag a new Table View Controller into the canvas and put it next to the Current Location View Controller.

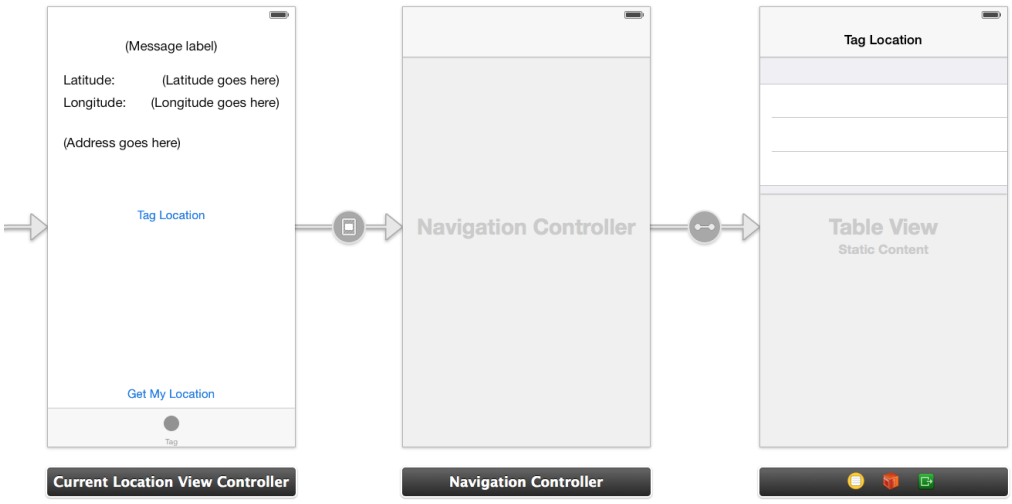
➤ With the Table View Controller selected, choose **Editor → Embed In → Navigation Controller** from Xcode's menubar to put it inside a new navigation controller.

➤ **Ctrl-drag** from the Tag Location button to this new navigation controller and create a **modal** segue. Give it the identifier **TagLocation**.

➤ In the **Identity inspector**, change the **Class** attribute of the table view controller to **LocationDetailsViewController** to link it with the `UITableViewController` subclass you just created.

➤ Double-click the Location Details View Controller's navigation bar and change the title to **Tag Location**. Switch the table content to **Static Cells** and its style to **Grouped**.

The storyboard now looks like this:



The Tag Location screen in the storyboard

➤ Run the app and make sure the Tag Location button works.

Of course, the screen won't do anything useful yet. Let's add some buttons.

➤ Drag a **Bar Button Item** into the left slot of the navigation bar. Make it a **Cancel** button and connect it to the `cancel:` action. (Note: the thing that you're supposed to connect is the Bar Button Item's "selector", under Sent Actions.)

➤ Also drag a **Bar Button Item** into the right slot. Set both the style and identifier attributes to **Done**, and connect it to the `done:` action.

➤ Change the code for these actions to the following:

```
- (IBAction)done:(id)sender
{
  [self closeScreen];
}

- (IBAction)cancel:(id)sender
{
  [self closeScreen];
}

- (void)closeScreen
{
  [self dismissViewControllerAnimated:YES completion:nil];
}
```

There's a good reason why you're putting this in a separate method but that won't become clear until later.

➤ Run the app again and make sure you can close the Tag Location screen after you've opened it.

Making the cells

There will be three sections in this table view:

1. The description text view and the category cell. These can be changed by the user.
2. The photo. Initially this cell says Add Photo but once the user has picked a photo you'll display the actual photo inside the cell. So it's good to have that in a section of its own.
3. The latitude, longitude, address and date rows. These are read-only information.

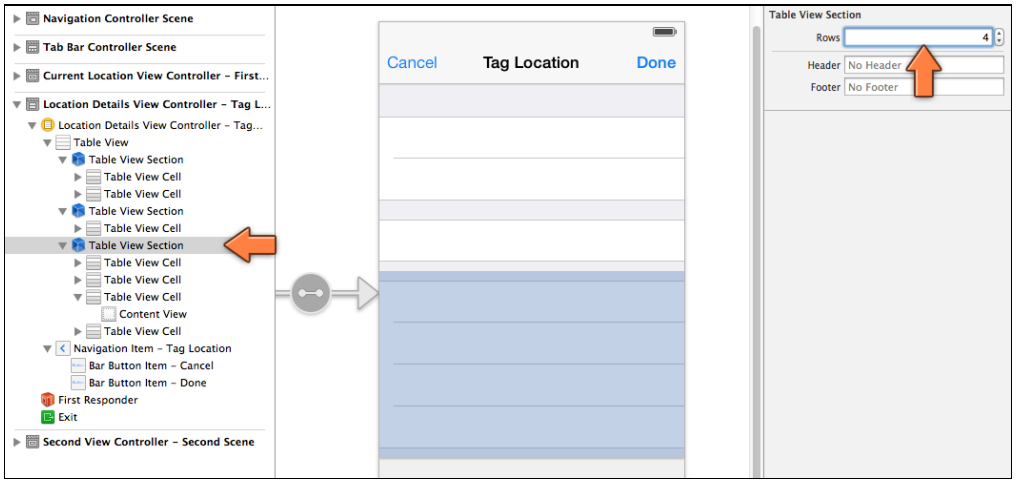
➤ Select the table view and go to the **Attributes inspector**. Change the **Sections** field from 1 to 3.

When you do this, the contents of the first section are automatically copied to the next sections. That isn't quite what you want, so you'll have to remove some rows here and there. The first section will have 2 rows, the middle section will have just 1 row, and the last section will have 4 rows.

➤ Select one cell in the first section and delete it.

➤ Delete two cells from the middle section.

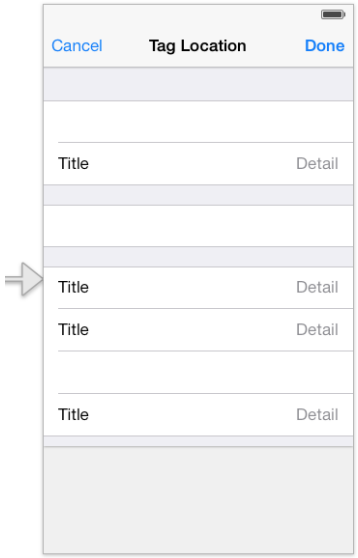
➤ Select the last section object (that is easiest in the bar on the left of the storyboard editor) and in the **Attributes inspector** set its **Rows** to 4. (Alternatively you can drag a new Table View Cell from the Object Library into the table.)



Adding a row to a table view section

The second row from the first section and the first, third and fourth rows in the last section will all use a standard cell style.

➤ Select these cells and set their **Style** attribute to **Right Detail**.



The cells with the Right Detail style

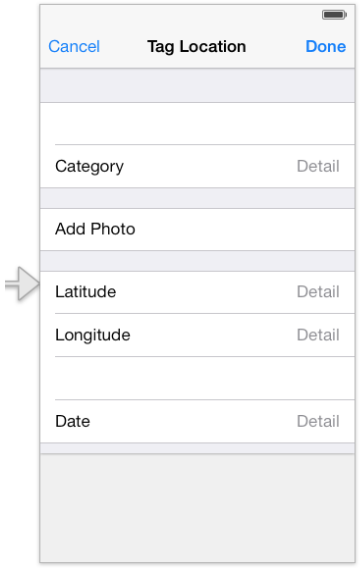
The labels in these standard cell styles are still regular `UILabels`, so you can select them and change their properties.

➤ Name these labels from top to bottom: **Category**, **Latitude**, **Longitude**, and **Date**.

➤ Drag a new **Label** into the cell from the middle section (the one that's still empty). You cannot use a standard cell style for this one so you'll design it yourself. Name this label **Add Photo**. (Later on you'll also add an image view to this cell.)

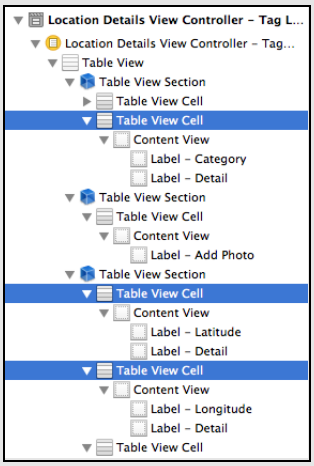
➤ Put the Add Photo label at X: 15 (in the **Size inspector**) and vertically centered in its cell. You can use the **Editor** → **Align** → **Vertical Center in Container** menu option for this.

The table should now look like this:



The labels in the Tag Location screen

Note: You're going to make a bunch of changes that are the same for each cell. For some of these, it is easier if you select all the cells at once and then change the setting. That will save you some time. My version of Xcode doesn't let you select more than one cell at a time when they are in different sections, but you can do it in the sidebar on the left by holding down ⌘ and clicking on multiple cells:

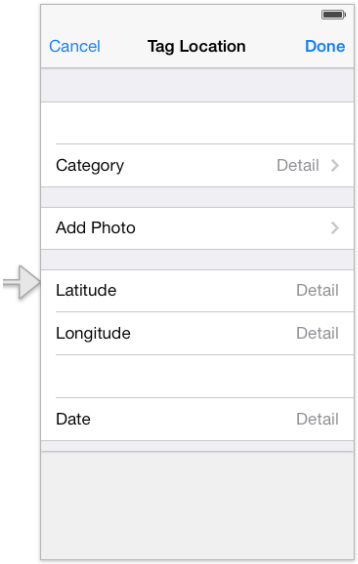


Unfortunately, some menu items are grayed out when you have a multiple selection, so you'll still have to change some of the settings for each cell individually.

Only the Category and Add Photo cells are tap-able, so you have to set the cell selection color to None on the other rows.

➤ Select all the cells except Category and Add Photo (this is easiest from the sidebar where you can select them all at once). In the **Attributes inspector**, set **Selection** to **None**.

➤ Select the Category and Add Photo cells and set **Accessory** to **Disclosure Indicator**.



Category and Add Photo now have a disclosure indicator

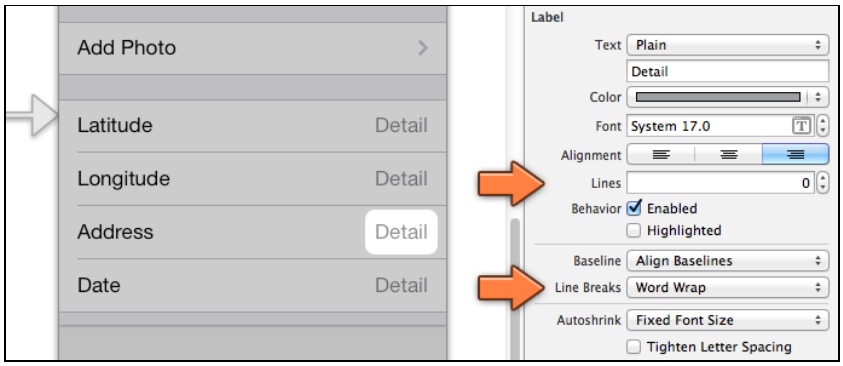
The empty cell in the last section is for the Address label. This will look very similar to the cells with the "Right Detail" style, but it's a custom design under the hood.

➤ Drag a new **Label** into that cell and name it **Address**. Put it on the left.

➤ Drag another **Label** into the same cell and name it **Detail**. Put it on the right. Change the **Alignment** to right-aligned and the **Color** to a medium gray (Red: 142, Green: 142, Blue: 147).

The detail label is special. Most likely the street address will be too long to fit in that small space, so you'll configure this label to have a variable number of lines. This requires a bit of programming in the view controller to make it work, but you also have to set up this label's attributes properly.

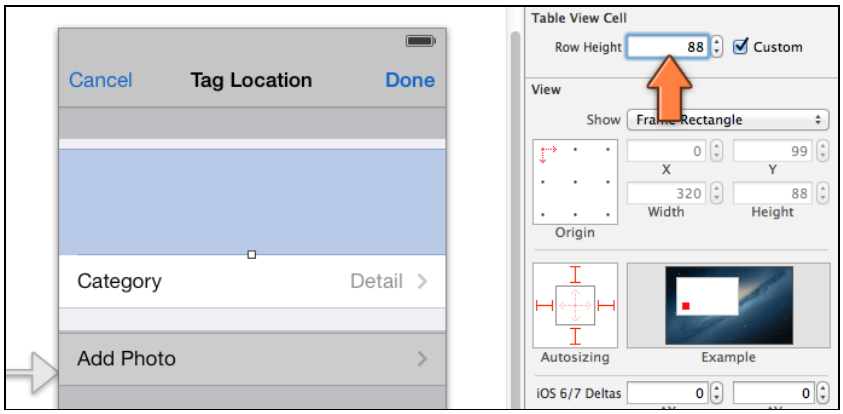
➤ In the **Attributes inspector** for the address detail label, set **Lines** to **0** and **Line Breaks** to **Word Wrap**. When the number of lines is 0, the label will resize vertically to fit all the text that you put into it, which is exactly what you need.



The address label can have multiple lines

So far you've left the cell at the top empty. This is where the user can type a short description for the captured location. Currently there is not much room to type anything, so first you'll make the cell larger.

› Click on the top cell to select it, then go into the **Size inspector** and type **88** into the **Row Height** field.



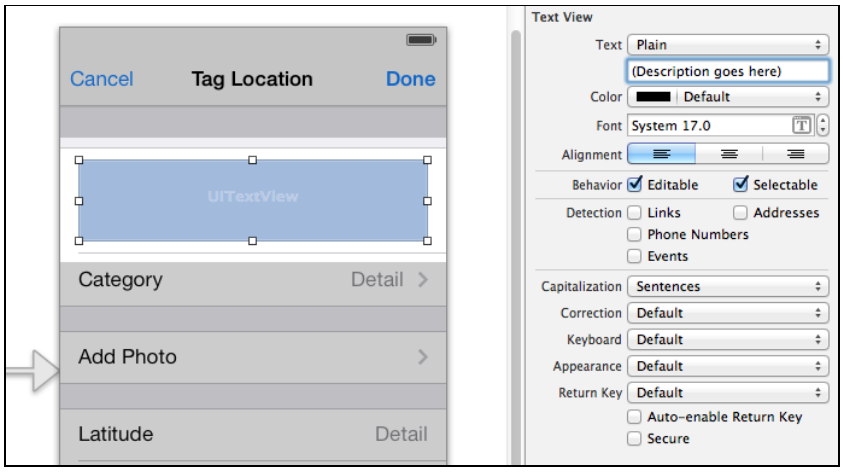
Changing the height of a row

You can also drag the cell to this new height by the handle at its bottom, but I prefer to simply type in the new value. The reason I chose 88 is that mostly everything in the iOS interface has a size of 44 points. The navigation bar is 44 points high, regular table view cells are 44 points high, and so on. Choosing 44 or a multiple of it keeps the UI looking balanced.

› Drag a **Text View** into the cell. Give it the following position and size, X: 15, Y: 10, Width: 290, Height: 68.

› By default, Interface Builder puts a whole bunch of fake Latin text (Lorem ipsum dolor, etc) into the text view, although it is not visible in the storyboard (at least in Xcode 5.0). Replace that text with "(Description goes here)". The user will never see that text, but it's handy to remind yourself what this view is for.

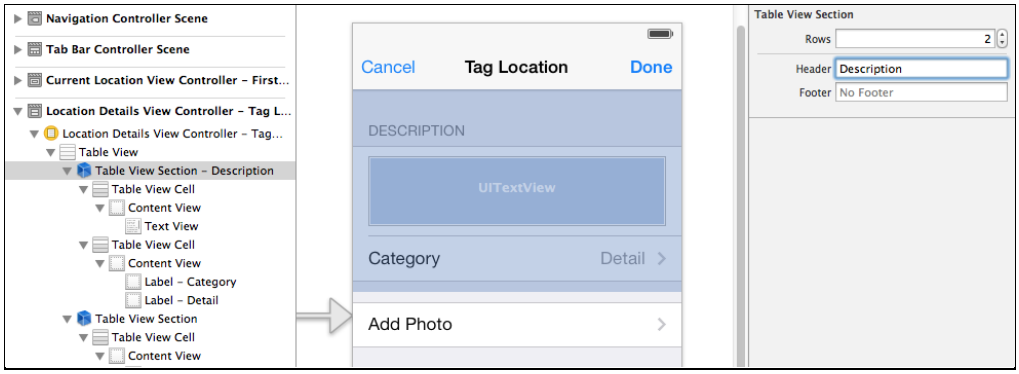
› Set the font to **System**, size **17**.



The attributes for the text view

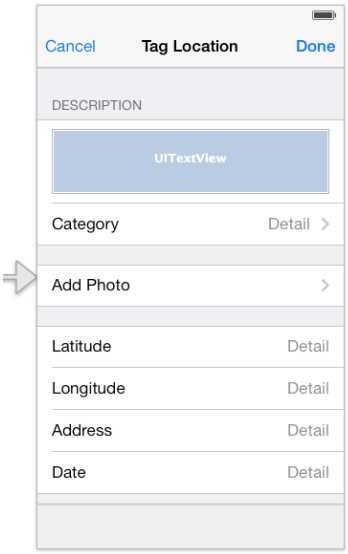
One more thing to do and then the layout is complete. Because the top cell doesn't have a label to describe what it does – and it will initially be empty as well – the user may not know what it is for. There really isn't any room to add a label in front of the text view, as you've done for the other rows, so let's add a header to the section. Table view sections can have a header and footer, and these can either be text or a complete view with controls of its own.

› Select the top-most Table View Section and in its **Attributes inspector** type **Description** into the **Header** field:



Giving the section a header

That's the layout done. The Tag Location screen should look like this in the storyboard:

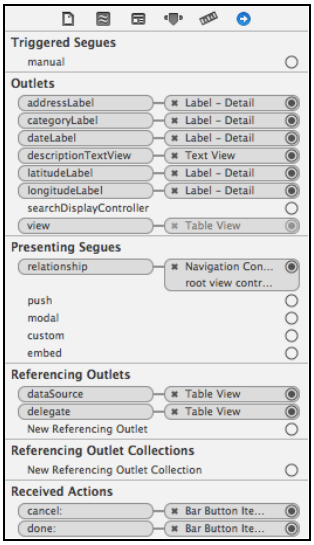


The finished design of the Tag Location screen

Now you can actually make the screen do stuff.

➤ Connect the labels and the text view to their respective outlets. It should be obvious which one goes where.

If you click on the Location Details View Controller and look at the **Connections inspector**, you should see the following:



The connections of the Location Details View Controller

➤ Run the app to test whether everything works.

Of course, the screen still says "Detail" in the labels instead of the location's coordinates and address because you haven't given it this data yet.

Putting the location info into the screen

➤ Add two properties to **LocationDetailsViewController.h**:

```
@interface LocationDetailsViewController : UITableViewController

@property (nonatomic, assign) CLLocationCoordinate2D coordinate;
@property (nonatomic, strong) CLPlacemark *placemark;

@end
```

You've seen the `CLPlacemark` object before. It contains the address information – street name, city name, and so on – that you've obtained through reverse geocoding.

The `CLLocationCoordinate2D` object is new. This contains the latitude and longitude from the `CLLocation` object that you received from the location manager. You only need those two fields, so there's no point in sending along the entire `CLLocation` object.

Notice that this time you added the properties to the .h file, not to the class extension inside the .m file. That's because these two properties must be available to other objects, i.e. they are "public" rather than "private". On the segue from the Current Location screen to the Tag Location screen you will fill in these two properties, and then the Tag Location screen can put their values into its labels.

Structs

I said `CLLocationCoordinate2D` was an object, but that's not strictly true. (If it was an object there would be a `*` behind its name.) `CLLocationCoordinate2D` is not an object but a **struct**.

Structs are a holdover from the C language. They allow you to combine multiple variables into a single data structure. In Objective-C you use a class for that purpose but C doesn't have classes, only structs. Unlike a class, a struct has no methods, just data. It's basically a list of variables.

The definition for `CLLocationCoordinate2D` is as follows:

```
typedef struct {
    CLLocationDegrees latitude;
    CLLocationDegrees longitude;
} CLLocationCoordinate2D;
```

This struct has two fields, `latitude` and `longitude`. Both these fields have the datatype `CLLocationDegrees`, which is a synonym for `double`:

```
typedef double CLLocationDegrees;
```

The `double` datatype is one of the primitive types built into the language. It's like a `float` but with higher precision. Don't let these synonyms confuse you; `CLLocationCoordinate2D` is basically this:

```
typedef struct {
    double latitude;
    double longitude;
} CLLocationCoordinate2D;
```

The reason the designers of Core Location used `CLLocationDegrees` instead of `double` is that "CL Location Degrees" tells you what this datatype is intended for: it stores the degrees of a location. Underneath the hood it's a `double`, but as a user of Core Location all you need to care about when you want to store latitude or longitude is that you can use the `CLLocationDegrees` type. The name of the type adds meaning.

So `CLLocationCoordinate2D` is a struct with two fields. Just like primitive types such as `int` and `BOOL`, you can use struct variables directly. You don't have to alloc or init them. (In fact, you can't because they're not objects.)

```
CLLocationCoordinate2D myCoordinate;
myCoordinate.latitude = 12.345;
myCoordinate.longitude = 67.890;
```

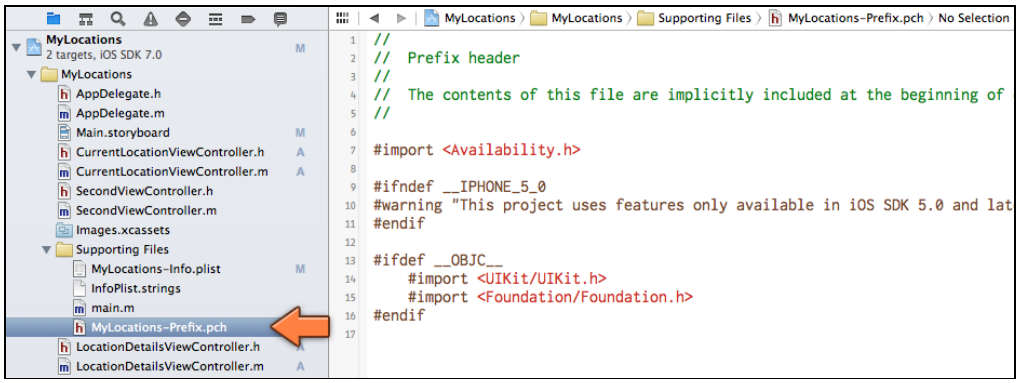
Notice the use of the dot to access the fields of a struct. This looks exactly like you're accessing properties, but you're not.

Structs are used quite regularly in UIKit and the other iOS frameworks. Other examples are `CGPoint` and `CGRect`. They are more lightweight than objects. If you just need to pass around a set of values then it's often easier to bundle them into a struct and pass that struct around, and that is exactly what Core Location does with coordinates.

If you now try to build the app, Xcode gives an error message in **LocationDetailViewController.h**: "Unknown type name 'CLLocationCoordinate2D'". This type is defined in the Core Location framework, but when Xcode compiles this source file it does not know anything about Core Location.

You could add an `#import` for the Core Location headers in this source file, just like in **CurrentLocationViewController.h**, but there's an easier solution: the **precompiled header file** (or "Prefix" file).

Every Xcode project has a **Prefix.pch** file. You can find it in the **Supporting Files** group:



The MyLocations-Prefix.pch file

In order to speed up compilation times, the precompiled header was introduced. Anything you put into this file will be automatically imported by all of your source files when they are compiled. You can see that the Prefix file already includes `#import`s for UIKit and Foundation, the two essential frameworks that any iOS app needs. That means none of your .h or .m files has to import these files anymore because they're already automatically added thanks to the Prefix file.

› Add the CoreLocation.h import to the **MyLocations-Prefix.pch** file:

```
#ifdef __OBJC__
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>
    #import <CoreLocation/CoreLocation.h>
#endif
```

Now you can build the app without problems. From this moment on, any source file in the project can use the objects and other types from the Core Location framework.

› Build the app to make sure there are no longer any errors.

You can remove the following imports from the **CurrentLocationViewController.h** if you want, as they are no longer necessary:

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

Personally, I add `#import`s to the Prefix.pch file for any system framework that I include in my projects. That saves some typing and you don't have to worry about them anymore.

Back to the new properties that you just added to `LocationDetailsViewController`. You need to set these properties when the user taps the Tag Location button.

› First, add an import statement at the top of **CurrentLocationViewController.m** to tell this class about `LocationDetailsViewController`'s properties:

```
#import "LocationDetailsViewController.h"
```

➤ Add the `prepareForSegue` method. Put it below `getLocation`:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"TagLocation"]) {
    UINavigationController *navigationController =
                                segue.destinationViewController;

    LocationDetailsViewController *controller =
                 (LocationDetailsViewController *)
                         navigationController.topViewController;

    controller.coordinate = _location.coordinate;
    controller.placemark = _placemark;
  }
}
```

You've seen how this works before. You obtain the proper destination view controller and then set its properties. Now when the segue is performed, the coordinate and address are given to the Tag Location screen. `viewDidLoad` is a good place to display these things on the screen.

➤ Add the `viewDidLoad` method to **LocationDetailsViewController.m**:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  self.descriptionTextView.text = @"";
  self.categoryLabel.text = @"";

  self.latitudeLabel.text = [NSString stringWithFormat:
                             @"%.8f", self.coordinate.latitude];
  self.longitudeLabel.text = [NSString stringWithFormat:
                            @"%.8f", self.coordinate.longitude];

  if (self.placemark != nil) {
    self.addressLabel.text = [self
                            stringFromPlacemark:self.placemark];
  } else {
    self.addressLabel.text = @"No Address Found";
  }
```

```
  self.dateLabel.text = [self formatDate:[NSDate date]];
}
```

This simply puts something in every label. It uses two helper methods that you haven't defined yet: `stringFromPlacemark` to format the `CLPlacemark` object into a string, and `formatDate` to do the same for an `NSDate` object.

▸ Add the `stringFromPlacemark:` method below `viewDidLoad:`

```
- (NSString *)stringFromPlacemark:(CLPlacemark *)placemark
{
  return [NSString stringWithFormat:@"%@ %@, %@, %@ %@, %@",
    placemark.subThoroughfare, placemark.thoroughfare,
    placemark.locality, placemark.administrativeArea,
    placemark.postalCode, placemark.country];
}
```

This is fairly straightforward. It is similar to how you formatted the placemark on the main screen, except that you also include the country.

▸ Add the `formatDate:` method as well:

```
- (NSString *)formatDate:(NSDate *)theDate
{
  static NSDateFormatter *formatter = nil;
  if (formatter == nil) {
    formatter = [[NSDateFormatter alloc] init];
    [formatter setDateStyle:NSDateFormatterMediumStyle];
    [formatter setTimeStyle:NSDateFormatterShortStyle];
  }

  return [formatter stringFromDate:theDate];
}
```

You've seen the `NSDateFormatter` class in the previous tutorial. You use it to convert the date and time that are encapsulated by the `NSDate` object into a human-readable string (in the user's language and locale settings).

Something's new here, though:

```
static NSDateFormatter *formatter = nil;
if (formatter == nil) {
  . . .
}
```

In the previous tutorial you created a new instance of `NSDateFormatter` every time you wanted to convert an `NSDate` to a string. Unfortunately, `NSDateFormatter` is a relatively expensive object to create. In other words, it takes quite long to initialize

this object. If you do that many times over then it may slow down your app (and drain the phone's battery faster).

It is better to create `NSDateFormatter` just once and then re-use that same object over and over. The trick is that you won't create the `NSDateFormatter` object until the app actually needs it. This principle is called **lazy loading** and it's a very important pattern for iOS apps.

Putting the keyword `static` in front of a local variable declaration creates a special type of variable, a so-called **static local** variable. This variable keeps its value even after the method ends. The next time you call this method, the variable isn't created anew but the existing one is used. You still can't use the variable outside of the method (it remains local) but it will stay alive once it has been created.

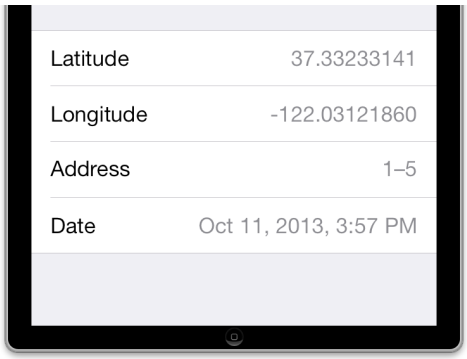
When the `formatter` variable is created it initially contains the value `nil`. Only that first time do you make the `NSDateFormatter` instance inside the if-statement. From that point on, `formatter` is no longer `nil` but contains a valid `NSDateFormatter` object. Subsequent calls to `formatDate:` can simply re-use the existing `NSDateFormatter` without having to make a new one.

Exercise. Think of another way that you could implement this type of lazy loading.

Answer: You could make the `NSDateFormatter` object an instance variable instead of a static local variable. There are other solutions as well, but I like the static local solution because it's simple. You may occasionally run into the `static` keyword when looking at other people's code so it's good to know what it does.

➤ Run the app. Choose the Apple location from the Simulator's Debug menu. Wait until the street address is visible and then press the Tag Location button.

The coordinates, address and date are all filled in, but only the first part of the address is visible (just the subthoroughfare or street number):



The Address label is too small to fit the entire address

You have earlier configured the label to fit multiple lines of text, but the problem is that the table view doesn't know about that.

➤ Add the following method to the bottom of **LocationDetailsViewController.m**:

```
#pragma mark - UITableViewDelegate

- (CGFloat)tableView:(UITableView *)tableView
            heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (indexPath.section == 0 && indexPath.row == 0) {
    return 88;
  } else if (indexPath.section == 2 && indexPath.row == 2) {

    CGRect rect = CGRectMake(100, 10, 205, 10000);
    self.addressLabel.frame = rect;
    [self.addressLabel sizeToFit];

    rect.size.height = self.addressLabel.frame.size.height;
    self.addressLabel.frame = rect;

    return self.addressLabel.frame.size.height + 20;
  } else {
    return 44;
  }
}
```

This delegate method is called by the table view when it loads its cells. You use it to tell the table view how tall each cell is. Usually, all the cells have the same height and you can simply set a property on the table view (`tableView.rowHeight` or the Row Height attribute in the storyboard) if you wanted to change the height of all the cells at once.

This table view, however, has three different cell heights:

- The Description cell at the top. You already set its height to 88 points in the storyboard.
- The Address cell. The height of this cell is variable. It may be anywhere from one line of text to several, depending on how big the address string is.
- The other cells. They all have the standard cell height of 44 points.

The three if-statements in `heightForRowAtIndexPath` correspond to those three situations. Let's take a look at the code for sizing the Address label:

```
// 1
CGRect rect = CGRectMake(100, 10, 205, 10000);

// 2
self.addressLabel.frame = rect;

// 3
```

```
  [self.addressLabel sizeToFit];

  // 4
  rect.size.height = self.addressLabel.frame.size.height;
  self.addressLabel.frame = rect;

  // 5
  return self.addressLabel.frame.size.height + 20;
```

This uses a bit of trickery to resize the `UILabel` to make it fit all the text that it contains to the width of the cell (using word-wrapping), and then you use the newly calculated height of that label to determine how tall the cell must be. Step-by-step:

1. `CGRect` is a struct that describes a rectangle. A rectangle has an origin made up of an X, Y coordinate, and a size (width and height). The `CGRectMake()` function takes four values – x, y, width and height – and puts them into a new `CGRect` struct. The width value is 205 points, but the height is a whopping 10,000 points. That is done to make the rectangle tall enough to fit a lot of text.
2. Once you have that initial rectangle that is way too high, you resize the label. The `frame` property is a `CGRect` that describes the position and size of a view. All `UIView` objects (and subclasses such as `UILabel`) have a frame rectangle. Changing the frame is how you can position views on the screen programmatically. Setting the frame on a multi-line `UILabel` has another effect: it will now word-wrap the text to fit the requested width (205 points). This works because you already set the text on the label in `viewDidLoad`.
3. Now that the label has word-wrapped its contents, you'll have to size the label back to the proper height because you don't want a cell that is 10,000 points tall. Remember the Size to Fit Content menu option from Interface Builder that you can use to resize a label to fit its contents? You can also do that from code with `sizeToFit`.
4. The call to `sizeToFit` removed any spare space to the right and bottom of the label. You want that to happen for the height of the label but not for the width, so you put the newly calculated height back into the `rect` from earlier. This gives you a rectangle with an origin at X: 100, Y: 10, a width of 205, and a height that exactly fits the text.
5. Now that you know how high the label is, you can add a margin (10 points at the top, 10 points at the bottom) to calculate the full height for the cell.

If you think this is a horrible way to figure out how large the contents are of a multiline label that does word wrapping, then I totally agree with you. (This is one of those things that Auto Layout can help with, but Auto Layout brings its own set of problems.)

▸ Run the app. Now the reverse geocoded address should completely fit in the Address cell. Try it out with a few different locations.

The description text view

Handling the description text view is very similar to what you did with the text fields in the previous tutorial.

▸ Add a new instance variable to **LocationDetailsViewController.m**. This will store the description text.

```
@implementation LocationDetailsViewController
{
  NSString *_descriptionText;
}
```

▸ Add the `initWithCoder:` method to the top of the class:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    _descriptionText = @"";
  }
  return self;
}
```

You do this just to give the instance variable an initial value.

▸ In `viewDidLoad`, put the contents of this variable in the text view (previously you put an empty string literal in there):

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  self.descriptionTextView.text = _descriptionText;

  . . .
```

▸ To the bottom of the source file, add the text view delegate methods:

```
#pragma mark - UITextViewDelegate

- (BOOL)textView:(UITextView *)textView
        shouldChangeTextInRange:(NSRange)range
        replacementText:(NSString *)text
{
  _descriptionText = [textView.text
                        stringByReplacingCharactersInRange:range
                        withString:text];
  return YES;
```

```
}

- (void)textViewDidEndEditing:(UITextView *)textView
{
  _descriptionText = textView.text;
}
```

These methods simply update the contents of the `_descriptionText` variable whenever the user types into the text view. Of course, those delegate methods won't do any good if you don't also tell the text view that it has a delegate.

▸ Change the view controller's class interface declaration to:

```
@interface LocationDetailsViewController () <UITextViewDelegate>
```

Important: this happens in the `@interface` line inside the **.m** file – in other words, in the class extension, not in the .h file. Just like you can put outlet properties in the class extension to keep them private, so you can for delegates. Why should any other object in the app care that `LocationDetailsViewController` is a delegate for a text view? That is what's called an *implementation detail*, an internal affair that callers of this class do not need to care about.

▸ Finally, go into the storyboard and **Ctrl-drag** from the text view to the view controller and connect the **delegate** outlet.

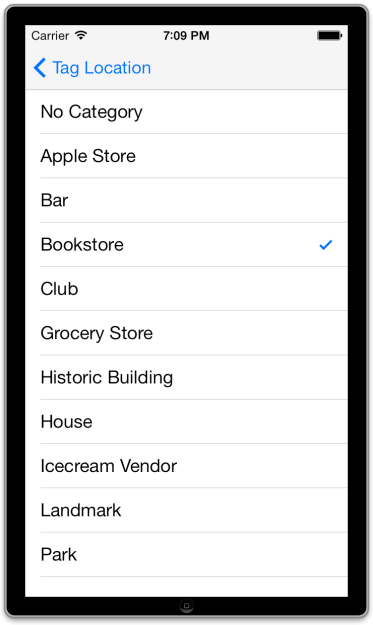
▸ Just to test whether the `_descriptionText` variable correctly captures whatever the user is typing into the text view, add an `NSLog()` to the `done` action:

```
- (IBAction)done:(id)sender
{
  NSLog(@"Description '%@'", _descriptionText);

  [self closeScreen];
}
```

▸ Run the app, tag a location and type something into the description text field. Press the Done button and verify that `NSLog()` writes the text you just typed to the debug area. Cool.

The category picker

When the user taps the Category cell, the app will show a list of category names:



The category picker

This is a new screen, so a new view controller. The way it works is very similar to the icon picker from the previous tutorial. I'm just going to give you the source code and tell you how to hook it up.

➤ Create a new table view controller subclass named **CategoryPickerViewController**.

➤ Replace the contents of **CategoryPickerViewController.h** with:

```
@interface CategoryPickerViewController : UITableViewController

@property (nonatomic, strong) NSString *selectedCategoryName;

@end
```

This is a table view controller that shows a list of category names. You can give it a category to initially select using the `selectedCategoryName` property. It will put a checkmark next to that name.

➤ Replace the contents of **CategoryPickerViewController.m** with the following:

```
#import "CategoryPickerViewController.h"

@implementation CategoryPickerViewController
{
  NSArray *_categories;
  NSIndexPath *_selectedIndexPath;
}
```

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  _categories = @[
    @"No Category",
    @"Apple Store",
    @"Bar",
    @"Bookstore",
    @"Club",
    @"Grocery Store",
    @"Historic Building",
    @"House",
    @"Icecream Vendor",
    @"Landmark",
    @"Park"];
}

#pragma mark - UITableViewDataSource

- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
  return [_categories count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
                      dequeueReusableCellWithIdentifier:@"Cell"];

  NSString *categoryName = _categories[indexPath.row];
  cell.textLabel.text = categoryName;

  if ([categoryName isEqualToString:
                                    self.selectedCategoryName]) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
    _selectedIndexPath = indexPath;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }

  return cell;
```

```
}

#pragma mark - UITableViewDelegate

- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (indexPath.row != _selectedIndexPath.row) {
    UITableViewCell *newCell = [tableView
                                cellForRowAtIndexPath:indexPath];
    newCell.accessoryType = UITableViewCellAccessoryCheckmark;

    UITableViewCell *oldCell = [tableView
                      cellForRowAtIndexPath:_selectedIndexPath];
    oldCell.accessoryType = UITableViewCellAccessoryNone;

    _selectedIndexPath = indexPath;
  }
}

@end
```

There's nothing much going on here. This screen is a table view controller, so it has a table view data source and delegate. The data source gets its rows from the `_categories` instance variable, an `NSArray` that you fill up in `viewDidLoad`.

The only special thing is the `_selectedIndexPath` instance variable. When the screen opens it shows a checkmark next to the currently selected category. When the user taps another row, you want to remove the checkmark from the previously selected cell and put it in the new cell. In order to be able to do that, you need to know which row is the currently selected one.

The caller of this screen fills in the `selectedCategoryName` property, but that is a name (a string), not a row number. Therefore, in `cellForRowAtIndexPath` you compare the name of the row to `self.selectedCategoryName`. If they match, you store the row's index-path in the `_selectedIndexPath` variable. That way you can remove the checkmark for this row later in `didSelectRowAtIndexPath`. It's a bit of work for such a small feature, but in a good app it's the details that matter.

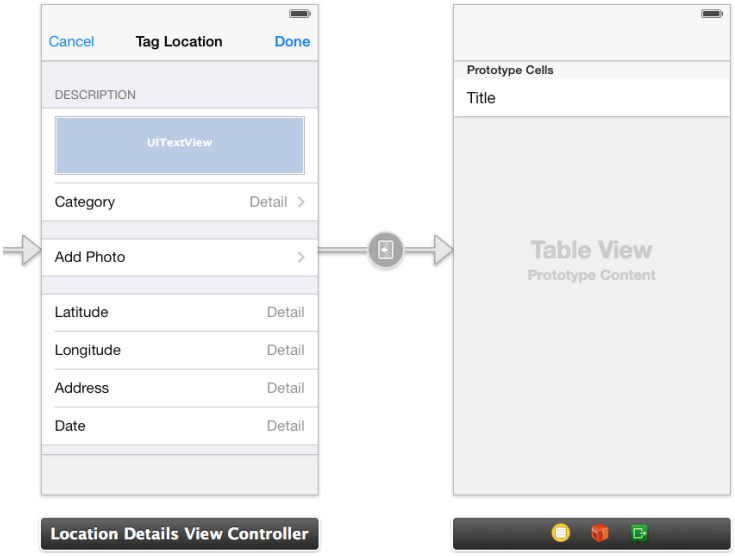
› Open the storyboard and drag a new **Table View Controller** into the canvas. Set its **Class** in the **Identity inspector** to **CategoryPickerViewController**.

› Change the **Style** of the prototype cell to **Basic**, and give it the re-use identifier **Cell**.

› **Ctrl-drag** from the Category cell on the Location Details View Controller to this new view controller and add a **Selection Segue - push** segue.

➤ Give the segue the identifier **PickCategory**.

The Category Picker View Controller now has a navigation bar at the top. You could change its title to "Choose Category", but as of iOS 7 Apple recommends that you do not to give view controllers a title if their purpose is obvious. This helps to keep the navigation bar uncluttered.



The category picker in the storyboard

That's enough for the storyboard. Now all that remains is to handle the segue.

➤ Add an import at the top **LocationDetailsViewController.m**:

```
#import "CategoryPickerViewController.h"
```

➤ Add a new instance variable called `_categoryName`. You'll use this to temporarily store the chosen category.

The instance variable section should now look like this:

```
@implementation LocationDetailsViewController
{
  NSString *_descriptionText;
  NSString *_categoryName;
}
```

➤ Change `initWithCoder:` to:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    _descriptionText = @"";
    _categoryName = @"No Category";
```

```
  }
  return self;
}
```

Initially you set the category name to "No Category". That is also the category at the top of the list in the category picker.

➤ Change `viewDidLoad` to put `_categoryName` into the label:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  self.descriptionTextView.text = _descriptionText;
  self.categoryLabel.text = _categoryName;

  . . .
```

➤ Finally, add the `prepareForSegue` method, below `cancel`:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"PickCategory"]) {
    CategoryPickerViewController *controller =
                                  segue.destinationViewController;
    controller.selectedCategoryName = _categoryName;
  }
}
```

This simply sets the `selectedCategoryName` property of the category picker. And with that, the app has categories.

➤ Run the app and play with the category picker.

Hmm, it doesn't seem to work very well. You can choose a category but the screen doesn't close when you tap a row. When you press the Back button, the category you picked isn't shown on the screen.

Exercise. Which piece of the puzzle is missing? □

Answer: The `CategoryPickerViewController` currently does not have a way to communicate back to the `LocationDetailsViewController` that the user picked a new category.

I hope that at this point you're thinking, "Of course, dummy! You forgot to give the category picker a delegate protocol. That's why it cannot send any messages to the other view controller." (If so, awesome! You're getting the hang of this.)

A delegate protocol is a fine solution indeed, but I want to show you a new storyboarding feature that can accomplish the same thing with less work: **unwind segues**. In case you were wondering, that's what the green "Exit" icons in the storyboard are for. Where a regular segue is used to open a new screen, an unwind segue closes the active screen. Sounds simple enough, however, making unwind segues is not very intuitive.

The green Exit icons don't appear to do anything. Try Ctrl-dragging from the prototype cell to the Exit icon, for example. It won't let you make a connection. First you have to add a special type of action method to the code.

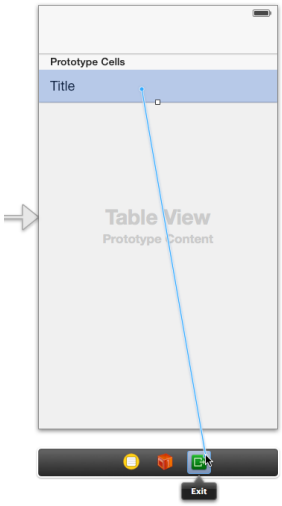
➤ In **LocationDetailsViewController.m**, add the following method:

```
- (IBAction)categoryPickerDidPickCategory:
                                        (UIStoryboardSegue *)segue
{
  CategoryPickerViewController *viewController =
                                       segue.sourceViewController;
  _categoryName = viewController.selectedCategoryName;
  self.categoryLabel.text = _categoryName;
}
```

This is an action method because it has the `IBAction` return type. What's different from a regular action method is the parameter, a `UIStoryboardSegue` object. Normally the parameter of an action method points to the control that triggered the action, such as a button. But in order to make an unwind segue you need to define an action method that takes a `UIStoryboardSegue` parameter.

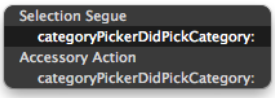
What happens inside the method is pretty straightforward. You look at the view controller that sent the segue, which of course is the `CategoryPickerViewController`, and then read the value of its `selectedCategoryName` property. Presumably that property contains the category that the user picked.

➤ Open the storyboard. **Ctrl-drag** from the prototype cell to the Exit button. This time it does allow you to make a connection:



Ctrl-dragging to the Exit icon to make an unwind segue

From the popup menu, choose **Selection Segue - categoryPickerDidPickCategory:**, the name of the unwind action method you just added.



Now when you tap a cell in the category picker, the screen will close and this new method is called.

➤ Try it out. That was easy! Unfortunately, the chosen category is ignored… That's because `categoryPickerDidPickCategory:` looks at the `selectedCategoryName` property, but that property isn't filled in anywhere yet.

You need some kind of mechanism that is invoked when the unwind segue is triggered, at which point you can fill in the `selectedCategoryName` based on the row that was tapped. What might such a mechanism be called? `prepareForSegue`, of course! This works for segues in both directions.

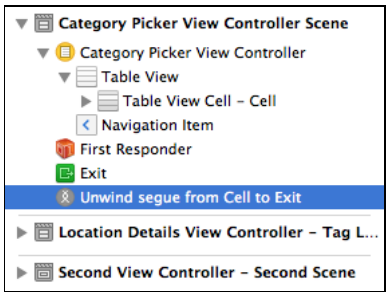
➤ Add the following method to **CategoryPickerViewController.m**:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"PickedCategory"]) {
    UITableViewCell *cell = sender;
    NSIndexPath *indexPath = [self.tableView
                                        indexPathForCell:cell];
    self.selectedCategoryName = _categories[indexPath.row];
  }
```

```
}
```

This looks at the selected index-path and then puts the corresponding category name into the `selectedCategoryName` property. This logic assumes the segue is named "PickedCategory", so you still have to set an identifier on the unwind segue.

Unfortunately, there is no visual representation of the unwind segue in the storyboard. There is no nice, big arrow that you can click on. To select the unwind segue you have to locate it in the document outline pane:



You can find unwind segues in the outline pane

➤ Select the unwind segue and go to the **Attributes inspector**. Give it the identifier **PickedCategory**.

➤ Run the app. Now the category picker should work properly. As soon as you tap the name of a category, the screen closes and the new category name is displayed.

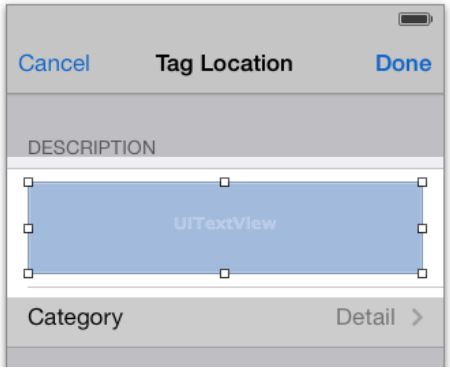
Unwind segues are pretty cool and are often easier than using a delegate protocol, especially for simply picker screens such as this one.

You can find the project files for the app up to this point under **02 - Tagging Locations** in the tutorial's Source Code folder.

Improving the user experience

The Tag Location screen is functional but it could do with some polish. These are the small details that will make your apps a delight to use and stand out from the competition.

Take a look at the design of the cell with the Description text view:



There is a margin between the text view and the cell border

There is 10-point margin between the text view and the cell border, but because the background of both the cell and the text view are white the user cannot see where the text view begins.

It is possible to tap inside the cell but just outside the text view. That is annoying when you want to start typing: you think that you're tapping in the text view but the keyboard doesn't appear. There is no feedback to the user that he's actually tapping outside the text view, and he will think your app is broken. In my opinion, rightly so.

You'll have to make the app a little more forgiving. When the user taps anywhere inside that first cell, the app should activate the text view, even the tap wasn't on the text view itself.

▸ Add the following two methods in the `#pragma UITableViewDelegate` section in **LocationDetailsViewController.m**:

```
- (NSIndexPath *)tableView:(UITableView *)tableView
            willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (indexPath.section == 0 || indexPath.section == 1) {
    return indexPath;
  } else {
    return nil;
  }
}

- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (indexPath.section == 0 && indexPath.row == 0) {
    [self.descriptionTextView becomeFirstResponder];
  }
}
```

The `willSelectRowAtIndexPath` method limits taps on rows to just the cells from the first two sections. The third section only has read-only labels anyway, so it doesn't need to allow taps.

The `didSelectRowAtIndexPath` handles the actual taps on the rows. You don't need to respond to taps on the Category or Add Photo rows as these cells are connected to segues. But if the user tapped in the first row of the first section – the row with the description text view – then this will give the input focus to that text view.

➤ Try it out. Run the app and click somewhere along the edges of the first cell. Anywhere you click inside that first cell should now bring up the keyboard.

Anything you can do to make screens less frustrating to use is worth putting in the effort.

Speaking of the text view, once you've activated it there's no way to get rid of the keyboard again. Because the keyboard takes up half of the screen that can be a bit annoying. It would be nice if the keyboard disappeared after you tapped anywhere else on the screen. As it happens, that is not so hard to add.

➤ Add the following to the end of `viewDidLoad`:

```
UITapGestureRecognizer *gestureRecognizer =
    [[UITapGestureRecognizer alloc]
        initWithTarget:self action:@selector(hideKeyboard:)];

gestureRecognizer.cancelsTouchesInView = NO;
[self.tableView addGestureRecognizer:gestureRecognizer];
```

A **gesture recognizer** is a very handy object that can recognize touches and other finger movements. You simply create the gesture recognizer object, give it a method to call when that particular gesture has been observed to take place, and add the recognizer object to the view. You're using a `UITapGestureRecognizer`, which recognizes simple taps, but there are several others for swipes, pans, pinches and much more.

Notice the `@selector` keyword again:

```
. . . initWithTarget:self action:@selector(hideKeyboard:) . . .
```

Using this syntax you can tell the `UITapGestureRecognizer` to call the method specified by the `@selector()` at a later time. This pattern is known as **target-action** and you've been using it all the time whenever you've connected `UIButtons`, `UIBarButtonItems`, and other controls to action methods. The target is the object that the message should be sent to, which is often `self`, and action is the message to send.

Here you've chosen the message `hideKeyboard:` to be sent when a tap is recognized anywhere in the table view, so you also have to implement that method.

➤ Add the `hideKeyboard:` method to the Location Details View Controller. It doesn't really matter where you put it, but below `viewDidLoad` is a good place:

```
- (void)hideKeyboard:(UIGestureRecognizer *)gestureRecognizer
{
  CGPoint point = [gestureRecognizer
                                  locationInView:self.tableView];

  NSIndexPath *indexPath = [self.tableView
                                   indexPathForRowAtPoint:point];

  if (indexPath != nil && indexPath.section == 0 &&
      indexPath.row == 0) {
    return;
  }

  [self.descriptionTextView resignFirstResponder];
}
```

Whenever the user taps somewhere in the table view, the gesture recognizer calls this method. It also passes a reference to itself as the parameter, which is handy because now you can ask `gestureRecognizer` where the tap happened.

`CGPoint` is another common struct that you see all the time in UIKit. It contains two fields, `x` and `y`, that describe a position on the screen. Using this `CGPoint`, you ask the table view which index-path is currently displayed at that position. This is important because you obviously don't want to hide the keyboard if the user tapped in the row with the description text view! If the user tapped anywhere else, you do hide the keyboard.

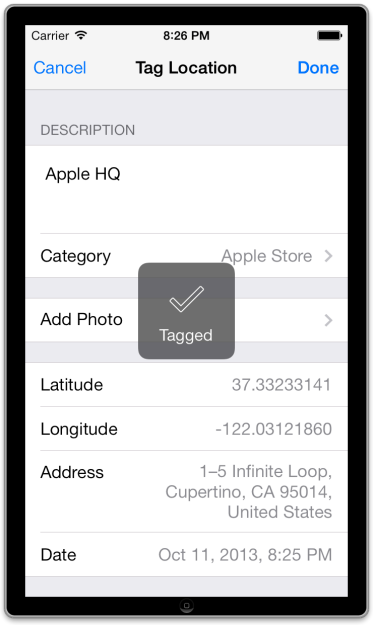
➤ Run the app. Tap in the text view to bring up the keyboard. Tap anywhere else in the table view to hide it.

As of iOS 7, the table view can automatically dismiss the keyboard when the user starts scrolling. You can enable this in the storyboard.

➤ Open the storyboard and select the table view with the static cells. In the **Attributes inspector** and change the **Keyboard** option from **Do not dismiss** to any of the others. Now scrolling should also hide the keyboard.

The HUD

There is one more improvement I wish to make to this screen, just to add a little spice. When you tap the Done button to close the screen, the app will show a quick animation to let the user know it successfully saved the location:



Before you close the screen it shows an animated checkmark

This type of overlay graphic is often called a HUD, for Heads-Up Display. Apps aren't quite fighter jets, but often HUDs are used to display a progress bar or spinner while files are downloading or another long-lasting task is taking place. You'll show your own HUD view for a brief second before the screen closes. It just adds a little extra liveliness to the app.

If you're wondering how you can display anything on top of a table, this HUD is simply a `UIView` subclass. You can add views on top of other views. That's what you've been doing all along: the labels are views that are added on top of the cells, which are also views. The cells themselves are added on top of the table view, and the table view in turn is added on top of the navigation controller's content view.

So far, when you've made your own objects, they have always been view controllers or data model objects, but it's also possible to make your own views. Often using the standard buttons and labels is sufficient, but when you want to do something that is not available as a standard view you can always make your own. You either subclass `UIView` or `UIControl` and do your own drawing. That's what you're going to do for the HUD view as well.

First let's add code to call the HUD, then that is out of the way.

➤ Change the `done:` method from **LocationDetailsViewController.m** to:

```
- (IBAction)done:(id)sender
{
  HudView *hudView = [HudView
        hudInView:self.navigationController.view animated:YES];
```

```
  hudView.text = @"Tagged";
}
```

This creates a `HudView` object and adds it to the navigation controller's view with an animation. You still have to write all that code as that's not something that regular `UIViews` can already do. You also set a `text` property on the new object.

Previously, `done:` called the `closeScreen` method to dismiss the view controller. However, for testing purposes you're not going to do that. You want to have time to see what the `HudView` looks like and if you immediately close the screen after showing the HUD, then it will be hard to see what's going on. You'll put the code to close the screen back later.

➤ Of course, you also need an import. Add the following line to the top of the source file:

```
#import "HudView.h"
```

Xcode complains about these latest additions because the project does not include the `HudView` source files yet. Let's make them.

➤ Add a new file to the project using the **Objective-C class** template. Make it a subclass of `UIView` named **HudView**.

➤ Replace the contents of **HudView.h** with the following:

```
@interface HudView : UIView

+ (instancetype)hudInView:(UIView *)view
                 animated:(BOOL)animated;

@property (nonatomic, strong) NSString *text;

@end
```

The interface for the `HudView` class just contains the signature for the method that you called earlier and the `text` property.

Let's build a minimal version of this class just so that you can get something on the screen. When that works, you'll make it look fancy.

➤ Replace the contents of **HudView.m** with the following:

```
#import "HudView.h"

@implementation HudView

+ (instancetype)hudInView:(UIView *)view animated:(BOOL)animated
```

```
{
  HudView *hudView = [[HudView alloc]
                                  initWithFrame:view.bounds];
  hudView.opaque = NO;

  [view addSubview:hudView];
  view.userInteractionEnabled = NO;

  hudView.backgroundColor = [UIColor colorWithRed:1.0f green:0
                                             blue:0 alpha:0.5f];

  return hudView;
}

@end
```

The `hudInView:animated:` method is a convenience constructor. It creates and returns a new `HudView` instance. A convenience constructor is always a **class method**, i.e. a method that works on the class as a whole and not on any particular instance. You can tell because its name begins with a `+` instead of the regular `–`. (The return type of a convenience constructor is `instancetype`, which is a special Objective-C keyword.)

When you call `[HudView hudInView:someView animated:YES]` you don't have an instance of `HudView` yet. In this case, the first word in the `[ ]` brackets is not the name of an instance variable or local variable that contains a `HudView` object, but the name of the `HudView` class itself. The whole purpose of this method is to create an instance of the HUD view for you, so you don't have to do that yourself, and to place it on top of another view.

You can see that making an instance is actually the first thing this method does:

```
HudView *hudView = [[HudView alloc] initWithFrame:view.bounds];

. . .

return hudView;
```

It does `alloc` and `init`, the steps to make a new object. At the end of the method that new instance is returned to the caller.

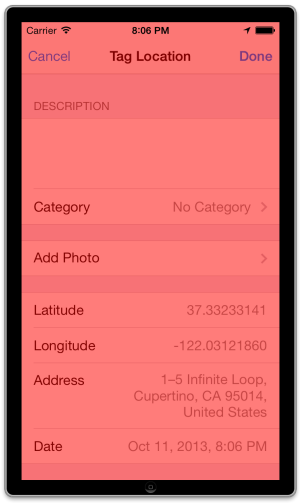
So why use this convenience constructor? As the name says, for convenience. There are more steps needed than just initializing the view, and by putting these in the convenience constructor, the caller doesn't have to worry about any of this.

One of these additional steps is that this method adds the new `HudView` object as a subview on top of the `view` object. This is actually the navigation controller's view so the HUD will cover the entire screen.

It also sets the `view`'s `userInteractionEnabled` property to `NO`. While the HUD is showing you don't want the user to interact with the screen anymore. The user has already pressed the Done button and the screen is in the process of closing. Most users will leave the screen alone at this point but there's always some joker who wants to try and break things. By setting `userInteractionEnabled` to `NO`, the view eats up any touches and all the underlying views become unresponsive.

Just for testing, the background color of the HUD view is 50% transparent red. That way you can see it covers the entire screen. By the way, whenever you see an `f` behind a literal number, as in `1.0f`, that just means this number is a `float`.

➤ Run the app. When you press the Done button, the screen will look like this:



The HUD view covers the whole screen

Often when you're working with views it's a good idea to set the background color to a bright color such as red or blue, so you can see exactly how big the view is.

➤ Remove the `backgroundColor` line from the `hudInView:animated:` method.

➤ Add the following method to **HudView.m**:

```
- (void)drawRect:(CGRect)rect
{
  const CGFloat boxWidth = 96.0f;
  const CGFloat boxHeight = 96.0f;

  CGRect boxRect = CGRectMake(
    roundf(self.bounds.size.width - boxWidth) / 2.0f,
    roundf(self.bounds.size.height - boxHeight) / 2.0f,
    boxWidth,
    boxHeight);

  UIBezierPath *roundedRect = [UIBezierPath
```

```
            bezierPathWithRoundedRect:boxRect cornerRadius:10.0f];
    [[UIColor colorWithWhite:0.3f alpha:0.8f] setFill];
    [roundedRect fill];
  }
```

The `drawRect` method is invoked whenever UIKit wants your view to redraw itself. Recall that everything in iOS is event-driven. You don't draw anything on the screen unless UIKit sends you the `drawRect` event. That means you should never call `drawRect` yourself.

If you want to force your view to redraw then you send the `setNeedsDisplay` message. UIKit will then trigger a `drawRect` event when it is ready to perform the drawing. This may seem strange if you're coming from another platform. You may be used to redraw the screen whenever you feel like it, but in iOS UIKit is in charge of who gets to draw when.

The above code draws a filled rectangle with rounded corners in the center of the screen. The rectangle is 96 by 96 points big (so I suppose it's really a square):

```
    const CGFloat boxWidth = 96.0f;
    const CGFloat boxHeight = 96.0f;
```

You haven't seen the `const` keyword before, but it means that `boxWidth` and `boxHeight` are really constants and not variables. Once you've given them a value it can never change. You use `const` to give a symbolic name to a numeric value. In the calculations that follow I find it clearer to refer to `boxWidth` than the number 96. That number doesn't mean much to me, but "box width" is a pretty clear description of its purpose.

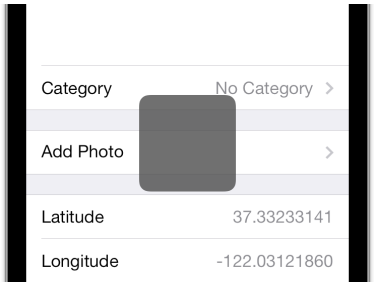
```
    CGRect boxRect = CGRectMake(
      roundf(self.bounds.size.width - boxWidth) / 2.0f,
      roundf(self.bounds.size.height - boxHeight) / 2.0f,
      boxWidth,
      boxHeight);
```

There is `CGRect` again. You use it to calculate the position for the rectangle that you'll be drawing. It should be centered horizontally and vertically in the screen. The size of the `HudView` is given by `self.bounds.size`. The above calculation uses the `roundf()` function to make sure the rectangle doesn't end up on fractional pixel boundaries because that makes the image look fuzzy.

```
    UIBezierPath *roundedRect = [UIBezierPath
            bezierPathWithRoundedRect:boxRect cornerRadius:10.0f];
    [[UIColor colorWithWhite:0.3f alpha:0.8f] setFill];
    [roundedRect fill];
```

The `UIBezierPath` is a very handy object for drawing rectangles with rounded corners. You just tell it how large the rectangle is and how round the corners should be. Then you fill it with an 80% opaque dark gray color.

The result looks like this:



The HUD view has a partially transparent background

There are two more things to add to the HUD, a checkmark and a text label. The checkmark is an image.

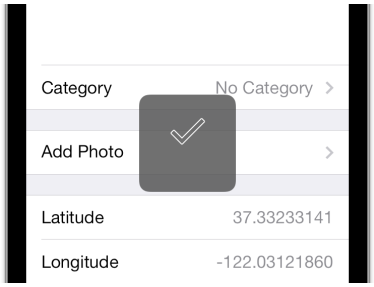
➤ The Resources folder for this tutorial has two files in the **Hud Images** folder, **Checkmark.png** and **Checkmark@2x.png**. Add these files to the asset catalog, **Images.xcassets**. You can do this with the + button or simply drag them from Finder into the Xcode window with the asset catalog open. (The images are white so they are a bit hard to see inside the asset list.)

➤ Add the following code to `drawRect:`

```
UIImage *image = [UIImage imageNamed:@"Checkmark"];

CGPoint imagePoint = CGPointMake(
  self.center.x - roundf(image.size.width / 2.0f),
  self.center.y - roundf(image.size.height / 2.0f)
                                        - boxHeight / 8.0f);

[image drawAtPoint:imagePoint];
```

This loads the checkmark image into a `UIImage` object. Then it calculates the position for that image based on the center coordinate of the HUD view (`self.center`) and the dimensions of the image (`image.size`). Finally, it draws the image at that position.

The HUD view with the image:



The HUD view with the checkmark image

Usually to draw text in your own view you'd add a `UILabel` object as a subview and let `UILabel` do all the hard work. However, for a view as simple as this you can also do your own text drawing.

➤ Complete `drawRect:` by adding the following code:

```
NSDictionary *attributes = @{
  NSFontAttributeName : [UIFont systemFontOfSize:16.0f],
  NSForegroundColorAttributeName : [UIColor whiteColor]
};

CGSize textSize = [self.text sizeWithAttributes:attributes];

CGPoint textPoint = CGPointMake(
  self.center.x - roundf(textSize.width / 2.0f),
  self.center.y - roundf(textSize.height / 2.0f)
                                          + boxHeight / 4.0f);

[self.text drawAtPoint:textPoint withAttributes:attributes];
```

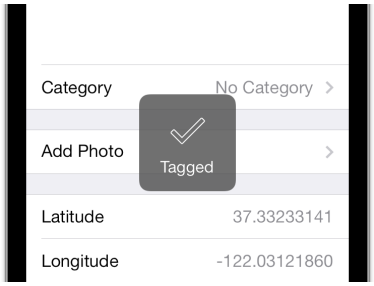
When drawing text you first need to know how big the text is, so you can figure out where to draw it. `NSString` has a bunch of handy methods for doing both.

First, you create the `UIFont` object that you'll use for the text. This is a "System" font of size 16. The system font on iOS 7 is Helvetica Neue. You also choose a color for text, plain white.

You use the font and the string from the `self.text` property to calculate how wide and tall the text will be. The result ends up in a `CGSize` variable. This is another struct. `CGPoint`, `CGSize` and `CGRect` are datatypes you use a lot when making your own views.

Finally, you calculate where to draw the text and then draw it. Quite simple, really.

➤ Run the app to try it out.



The HUD view with the checkmark and the text

OK, this shows a rounded box with a checkmark, but it's still far from spectacular. Time to liven it up a little with an animation. You've already seen a bit about animations before – they're really easy to add.

➤ Add the `showAnimated:` method to **HudView.m**:

```
- (void)showAnimated:(BOOL)animated
{
  if (animated) {
    // 1
    self.alpha = 0.0f;
    self.transform = CGAffineTransformMakeScale(1.3f, 1.3f);

    // 2
    [UIView animateWithDuration:0.3 animations:^{

      // 3
      self.alpha = 1.0f;
      self.transform = CGAffineTransformIdentity;
    }];
  }
}
```

In tutorial 1 you made a crossfade animation using the Core Animation framework. `UIView`, however, has its own animation mechanism. That still uses Core Animation behind the scenes, but it's a little more convenient to use.

The standard steps for doing UIView-based animations are as follows:

1. Set up the initial state of the view before the animation starts. Here you set `alpha` to 0, which means the view is fully transparent. You also set the `transform` to a scale factor of 1.3. We're not going to go into depth on transforms here, but basically this means the view is initially stretched out.
2. Call `[UIView animateWithDuration:. . .]` to set up an animation block.
3. Inside the block, set up the new state of the view that it should have after the animation completes. You set `alpha` to 1.0, which means it is now fully opaque.

You also set the `transform` to the "identity" transform, which means the scale is back to normal.

The animation will animate the properties that you changed from their initial state to the final state. The HUD view will quickly fade in as its opacity goes from fully transparent to fully opaque, and it will scale down from 1.3 times its original size to its regular width and height.

This is only a simple animation but it looks quite smart.

▸ Change the `hudInView` convenience constructor to call the `showAnimated:` method just before it returns:

```
+ (instancetype)hudInView:(UIView *)view animated:(BOOL)animated
{
  . . .

  [hudView showAnimated:animated];
  return hudView;
}
```

▸ Run the app and marvel at the magic of `UIView` animation.

Back to `LocationDetailsViewController`. You still need to close the screen when the user taps Done.

▸ Change the `done:` action method to the following:

```
- (IBAction)done:(id)sender
{
  HudView *hudView = [HudView
         hudInView:self.navigationController.view animated:YES];
  hudView.text = @"Tagged";

  [self performSelector:@selector(closeScreen) withObject:nil
             afterDelay:0.6];
}
```

What's that? Why aren't you just calling `[self closeScreen]` after showing the HUD view?

You could, but that doesn't look very good as the screen already closes before the HUD is finished animating. By calling `performSelector:withObject:afterDelay:`, you schedule the `closeScreen` method to be called after 0.6 seconds.

I spent some time tweaking that number. The HUD view takes 0.3 seconds to fully fade in and then you wait another 0.3 seconds before the screen disappears. That felt right to me. You don't want to do it too quickly or the effect from showing the

HUD is lost, but you don't want it to take too long either or it will annoy the user. Animations shouldn't make the app more frustrating to use!

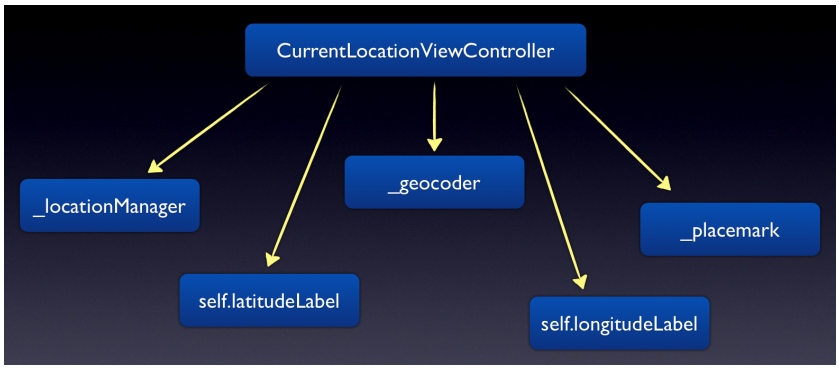
➤ Run the app. Press the Done button and watch how the screen disappears. This looks pretty smooth, if I do say so myself.

You can find the project files for this section under **03 - UI Improvements** in the tutorial's Source Code folder.

The object graph

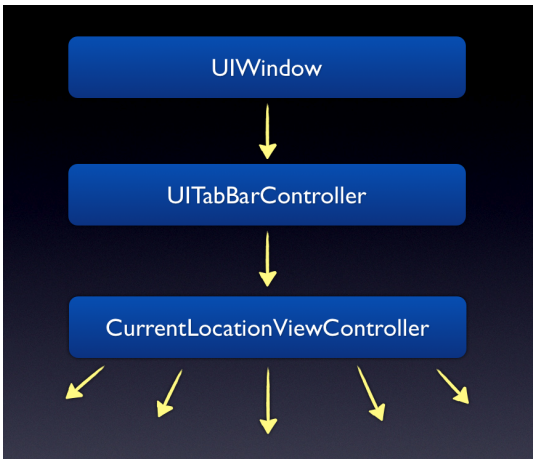
Objects are rarely hermits, off by themselves on a mountain somewhere. Your apps will have many objects that all need to work together. The relationships between these objects are described by the **object graph**.

For example, the `CurrentLocationViewController` has relationships with several objects:



Some of the objects that CurrentLocationViewController owns

These are its instance variables and its properties, in other words the objects it "owns". That's not the whole story. The `CurrentLocationViewController` itself is also owned by some object, as it belongs to a `UITabBarController`, which in turn belongs to the `UIWindow`.



The UITabBarController owns CurrentLocationViewController

This is only a small part of the object graph for the app. It shows the ownership relations between the various objects. (Don't confuse this diagram with the class hierarchy, which shows how the datatypes of the objects are related, but not the objects themselves.)

Object ownership is an important topic in iOS programming. You need to have a clear picture of which objects own what other objects, as the existence of those objects depends on it – and so does the proper functioning of your app! An object that no longer has any owners is immediately deallocated (destroyed!) and that's a problem if your app does not expect that to happen. On the other hand, if an object has too many owners it will stay in memory forever, which may cause your app to run out of free memory and crash.

There can be more than one owner of an object. For example, when the user taps the Tag Location button and the `CurrentLocationViewController` passes the `CLPlacemark` object to the `LocationDetailsViewController`, that view controller assumes shared ownership of it.

You declared that with the following statement in **LocationDetailsViewController.h**:

```
@property (nonatomic, strong) CLPlacemark *placemark;
```

This gives `LocationDetailsViewController` a new property, `placemark`, with a `strong` relationship. When you put an object into that property, the Tag Location screen becomes co-owner of that object.

There are two types of object relationships in iOS: **strong** and **weak**. In a strong relationship, an object assumes ownership of another, a responsibility that it can share with other owners. In a weak relationship, there is no such ownership.

You have used `weak` properties before, for example with outlet properties:

```
@property (nonatomic, weak) IBOutlet UILabel *latitudeLabel;
```

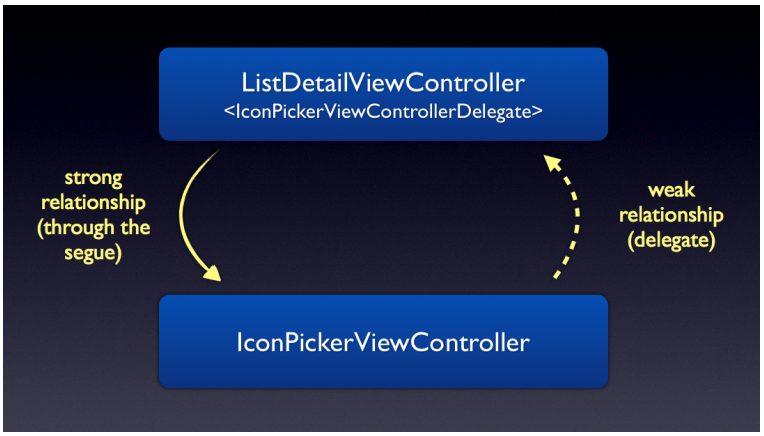
The reason outlet properties are weak is that the view controller isn't really their owner; instead, the outlets belong to the view controller's main view.

Weak is also commonly used with delegates:

```
@property (nonatomic, weak) id
                    <IconPickerViewControllerDelegate> delegate;
```

This is from the `IconPickerViewController` class from the Checklists tutorial. The icon picker has a delegate, an object that listens to events that may happen on the icon picker screen.

It is customary for an object to not own its delegate, because that could cause an **ownership cycle** – and you don't want two objects owning each other. By making the delegate relationship weak, this situation is avoided:



Delegate relationships are weak to avoid ownership cycles

Weak relationships have a cool side effect. If you have a weak reference to an object and that object is deallocated because its previous owners have all given up ownership, then your reference automatically becomes `nil`. That's a good thing too because if it didn't, you'd be pointing to a dead object and all sorts of mayhem could break loose if you tried to use it. (Remember that in Objective-C it's okay to send messages to `nil`, but not to objects that no longer exist.)

Life and death of objects

After you create an object, it stays alive while it has at least one owner. Objects keep track of this with a so-called **retain count**. The retain count increases when you give that object to someone else using a strong relationship, and it decreases when one of the owners stops using it. Therefore, the retain count represents the number of owners an object has. Weak relationships have no influence on the retain count.

You don't need to know anything about this retain count, except this: when the retain count becomes zero – i.e. when there are no more owners – the object is immediately deallocated and deleted. Gone. So the lifetime of an object is determined by how many owners it has at any given point.

Here are some examples of how this works:

```
- (void)greetPerson:(NSString *)name
{
  NSString *text = [NSString stringWithFormat:
                                        @"Hello, %@", name];
  NSLog(@"The string is: %@", text);
}
```

This creates an `NSString` object and stores it in a variable named `text`. Its retain count at that point is 1. This string object is valid until the end of the method, as `text` is a local variable and locals cease to exist when the method ends. At that point there is no more owner for the string, its retain count drops to 0, and the string object is deallocated.

In the next example, `_text` is an instance variable:

```
@implementation GreetingBot
{
  NSString *_text;
}

- (void)greetPerson:(NSString *)name
{
  _text = [NSString stringWithFormat:@"Hello, %@", name];
}

- (void)printText
{
  NSLog(@"The string is: %@", _text);
}
```

When `greetPerson` is called, it puts a new string object into the `_text` instance variable. When `greetPerson` ends, that string object stays alive because the lifetime of an instance variable is not bound to that of the method. You can call `printText` at any point afterward and it will display the string just fine.

The object from `_text` isn't deallocated until:

- The `GreetingBot` instance itself gets deallocated. When that happens it releases all the objects it owns. Since the string from `_text` has no other owner, it will be destroyed.
- You put a new value into `_text`, for example by calling `greetPerson` again.

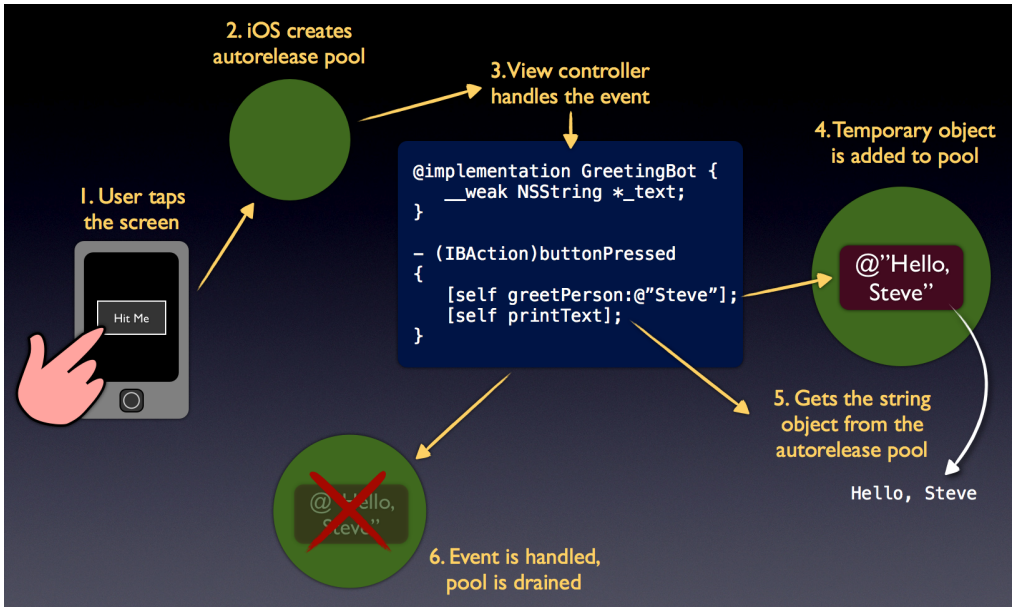
- You do `_text = nil`. That doesn't put a new object into the variable but it does release the existing one.

Instance variables are always considered strong relationships, unless you use the `__weak` symbol. You can change the string example to the following:

```
@implementation GreetingBot
{
  __weak NSString *_text;
}
```

Calling `printText` some time after `greetPerson` might print the string or it might print `(null)`, depending on exactly when you call it.

Whenever iOS handles a new event – caused by the user tapping on a button, a timer, or some other trigger – it first creates an **autorelease pool**. This pool becomes the temporary owner of all objects that don't have a strong relationship. After the event is over, the autorelease pool is drained and lets go of all its temporary objects. If you are not holding on to any of those objects, they are deallocated. But until that happens – i.e. while the current event is still being handled – the objects remain valid, even if they have no real owner.



The autorelease pool holds on to temporary objects for the duration of the event

Should you worry about any of this? Not really, but it's still good to know how it works. Just remember that if you want to keep an object around beyond the current event, you should store it in a strong relationship. If you don't and the object has no other owners, then the autorelease pool will dispose of it.

Copy and assign properties

Previously, you've declared properties as `weak` and `strong`:

```
@property (nonatomic, weak) IBOutlet UILabel *latitudeLabel;

@property (nonatomic, strong) CLPlacemark *placemark;
```

You can also declare them `assign` or `copy`.

By the way, the `nonatomic` keyword (and its opposite, `atomic`) are used in **multi-threading** code, a topic beyond this lesson. Most of the time, using `nonatomic` for your properties is the right choice.

You declare a property as `assign` if it has a primitive value:

```
@property (nonatomic, assign) int someNumber;
```

You also use `assign` for structs, because they aren't objects either:

```
@property (nonatomic, assign) CLLocationCoordinate2D coordinate;
```

Xcode will give the error message: "Property with 'retain (or strong)' attribute must be of object type" when you try to use the wrong qualifier.

When you declare a property as `copy`, whatever object you try to give that property is copied first and the copy is stored instead of the original:

```
@property (nonatomic, copy) NSString *text;
```

This is a strong relationship, but with the newly copied object.

Making a property `copy` is used most often with strings and arrays to ensure that you get a unique object that cannot be changed out from under you. Even though `NSString` and `NSArray` are immutable and cannot be changed once created, they still have mutable subclasses. Therefore, this is possible:

```
// on someObject:
@property (nonatomic, strong) NSString *text;

. . .

NSMutableString *m = @"Strawberry";
someObject.text = m;
// at this point, someObject.text is @"Strawberry"

[m appendString:@" and banana"];
// now both m and someObject.text are @"Strawberry and banana"
```

Because `someObject`'s `text` property is `strong` and not `copy`, both `someObject.text` and `m` refer to the same object. If `m` changes from `@"Strawberry"` to `@"Strawberry and banana"` then so will `someObject.text`. This is valid code because `NSMutableString` is a subclass of `NSString` and can therefore be used everywhere as if it were an `NSString` – but it can also do more and `someObject` may not be expecting that. By declaring the property `copy` instead, the value `@"Strawberry"` is copied into a new `NSString` (not a mutable one!) and any changes to `m` are not seen by `someObject`.

Sometimes you may still see a property declared as `retain`. That's how you would declare a strong relationship using manual memory management, but as of iOS 5 that is no longer necessary. In the next section on Core Data you'll have Xcode generate classes for your data model objects and it still puts the keyword `retain` in the property definitions. `retain` is simply a synonym for `strong`, and it acts in the exact same way.

Manual memory management

Before ARC (Automatic Reference Counting), which was introduced with iOS 5, you were responsible for explicitly taking and releasing ownership of objects. This is known as **manual memory management** and it was a pain in the butt.

If you did this manual retaining and releasing of objects wrong – and mistakes were very easy to make – then either you ended up with dead objects or with objects that would never be deallocated. The latter is called a **memory leak**. The app slowly "leaks" memory that cannot be reclaimed. If there are too many leaks in the app, at some point it will run out of available memory and crash.

These days it's harder to have memory leaks but not impossible. If you keep holding onto objects for longer than strictly necessary, then your app is using memory it should really release. Remember that objects that you have a strong relationship with will stay in memory forever, unless you replace them with other objects, set their pointers to `nil`, or the owning object gets destroyed.

Important: If you no longer need to use an object, then set its pointer to `nil`. This will deallocate the object (if you're the only owner) and return free memory to your app. But if you keep holding on to that object, the system can never reclaim its memory and you have created a leak.

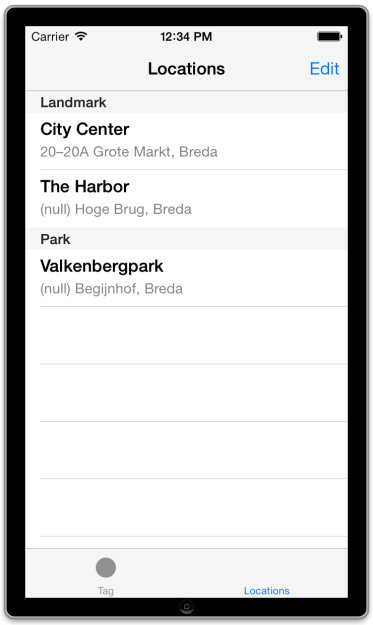
Even though you don't have to worry about manual memory management anymore, you do have to think about the relationships between your objects – are they `strong` or `weak` – and you must set the pointer to `nil` when you're done with an object so it won't stay in memory forever.

The next tutorial has a short explanation of how to do manual memory management because sometimes you still need it when you're dealing with older code or reading through older books and examples.

Storing the locations with Core Data

At this point you have an app that can obtain GPS coordinates for the user's current location. It also has a screen where the user can "tag" that location, which currently consists of entering a description and choosing a category. Later on, you'll also allow the user to pick a photo. The next feature is to make the app remember the locations that the user has tagged and show them in a list.

The Locations screen will look like this:



The Locations screen

You have to persist the data for these captured locations somehow, because they should be remembered even when the app terminates. The last time you did this, you made data model objects that conformed to the `NSCoding` protocol and saved them to a .plist file using `NSKeyedArchiver`. That works fine but in this lesson I want to introduce you to a new framework that can take a lot of work out of your hands: Core Data.

Core Data is an object persistence framework for iOS apps. The official documentation may be a little daunting but the principle is quite simple. You've learned that objects get destroyed when there are no more references to it. In addition, all objects get destroyed when the app terminates. With Core Data, you can designate some objects as being persistent so they will always be saved to a **data store**. Even when all references to such a **managed object** are gone and the instance gets destroyed, its data is still safely stored in Core Data and you can get it back at any time.

If you've worked with databases before, then you might be tempted to think of Core Data as a database but that's a little misleading. In some respects the two are

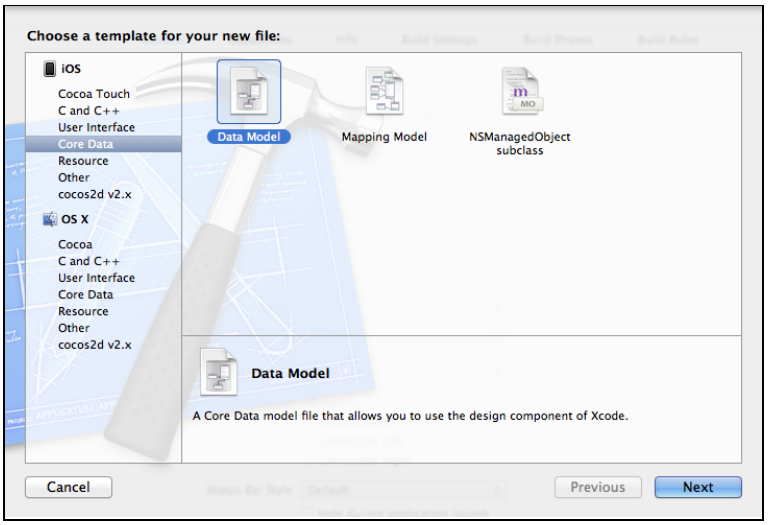
similar but Core Data is about storing objects, not relational tables. It is just another way to make sure the data from certain objects doesn't get deleted when these objects are deallocated or the app terminates.

Adding Core Data to the app

› Go to the **Linked Framework and Libraries** section in the **Project Settings** screen and add **CoreData.framework** to the project.

Core Data requires the use of a data model. This is a special file that you add to your project to describe the objects that you want to persist. These *managed* objects, unlike regular objects, will keep their data in the data store unless you explicitly delete them.

› Add a new file to the project. Choose the **Data Model** template under the **Core Data** section:

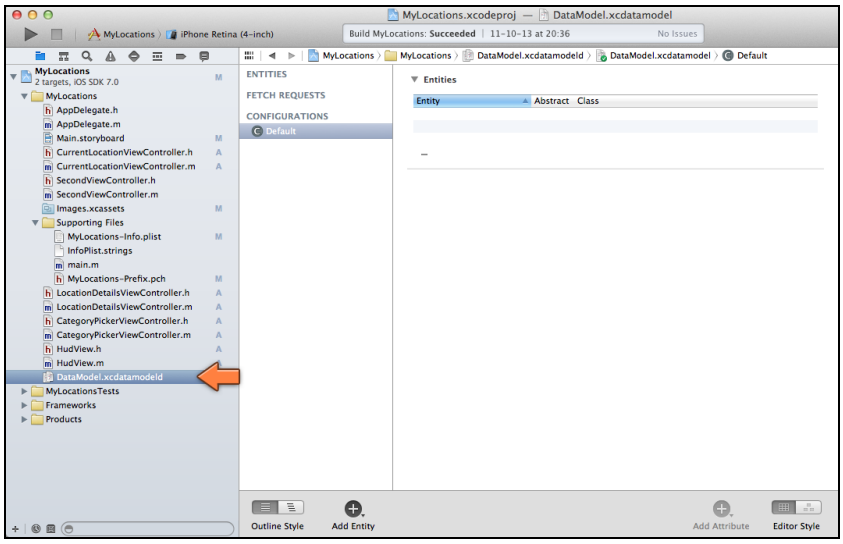


Adding a Data Model file to the project

› Save it as **DataModel**.

This will add a new file to the project, DataModel.xcdatamodeld.

› Click **DataModel.xcdatamodeld** to open the Data Model editor:



The empty data model

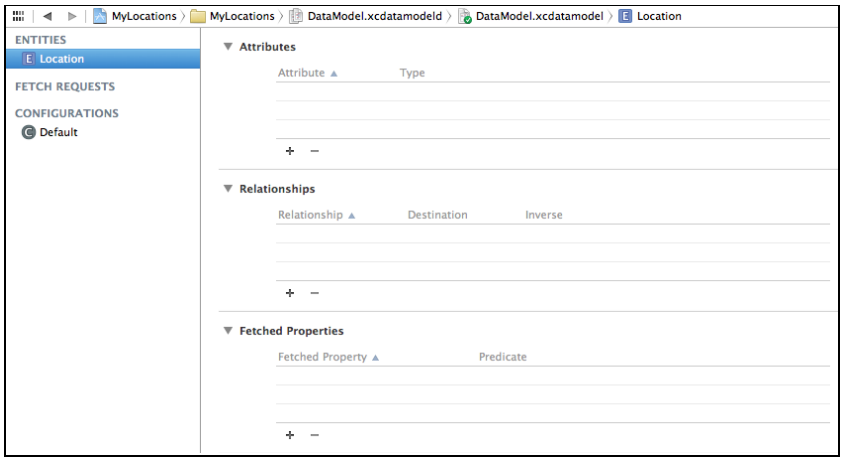
For each object that you want Core Data to manage, you have to add an **entity**. An entity describes which data fields your objects will have. In a sense it serves the same purpose as a class but specifically for Core Data's data store. (If you've worked with SQL databases before, you can think of an entity as a table.)

This app will only have one entity, Location, which stores all the properties for a location that the user tagged. Each Location will keep track of the following data:

- latitude and longitude
- placemark (the street address)
- the date when the location was tagged
- description
- category

These are the items from the Tag Location screen, except for the photo. Photos are potentially very big and can take up several megabytes of storage space. Even though the Core Data store can handle big "blobs" of data, it is usually better to store photos as separate files in the app's Documents directory. More about that later.

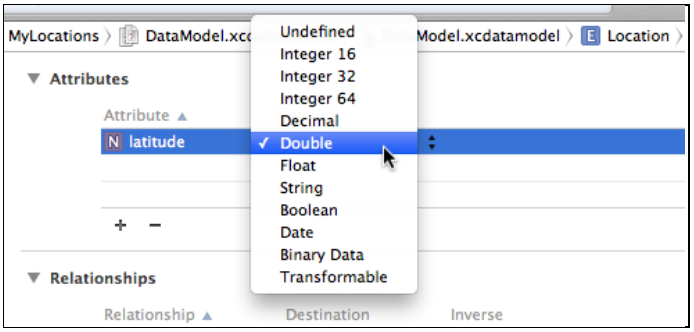
› Click the **Add Entity** button at the bottom of the data model editor. This adds a new entity under the ENTITIES heading. Name it **Location**. (You can rename the entity by clicking its name or from the Data Model inspector.)



The new Location entity

To the right there are three sections: Attributes, Relationships and Fetched Properties. The Attributes are the entity's data fields. This app only has one entity, but often apps will have many entities that are all related to each other somehow. With Relationships and Fetched Properties you can tell Core Data how your objects depend on each other. For this app you will only use the Attributes section.

› Click the **Add Attribute** button at the bottom of the editor (or the small + button below the Attributes section). Name the new attribute **latitude** and set its **Type** to **Double**:



Choosing the attribute type

Attributes are basically the same as instance variables and therefore they have a datatype. You've seen earlier that the latitude and longitude coordinates really have the datatype `double`, so that's what you're choosing for the attribute as well.

Don't let the change in terminology scare you. Just think:

- entity = object (or class)
- attribute = variable

> If you're wondering where you'll define methods in Core Data, then the answer is: you don't. Core Data is only for storing the data portion of objects. That is

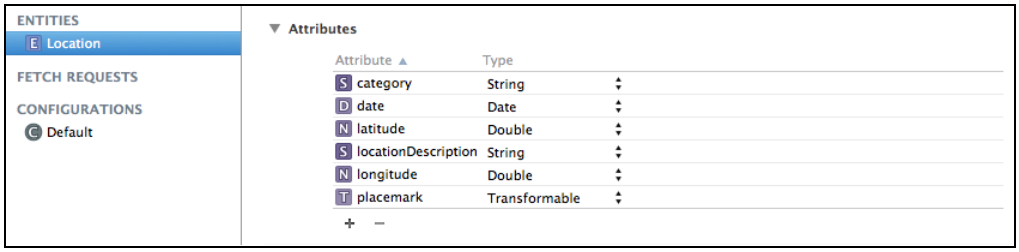
what an entity describes, the data of an object, and optionally how that object relates to other objects if you use Relationships and Fetched Properties.

You are still going to define your own `Location` class in code by creating a .h and .m file, just as you've been doing all along. Because it describes a managed object, this class will be associated with the Location entity in the data model. But it's still a regular class, so you can add your own methods to it.

› Add the other attributes to the Location entity:

- **longitude**, type Double
- **date**, type Date
- **locationDescription**, type String
- **category**, type String
- **placemark**, type Transformable

The data model should look like this when you're done:



All the attributes of the Location entity

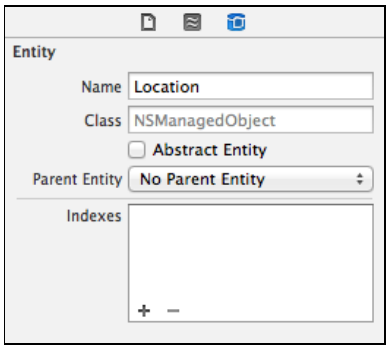
Why didn't you just call it "description" instead of "locationDescription"? As it turns out, `description` is the name of a method from `NSObject`. If you try to name an attribute "description", then this will cause a naming conflict with that method. Xcode will give you an error message if you try to do this.

The type of the placemark attribute is Transformable. Core Data only supports a limited number of datatypes right out the box, such as String, Integer 32 and Date. The placemark will be a `CLPlacemark` object and that is not in the list of supported datatypes.

Fortunately, Core Data has a provision for handling arbitrary datatypes. Any class that conforms to the `NSCoding` protocol can be stored in a Transformable attribute without additional work. Fortunately for us, `CLPlacemark` does conform to `NSCoding`, so you can store it in Core Data with no trouble.

There is one more change to make in the data model and then you're done here.

› Click on the Location entity to select it. In the inspectors panel, switch to the **Data Model inspector**.



The Data Model inspector

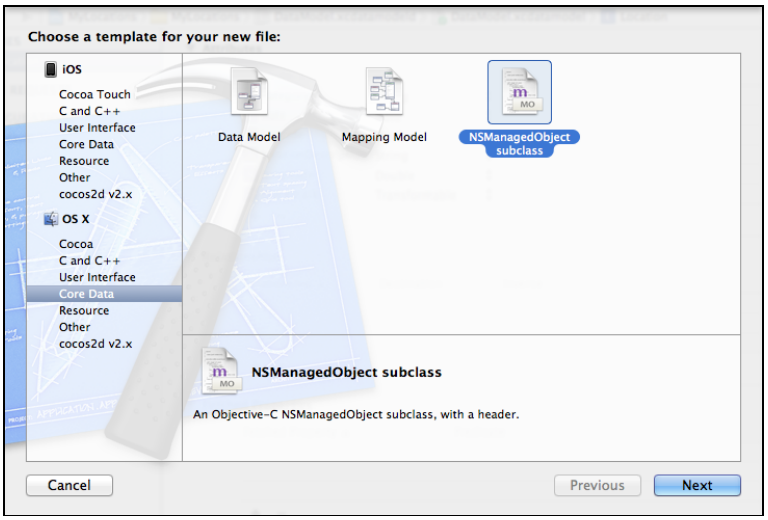
The Class field currently says "NSManagedObject". This means that when you retrieve a Location entity from Core Data, it gives you an object of the class `NSManagedObject`. That is the base class for all objects that are managed by Core Data. Regular objects inherit from `NSObject`, but objects from Core Data extend `NSManagedObject`. Because using `NSManagedObject` directly is a bit limiting, you are going to use your own class instead.

› Change the **Class** field to **Location**. That is the name of the new class you are going to make.

You're not required to make your own classes for your entities, but it makes Core Data easier to use. When you now retrieve a Location entity from the data store, Core Data doesn't give you an `NSManagedObject` but an instance of your own `Location` class.

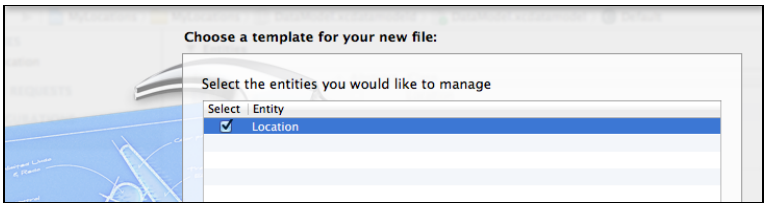
Xcode trick makes it easy to generate classes automatically from the data model.

› Add a new file to the project. From the **Core Data** section choose the **NSManagedObject subclass** template:



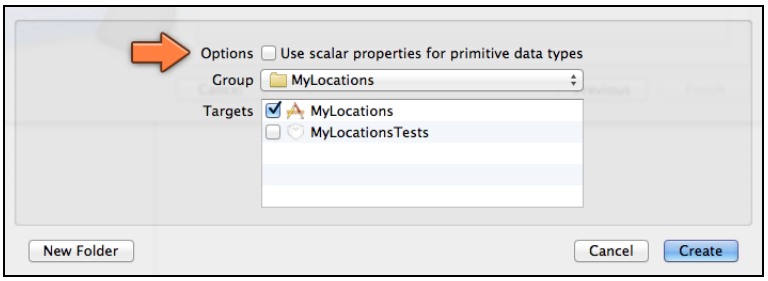
Choose the NSManagedObject subclass template

The assistant will now ask you for which data model and which entity you wish to create the class.



Select the Location entity

› Make sure **Location** is selected. When asked where to save the source files, make sure there is no checkmark in front of **Use scalar properties for primitive data types**:



The "Use scalar properties" option should be unchecked

This adds two files to the project, **Location.h** and **Location.m**. The header file looks something like this:

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

@interface Location : NSManagedObject

@property (nonatomic, retain) NSNumber * latitude;
@property (nonatomic, retain) NSNumber * longitude;
@property (nonatomic, retain) NSDate * date;
@property (nonatomic, retain) NSString * locationDescription;
@property (nonatomic, retain) NSString * category;
@property (nonatomic, retain) id placemark;

@end
```

As you can see in the `@interface` line, the `Location` class extends `NSManagedObject` instead of the regular `NSObject`. Xcode has also created properties for the attributes that you specified in the Data Model editor.

Even though you chose the datatype Double for latitude and longitude, these properties are listed as `NSNumber` objects instead of `double` values. Core Data stores

everything as objects, not as primitive values. Anything that you would normally use an `int`, `float`, `double` or `BOOL` for will need to become an `NSNumber` in Core Data.

Because you made placemark a Transformable property, Xcode doesn't really know what kind of object this will be, so it chose the generic datatype `id`. You know it's going to be a `CLPlacemark` object, so you can make things easier for yourself by changing it.

➤ Change the `placemark` property to:

```
@property (nonatomic, retain) CLPlacemark *placemark;
```

The **Location.m** file is very simple:

```
#import "Location.h"

@implementation Location

@dynamic latitude;
@dynamic longitude;
@dynamic date;
@dynamic locationDescription;
@dynamic category;
@dynamic placemark;

@end
```

Normally a property gets a backing instance variable to store its values. But because this is a *managed* object, and the data lives inside a data store, Core Data will handle the properties another way. The `@dynamic` keyword tells the compiler that these properties will be resolved at runtime by Core Data. When you put a new value into one of these properties, Core Data will put that value into the data store for safekeeping, instead of in an instance variable. That's all there is to it.

This concludes the definition of the data model for this app. Now you have to hook it up to a data store.

The data store

On iOS, Core Data stores all of its data into a SQLite database. It's OK if you have no idea what that is. You'll take a peek into that database later, but you don't really need to know what goes on inside the data store in order to use Core Data. However, you do need to initialize the data store when the app starts. This code is almost the same for just about any app that uses Core Data and it goes in the app delegate class.

The app delegate is the object that gets notifications that concern the application as a whole. This is where iOS notifies the app that it has started up, for example. You're going to make a few changes to the project's `AppDelegate` class.

➤ Add the following to **AppDelegate.m**, between the `#import` and `@implementation` lines:

```
@interface AppDelegate ()
@property (nonatomic, strong) NSManagedObjectContext
                                          *managedObjectContext;
@property (nonatomic, strong) NSManagedObjectModel
                                            *managedObjectModel;
@property (nonatomic, strong) NSPersistentStoreCoordinator
                                       *persistentStoreCoordinator;
@end
```

Remember that an `@interface` block inside a .m file is for a class extension. You added one before on `LocationDetailsViewController` for its outlet properties to keep them private. This time you're putting regular properties into the class extension; these properties are only meant to be used inside the **AppDelegate.m** file.

➤ Add the following code at the bottom of the source file, before `@end`:

```
#pragma mark - Core Data

- (NSManagedObjectModel *)managedObjectModel
{
  if (_managedObjectModel == nil) {
    NSString *modelPath = [[NSBundle mainBundle]
                   pathForResource:@"DataModel" ofType:@"momd"];
    NSURL *modelURL = [NSURL fileURLWithPath:modelPath];
    _managedObjectModel = [[NSManagedObjectModel alloc]
                                  initWithContentsOfURL:modelURL];
  }
  return _managedObjectModel;
}

- (NSString *)documentsDirectory
{
  NSArray *paths = NSSearchPathForDirectoriesInDomains(
                    NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths lastObject];
  return documentsDirectory;
}

- (NSString *)dataStorePath
```

```
{
  return [[self documentsDirectory]
            stringByAppendingPathComponent:@"DataStore.sqlite"];
}

- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
  if (_persistentStoreCoordinator == nil) {
    NSURL *storeURL = [NSURL fileURLWithPath:
                                          [self dataStorePath]];

    _persistentStoreCoordinator =
        [[NSPersistentStoreCoordinator alloc]
            initWithManagedObjectModel:self.managedObjectModel];

    NSError *error;
    if (![_persistentStoreCoordinator
             addPersistentStoreWithType:NSSQLiteStoreType
             configuration:nil URL:storeURL options:nil
             error:&error]) {
      NSLog(@"Error adding persistent store %@, %@", error,
                                              [error userInfo]);
      abort();
    }
  }
  return _persistentStoreCoordinator;
}

- (NSManagedObjectContext *)managedObjectContext
{
  if (_managedObjectContext == nil) {
    NSPersistentStoreCoordinator *coordinator =
                                self.persistentStoreCoordinator;
    if (coordinator != nil) {
      _managedObjectContext =
                           [[NSManagedObjectContext alloc] init];
      [_managedObjectContext
                     setPersistentStoreCoordinator:coordinator];
    }
  }
  return _managedObjectContext;
}
```

This is the code you need to load the data model that you've defined earlier, and to connect it to an SQLite data store. This is very standard stuff that will be the same for almost any Core Data app you'll write. I won't bore you with the details of how

it works just yet. The only part that you'll be using from now on is the `NSManagedObjectContext`.

Xcode now gives a bunch of error messages because you're using Core Data classes inside the App Delegate but you haven't imported the Core Data header files yet. Let's add those to the Prefix file so they are automatically available in each source file.

▸ Add the following line to **MyLocations-Prefix.pch**:

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

▸ Build the app to make sure it compiles. If you run it you won't notice any difference because you're not actually using Core Data anywhere yet.

Passing around the context

When the user presses the Done button in the Tag Location screen, the app currently just closes the screen. Let's improve on that so that pressing Done saves a new `Location` object into the Core Data store.

I mentioned the `NSManagedObjectContext` object before. This is the object that you use to talk to Core Data. It is often described as the "scratchpad". You first make your changes to the context and then you call its `save` method to store those changes permanently in the data store. That means every object that needs to do something with Core Data needs to have a reference to that `NSManagedObjectContext` object.

▸ Add a new property to **LocationDetailsViewController.h**:

```
@property (nonatomic, strong) NSManagedObjectContext
                                            *managedObjectContext;
```

The problem is: how do you put the `NSManagedObjectContext` into that property? The context object is created by `AppDelegate` (in that big block of code you just added) but `AppDelegate` has no reference to the `LocationDetailsViewController`. That's not so strange as this view controller doesn't exist until the user taps the Tag Location button. Prior to initiating that segue, there is no `LocationDetailsViewController` object.

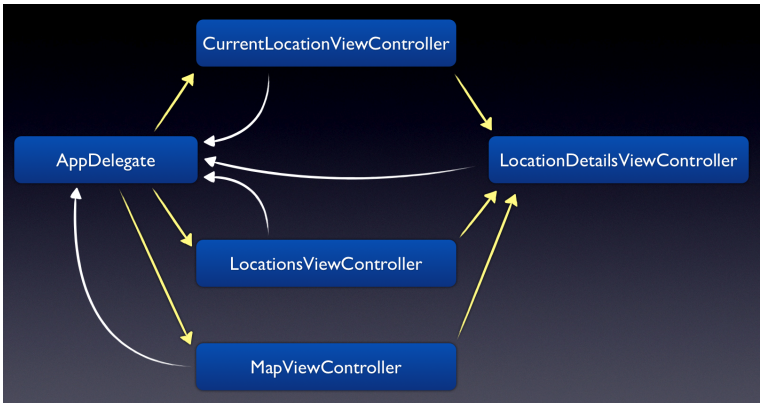
The answer is to set the `managedObjectContext` property in the `prepareForSegue` method from `CurrentLocationViewController` when it opens the Tag Location screen. Now you need to find a way to get the `NSManagedObjectContext` object into the `CurrentLocationViewController` in the first place.

I see a lot of code that does the following:

```
#import "AppDelegate.h"
```

```
. . .

AppDelegate *appDelegate = (AppDelegate *)[[NSApplication
                                    sharedApplication] delegate];

NSManagedObjectContext *context =
                                appDelegate.managedObjectContext;

// do something with the context
```

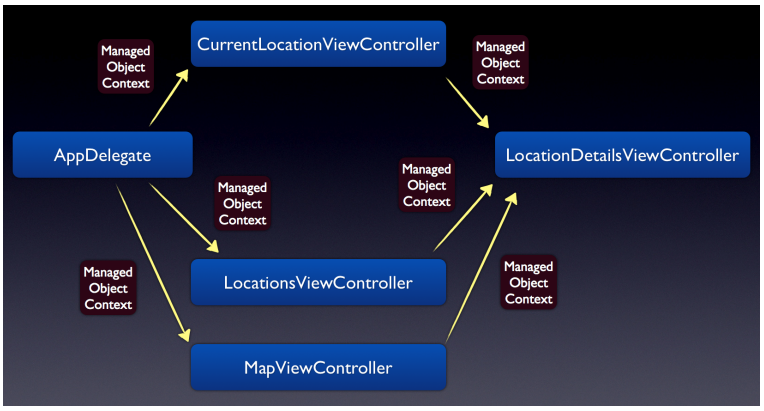
This depends on the `managedObjectContext` being a property on the app delegate (in its .h file). From anywhere in your source code you can get a reference to the context simply by asking the `AppDelegate`. Sounds like a good solution, right? Not quite… Suddenly all your objects are dependent on the app delegate. This introduces a dependency that can make your code messy really quickly.



Bad: All classes depend on AppDelegate

As a general design principle, it is best to make your classes depend on each other as little as possible. The fewer interactions there are between the different parts of your program, the simpler it is to understand. If many of your classes need to reach out to some shared object such as the app delegate, then you may want to rethink your design.

A better solution is to pass along the `NSManagedObjectContext` to each object that needs it.



Good: The context object is passed from one object to the next

Using this architecture, `AppDelegate` gives the managed object context to `CurrentLocationViewController`, which in turn will pass it to the `LocationDetailsViewController` when it performs the segue.

Note that you made the `managedObjectContext` property from `AppDelegate` private by putting it in the class extension instead of the `@interface` section. Inside **AppDelegate.m** you can freely use that property but other objects cannot ask `AppDelegate` for it. This is done on purpose to prevent other parts of the app from "abusing" the app delegate.

▸ In **AppDelegate.m**, add the following import:

```
#import "AppDelegate.h"
#import "CurrentLocationViewController.h"
```

▸ Change the `didFinishLaunchingWithOptions` method to:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  UITabBarController *tabBarController =
           (UITabBarController *)self.window.rootViewController;

  CurrentLocationViewController *currentLocationViewController
         = (CurrentLocationViewController *)
                           tabBarController.viewControllers[0];

  currentLocationViewController.managedObjectContext =
                                      self.managedObjectContext;

  return YES;
}
```

Unfortunately, Interface Builder does not allow you to make outlets for your view controllers on the App Delegate. Instead, you have to look up these view controllers by digging through the storyboard. So in order to get a reference to the `CurrentLocationViewController` you first have to find the `UITabBarController` and then look at its `viewControllers` array.

Once you have the `CurrentLocationViewController` object, you do:

```
currentLocationViewController.managedObjectContext =
                                    self.managedObjectContext;
```

This uses `self.managedObjectContext` to get a pointer to the App Delegate's `NSManagedObjectContext` object, even though you've never created that object anywhere yet. You can do this because you've declared `managedObjectContext` as a property in the class extension and because you've added the corresponding getter method for this property.

> **Note:** Remember that when you access a property through `self.propertyName`, what really happens behind the scenes is that the property's getter method is called. For a normal property the getter method simply returns the value of the backing instance variable, but if you provide your own getter method you can do other things as well.

Let's take a look at that getter method:

```
- (NSManagedObjectContext *)managedObjectContext
{
  if (_managedObjectContext == nil) {
    NSPersistentStoreCoordinator *coordinator =
                                self.persistentStoreCoordinator;

    if (coordinator != nil) {
      _managedObjectContext =
                          [[NSManagedObjectContext alloc] init];
      [_managedObjectContext
                     setPersistentStoreCoordinator:coordinator];
    }
  }
  return _managedObjectContext;
}
```

The details don't matter; I just want to illustrate the general principle. This method is called whenever you access `self.managedObjectContext`. The very first time this happens, the backing instance variable `_managedObjectContext` is `nil` (because that is the default value for all instance variables). In that case, you create a new `NSManagedObjectContext` object and put it in the variable. Now

`_managedObjectContext` is no longer `nil` and any subsequent calls to this method simply return the existing context object.

This is an example of **lazy loading**. The context object isn't created until you ask for it. Notice that this method also accesses the `self.persistentStoreCoordinator` property. That will lazy-load the persistent store coordinator, the object that handles the SQLite data store. In turn, the `persistentStoreCoordinator` getter method accesses the `self.managedObjectModel` property and that lazy-loads the data model. When these three things have been loaded, Core Data is ready for use.

Just by using the `self.managedObjectContext` property, you set off a chain of events that initializes the whole Core Data stack. That's the combined power of properties and lazy loading for ya!

Of course, `CurrentLocationViewController` still needs its own property for the `NSManagedObject` context.

➤ Add the following property to **CurrentLocationViewController.h**:

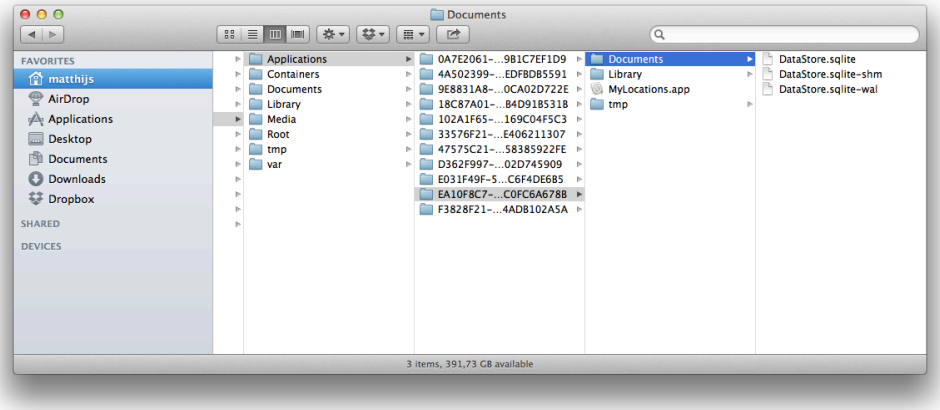
```
@property (nonatomic, strong) NSManagedObjectContext
                                            *managedObjectContext;
```

➤ Finally, you can pass the context on to the Tag Location screen:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"TagLocation"]) {
    UINavigationController *navigationController =
                                segue.destinationViewController;
    LocationDetailsViewController *controller =
              (LocationDetailsViewController *)
                          navigationController.topViewController;
    controller.coordinate = _location.coordinate;
    controller.placemark = _placemark;
    controller.managedObjectContext = self.managedObjectContext;
  }
}
```

➤ Run the app. Everything should still be the way it was, but behind the scenes a new database has been created because now you've truly used Core Data.

When you initialized the persistent store coordinator, you gave it a path to a database file. That file is named **DataStore.sqlite** and it lives in the app's Documents folder. You can see it in Finder if you go to **~/Library/Application Support/iPhone Simulator** and then to the folder that contains the MyLocations app:



The new database in the app's Documents directory

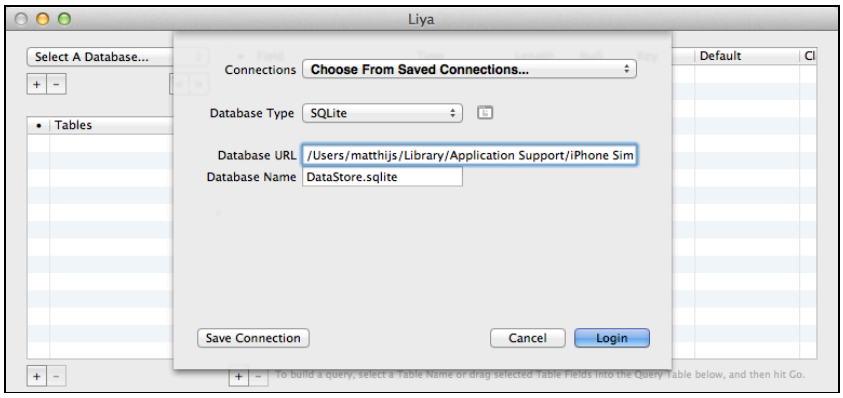
The **DataStore.sqlite-shm** and **-wal** files are also part of the data store.

> **Tip:** If you cannot find the Library folder in your home directory, open Terminal and type the following command:
>
> ```
> chflags nohidden ~/Library
> ```
>
> You can also open Finder's **Go** menu and hold down Alt/Option to reveal the Library folder.

This database is still empty because you haven't stored any objects inside it yet, but just for the fun of it you'll take a peek inside.

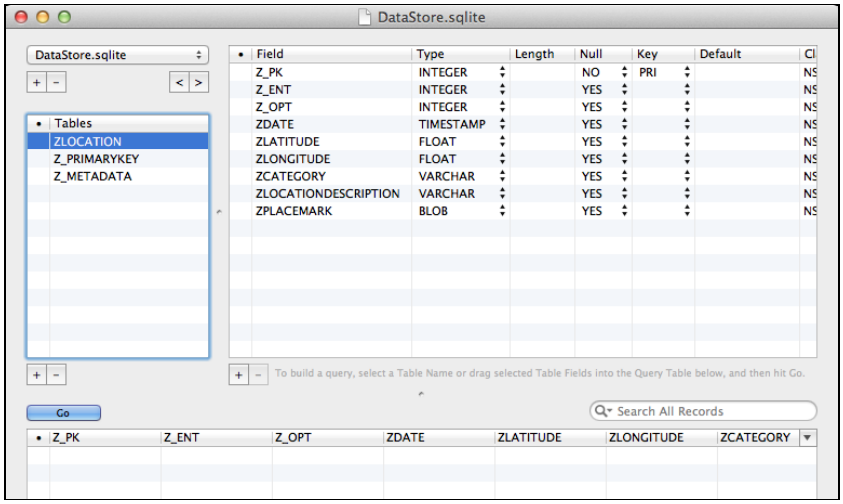
There are several handy (free!) tools that give you a graphical interface for interacting with your SQLite databases. In this section you will use **Liya** to examine the data store file. Download it from http://www.cutedgesystems.com/software/liya/ or the Mac App Store.

▸ Start Liya. It asks you for a database connection. Under **Database Type** choose **SQLite**. The **Database URL** is the folder that contains the DataStore.sqlite file on your computer. Simply drag the **iPhone Simulator/.../Documents** folder from Finder into this text field. **Database Name** should be **DataStore.sqlite**. (Note: the Database URL field should not include the DataStore.sqlite part of the path.)



Connecting to the SQLite database

› Click **Login** to proceed. The screen should look something like this:



The empty DataStore.sqlite database in Liya

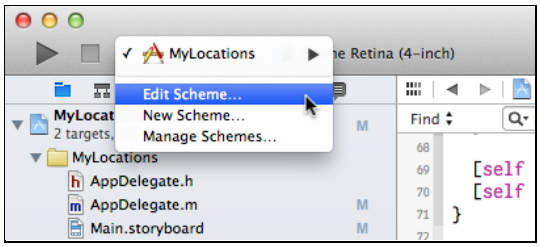
The ZLOCATION table is where your Location objects will be stored. It's currently empty but on the right you can already see the column names that correspond to your fields: ZDATE, ZLATITUDE, and so on. Core Data adds its own columns and tables (with the Z_ prefix). You're not really supposed to change anything in this database by hand, but sometimes using a visual tool like this is handy to see what's going on. You'll come back to Liya once you've inserted new Location objects.

> **Note:** An alternative to Liya is SQLiteStudio, http://sqlitestudio.pl. You can find more tools, paid and free, on the Mac App Store by searching for "sqlite".

There is another handy tool for troubleshooting Core Data. By setting a special flag on the app, you can see the SQL statements that Core Data uses under the hood to talk to the data store. Even if you have no idea of how to speak SQL, this is still valuable information. At least you can use it to tell whether Core Data is doing something or not. To enable this tool, you have to edit the project's **scheme**.

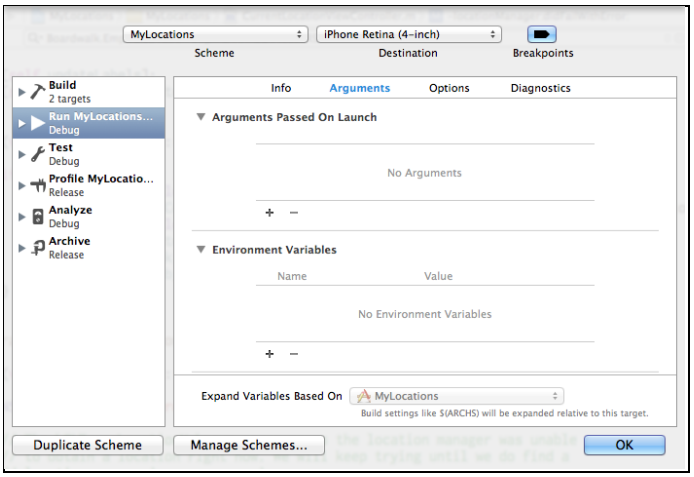
Schemes are how Xcode lets you configure your projects. A scheme is a bunch of settings for building and running your app. Standard projects have just one scheme but you can add additional schemes, which is handy when your project becomes bigger.

› Click on the left part of **MyLocations > iPhone Retina (4-inch)** bar at the top of the screen and choose Edit Scheme... from the menu.



The Edit Scheme... option

The following screen should show up:



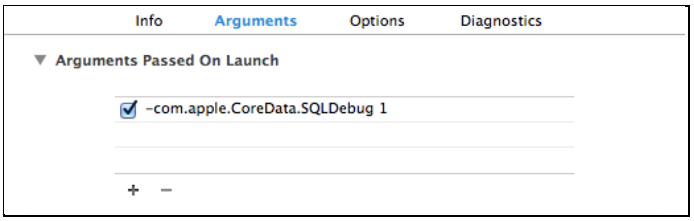
The scheme editor

› Click the **Run MyLocations** option on the left-hand side.

› Make sure the **Arguments** tab is selected.

› In the **Arguments Passed On Launch** section, add the following:

```
-com.apple.CoreData.SQLDebug 1
```



Adding the SQLDebug argument

➤ Press **OK** to close this dialog, and run the app.

When it starts, you should see something like this in the debug area:

```
MyLocations[30818:a0b] CoreData: annotation: Connecting to sqlite
database file at "/Users/matthijs/Library/Application Support/iPhone
Simulator/7.0/Applications/EA10F8C7-7E96-40AB-BBE5-
39C0FC6A678B/Documents/DataStore.sqlite"

MyLocations[30818:a0b] CoreData: sql: SELECT TBL_NAME FROM SQLITE_MASTER
WHERE TBL_NAME = 'Z_METADATA'

MyLocations[30818:a0b] CoreData: sql: pragma journal_mode=wal

MyLocations[30818:a0b] CoreData: sql: pragma cache_size=200

MyLocations[30818:a0b] CoreData: sql: SELECT Z_VERSION, Z_UUID, Z_PLIST
FROM Z_METADATA
```

This is the debug output from Core Data. The specifics don't matter, but it's clear that Core Data is connecting to the data store at this point. Excellent!

Saving the locations

You've successfully initialized Core Data and passed the `NSManagedObjectContext` to the Tag Location screen. Now it's time to make that screen put a new `Location` object into the data store when the Done button is pressed.

➤ Add the following import to **LocationDetailsViewController.m**:

```
#import "Location.h"
```

➤ Add a new instance variable named `_date`:

```
@implementation LocationDetailsViewController
{
  NSString *_descriptionText;
  NSString *_categoryName;
  NSDate *_date;
```

```
}
```

You're adding this variable because you need to store the current date in the new `Location` object.

➤ Change `initWithCoder:` to initialize this new variable:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    _descriptionText = @"";
    _categoryName = @"No Category";
    _date = [NSDate date];
  }
  return self;
}
```

➤ In `viewDidLoad`, change the line that sets the `dateLabel`'s text to:

```
self.dateLabel.text = [self formatDate:_date];
```

➤ Change the `done:` method to the following:

```
- (IBAction)done:(id)sender
{
  HudView *hudView = [HudView hudInView:
                      self.navigationController.view animated:YES];
  hudView.text = @"Tagged";

  // 1
  Location *location = [NSEntityDescription
     insertNewObjectForEntityForName:@"Location"
              inManagedObjectContext:self.managedObjectContext];

  // 2
  location.locationDescription = _descriptionText;
  location.category = _categoryName;
  location.latitude = @(self.coordinate.latitude);
  location.longitude = @(self.coordinate.longitude);
  location.date = _date;
  location.placemark = self.placemark;

  // 3
  NSError *error;
  if (![self.managedObjectContext save:&error]) {
    NSLog(@"Error: %@", error);
    abort();
```

```
  }

  [self performSelector:@selector(closeScreen) withObject:nil
           afterDelay:0.6];
}
```

This is where you do all the work:

1. First, you create a new `Location` object. This is different from how you created objects before. If `Location` were a regular `NSObject`, you would do `[[Location alloc] init]` to create a new instance. However, this is a Core Data managed object, and they are created in a different manner.

   You have to ask the `NSEntityDescription` class to insert a new object for your entity into the managed object context. It's a bit of a weird way to make new objects but that's how you do it in Core Data. The string `@"Location"` is the name of the entity that you added in the data model earlier.

2. Once you have created the `Location` object, you can use it like any other object. Here you set its properties. Note that you convert the latitude and longitude into `NSNumber` objects using the `@()` notation. You don't have to do anything special for the `CLPlacemark` object.

3. You now have a new `Location` object whose properties are set to whatever the user entered in the screen, but if you were to look in the data store at this point you'd still see no objects there. That won't happen until you save the context. This takes any objects that were added to the context, or any managed objects that had their contents changed, and permanently saves these changes into the data store. That's why they call the context the "scratchpad"; its changes aren't persisted until you save them.

Pass-by-reference

The `save:` method takes a parameter, `&error`. The `&` ampersand character means that `save:` will put the results of the operation into an `NSError` object and then stores that object into the `error` variable. This is sometimes called an **output parameter** or **pass-by-reference**.

Most of the time, parameters are used to send data into a method but an output parameter works the other way around. Methods can only return one value and usually that is enough, but sometimes you want to return more than one value to the caller. In that case, you can use an output parameter.

The save method returns a `BOOL` to indicate whether the operation was successful or not. If not, then it also fills up the `NSError` object with additional error information. This is a pattern that you see a lot in the iOS SDK.

The important thing is to not forget the `&`, or Xcode will give cryptic error messages such as: "Incompatible pointer types sending 'NSError *__strong' to

parameter of type 'NSError *__autoreleasing *'". What Xcode really means to say is: You forgot the `&`, dummy!

The `&` operator actually gives you the address of a variable. So `&error` is a pointer to the `error` variable, which makes it a pointer to `NSError *`. In other words, a pointer to a pointer, or `NSError **`. Yikes!

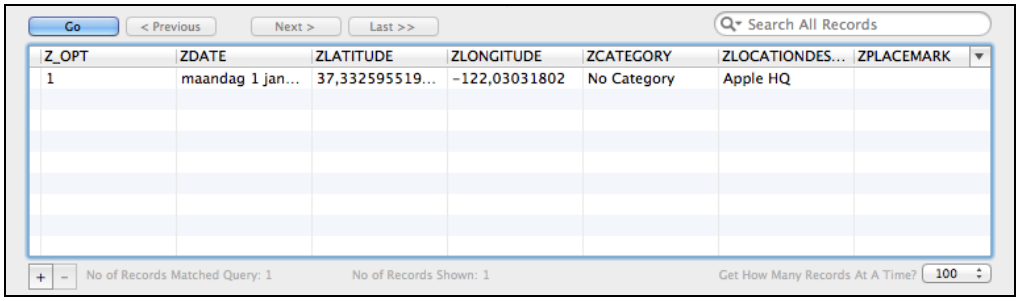
▸ Run the app and tag a location. Enter a description and press the Done button.

If everything went well, Core Data will dump a whole bunch of debug information into the debug area:

```
MyLocations[31071:a0b] CoreData: sql: BEGIN EXCLUSIVE

. . .

MyLocations[31071:a0b] CoreData: sql: INSERT INTO ZLOCATION(Z_PK, Z_ENT,
Z_OPT, ZCATEGORY, ZDATE, ZLATITUDE, ZLOCATIONDESCRIPTION, ZLONGITUDE,
ZPLACEMARK) VALUES(?, ?, ?, ?, ?, ?, ?, ?, ?)

MyLocations[31071:a0b] CoreData: sql: COMMIT

. . .

MyLocations[31071:a0b] CoreData: annotation: sql execution time: 0.0001s
```

These are the SQL statements that Core Data performs to store the new `Location` object in the database.

▸ If you have Liya open, then refresh the contents of the ZLOCATIONS table (press the Go button). There should now be one row in that table:



A new row was added to the table

As you can see, the columns in this table contain the property values from the `Location` object. The only column that is not readable is ZPLACEMARK. Its contents have been encoded as a binary "blob" of data. That is because it's a Transformable attribute and the `NSCoding` protocol has converted its fields into a binary chunk of data.

If you don't have Liya or are a command line junkie, then there is another way to examine the contents of the database. You can use the Terminal app and the `sqlite3` tool, but you'd better know your SQL's from your ABC's:

Examining the database from the Terminal

Handling Core Data errors

To save the contents of the context to the data store, you did:

```
NSError *error;
if (![self.managedObjectContext save:&error]) {
  NSLog(@"Error: %@", error);
  abort();
}
```

What if something goes wrong with the save? The save method returns `NO` and you call the `abort()` function. True to its name, `abort()` will immediately kill the app and return the user to the iPhone's Springboard. That's a nasty surprise for the user, and therefore not recommended.

The good news is that Core Data only gives an error if you're trying to save something that is not valid. In other words, when there is some bug in your app. Of course, you'll get all the bugs out during development so users will never experience any, right? The sad truth is that you'll never catch all your bugs. Some always manage to slip through.

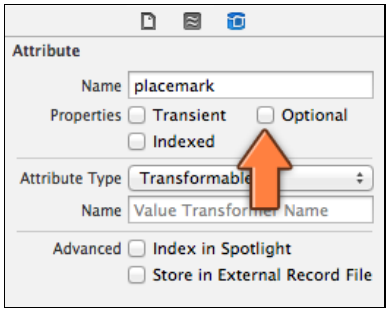
The bad news is that there isn't much else to do but crash when Core Data does give an error. Something went horribly wrong somewhere and now you're stuck with invalid data. If the app were allowed to continue, things would likely only get worse, as there is no telling what state the app is in. The last thing you want to do is to corrupt the user's data.

Instead of making the app crash hard with `abort()`, it will be nice to tell the user about it first so at least they know what is happening. In this section, you'll add an alert view for handling such situations. Again, these errors should happen only

during development, but just in case they occur to an actual user, you'll try to handle it with at least a little bit of grace.

Here's a way to fake such a fatal error, just to illustrate what happens.

➤ Open the data model (**DataModel.xcdatamodeld** in the file list), and select the **placemark** attribute. In the inspector, uncheck the **Optional** flag.



Making the placemark attribute non-optional

That means `location.placemark` may never be `nil`. This is a constraint that Core Data will enforce. When you try to save a `Location` object to the data store whose `placemark` property is `nil`, then Core Data will throw a tantrum. So that's exactly what you're going to do here.

➤ Run the app. Whoops, it crashes right away.

The app crashed in the `persistentStoreCoordinator` method from `AppDelegate`. The debug area says why:

```
reason = "The model used to open the store is incompatible with the one
used to create the store";
```

You have just changed the data model by making changes to the `placemark` attribute. But these changes were only made to the data model inside the application bundle, not to the data store that is in the Documents folder. The DataStore.sqlite file is now out of date with respect to the changed data model.

There are two ways to fix this: 1) Simply throw away the DataStore.sqlite file from the Documents directory; 2) Remove the entire app from the Simulator.

➤ Remove the DataStore.sqlite file and run the app again.

This wasn't actually the crash I wanted to show you, but it's important to know that changing the data model requires you to throw away the database file or Core Data cannot be initialized properly.

Note: There is a migration mechanism in Core Data that is useful for when you release an update to your app with a new data model. Instead of crashing, this mechanism allows you to convert the contents of the user's

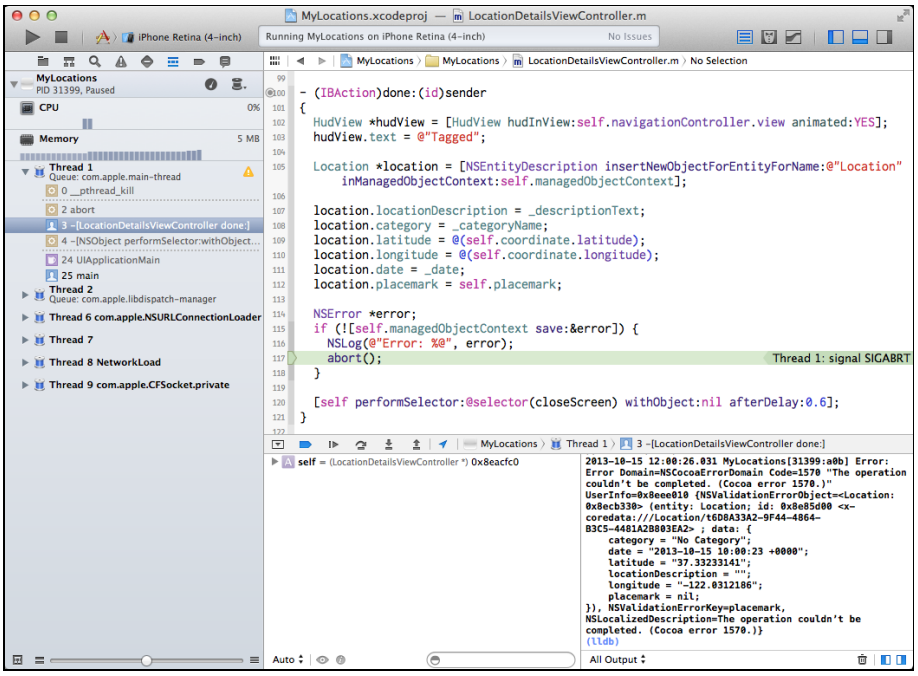
existing data store to the new format. However, when developing it's just as easy to toss out the old database.

› Now here's the trick. Tap the Get My Location button and then tap immediately on Tag Location. If you do that quickly enough, you can beat the reverse geocoder to it and the Tag Location screen will say: "No Address Found". It only says that when `placemark` is `nil`.

(If geocoding happens too fast for you, then you can also temporarily comment out the line `_placemark = [placemarks lastObject];` in `didUpdateLocations` inside **CurrentLocationsViewController.m**. This will make it seem no address was found.)

› Tap the Done button to save the new `Location` object.

The app will crash on the call to `abort()`:



The app crashes after a Core Data error

In the debug area, you can see that it says:

```
NSValidationErrorKey=placemark
```

This means the `placemark` attribute did not validate properly. Because you set it to non-optional, Core Data does not accept a placemark value that is `nil`.

This is what happens when you run the app from within Xcode. When it crashes, the debugger takes over and points at the line with the error. But that's not what the user sees.

➤ Stop the app. Now tap the app's icon in the Simulator to launch the app outside of Xcode. Repeat the same procedure to make the app crash. The app will simply cease functioning and disappear from the screen.

Imagine this happening to a user who just paid 99 cents (or more) for your app. They'll be horribly confused, "What just happened?!" They may even ask for their money back.

It's better to show an alert view when this happens. After the user dismisses that alert, you'll still call `abort()` to make the app crash, but at least the user knows the reason why. (The alert message should probably ask them to contact you and explain what they did, so you can fix that bug in the next version of your app.)

➤ Replace the error handling code in the `done` action with:

```
  NSError *error;
  if (![self.managedObjectContext save:&error]) {
    FATAL_CORE_DATA_ERROR(error);
    return;
  }
```

Something called `FATAL_CORE_DATA_ERROR()` has taken the place of `NSLog()` and `abort()`. It may look like a function but it's actually a so-called **macro**.

➤ Add the `FATAL_CORE_DATA_ERROR` macro to **MyLocations-Prefix.pch**:

```
extern NSString * const
                    ManagedObjectContextSaveDidFailNotification;

#define FATAL_CORE_DATA_ERROR(__error__)\
  NSLog(@"*** Fatal error in %s:%d\n%@\n%@",\
      __FILE__, __LINE__, error, [error userInfo]);\
  [[NSNotificationCenter defaultCenter] postNotificationName:\
      ManagedObjectContextSaveDidFailNotification object:error];
```

(Important: When you type in this code, you should not change the wrapping of the lines following the `#define`. Each line except the last needs to end with a backslash.)

Remember that anything you define in the Prefix file is visible in all your source files, and so is this macro.

Macros and the preprocessor

Before the compiler gets to have at your source files, Xcode first feeds them to the **preprocessor**. This tool is responsible for the `#import` statements and any other statement starting with a # hash sign.

When the preprocessor encounters an `#import "MyClass.h"` statement, it reads the MyClass.h file and literally inserts it into the source code. The combined text of the source code and the contents of all the .h files it imports is then sent to the compiler.

The preprocessor also handles `#define` macros. The contents of that macro are also literally placed in your source code wherever the macro is used. Macros aren't very common in Objective-C code but you still see them from time to time.

For example, when you do,

```
#define SOME_NUMBER 123

- (void)myMethod
{
  int temp = SOME_NUMBER;
}
```

the preprocessor turns this into:

```
- (void)myMethod
{
  int temp = 123;
}
```

For this reason, `#define`s are often used to make symbolic names for numeric constants, although you've also seen that you can use the `const` keyword for this. It is customary for `#define` symbols to be written in `ALL_CAPS_AND_UNDERBARS`.

Macros can be quite complex and can even have parameters, just like functions. The difference with functions is that macros are "expanded" when the source code is compiled. After the compiler is done and your app is built, the macros no longer exist. Your code will never "call" a macro during runtime. They are just a convenience to save you some typing.

When you put `FATAL_CORE_DATA_ERROR()` anywhere in your code, the preprocessor inserts the statements from that macro into the code for you when it is compiled.

So the context saving code becomes:

```
NSError *error;
if (![self.managedObjectContext save:&error]) {

  NSLog(@"*** Fatal error in %s:%d\n%@\n%@",
                  __FILE__, __LINE__, error, [error userInfo]);
```

```
    [[NSNotificationCenter defaultCenter] postNotificationName:
        ManagedObjectContextSaveDidFailNotification object:error];
}
```

That looks like no fun to type it everywhere you want to save the context – which is exactly why you replaced it with a macro. You'll be using that macro in several other places as well, so if you decide to change how to handle Core Data errors then you only have to change the macro.

So what does the code from the macro do, actually?

It first outputs the error message to the Debug Area using `NSLog()`:

```
    NSLog(@"*** Fatal error in %s:%d\n%@\n%@",
                    __FILE__, __LINE__, error, [error userInfo]);
```

This is not very different from what you did before, except for the addition of `__FILE__` and `__LINE__`. These are two special symbols provided by the preprocessor that refer to the name of the source file and the line number at that point. Using these, the `NSLog()` not only gives you the error but also roughly the place in the source code where that error happened.

After dumping the debug info, the macro does the following:

```
    [[NSNotificationCenter defaultCenter] postNotificationName:
        ManagedObjectContextSaveDidFailNotification object:error];
```

I've been using the term "notification" to mean any generic event or message being delivered but iOS also has an object called the `NSNotificationCenter` (not to be confused with Notification Center on your phone).

The code above uses `NSNotificationCenter` to post a **notification**. Any object in your app can subscribe to such notifications and when these occur, `NSNotificationCenter` will call a certain method on those listener objects. Using this official notification system is yet another way that your objects can communicate with each other. The handy thing is that the object that sends the notification and the object that receives the notification don't need to know anything about each other.

UIKit defines a lot of standard notifications that you can subscribe to. For example, there is a notification that lets you know that the app is about to be suspended. But you can also define your own notifications, and that is what you've done here. Your custom notification is called `ManagedObjectContextSaveDidFailNotification`.

The idea is that there is one place in the app that listens for this notification, pops up an alert view, and aborts. The great thing about using `NSNotificationCenter` is that your Core Data code does not need to care about any of this. Whenever a saving error occurs, no matter at which point in the app, the

`FATAL_CORE_DATA_ERROR()` macro sends out this notification, safe in the belief that some other object is listening for the notification and will handle the error.

So who will handle the error? The app delegate is a good place for that. It's the top-level object in the app and you're always guaranteed this object exists.

➤ Add the following to `didFinishLaunchingWithOptions` in **AppDelegate.m**, just before the `return` statement:

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

This tells `NSNotificationCenter` that the `AppDelegate` wants to be notified whenever a `ManagedObjectContextSaveDidFailNotification` is posted. The selector contains the name of the method that should be invoked, `fatalCoreDataError:`.

➤ Add this method to **AppDelegate.m:**

```
- (void)fatalCoreDataError:(NSNotification *)notification
{
  UIAlertView *alertView = [[UIAlertView alloc]
    initWithTitle:NSLocalizedString(@"Internal Error", nil)
    message:NSLocalizedString(@"There was a fatal error in the app and
it cannot continue.\n\nPress OK to terminate the app. Sorry for the
inconvenience.", nil)
    delegate:self
    cancelButtonTitle:NSLocalizedString(@"OK", nil)
    otherButtonTitles:nil];

  [alertView show];
}
```

This simply shows an alert view. It sets the alert view's delegate to `self`, so you also have to make `AppDelegate` conform to the `UIAlertViewDelegate` protocol.

➤ Add the `UIAlertViewDelegate` protocol to the class extension:

```
@interface AppDelegate () <UIAlertViewDelegate>
```

➤ And implement the delegate method:

```
#pragma mark - UIAlertViewDelegate

- (void)alertView:(UIAlertView *)alertView
        didDismissWithButtonIndex:(NSInteger)buttonIndex
{
```

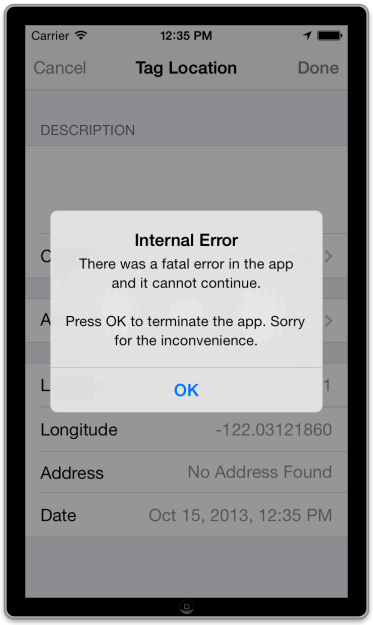
```
    abort();
}
```

There is one more thing you need to do. Because you're creating your own notification, you still need to define its name somewhere. This is just an `NSString`.

▸ Add the following line at the top of **AppDelegate.m**, just below the imports:

```
NSString * const ManagedObjectContextSaveDidFailNotification =
                 @"ManagedObjectContextSaveDidFailNotification";
```

▸ Run the app again and try to tag a location before the street address has been obtained. Now before the app crashes, at least it tells the user what's going on:



The app crashes with a message

Again, I should stress that you test your app very well to make sure you're not giving Core Data any objects that do not validate. You want to avoid these save errors at all costs! Ideally, users should never have to see that alert view, but it's good to have it in place because there are no guarantees your app won't have bugs.

Note: You can legitimately use `[managedObjectContext save:]` to let Core Data validate user input. There is no requirement that you make your app crash after an unsuccessful `save`, only if the error was unexpected and definitely shouldn't have happened!

> Besides the "optional" flag, there are many more validation settings you can put on the attributes of your entities. If you let the user of your app enter data that needs to go into these attributes, then it's perfectly acceptable to use `save:` to perform the validation. If it returns `NO`, then there was an error in whatever value the user entered and you should handle it.

➤ In the data model, set the **placemark** attribute back to optional. Remember to throw away the DataStore.sqlite file again! Run the app just to make sure everything works as it should. (If you commented out the line in `didUpdateLocations` to fake the error, then put that line back in.)

You can find the project files for the app up to this point under **04 - Core Data** in the tutorial's Source Code folder.

The Locations tab

You've set up the data model and gave the app the ability to save new locations to the data store. Now you'll show those locations in a table view in the second tab.

➤ Open the storyboard editor and delete the **Second View Controller**. This is a leftover from the project template and you don't need it.

➤ Drag a new **Navigation Controller** into the canvas. (This has a table view controller attached to it, which is fine. You'll use that in a second.)

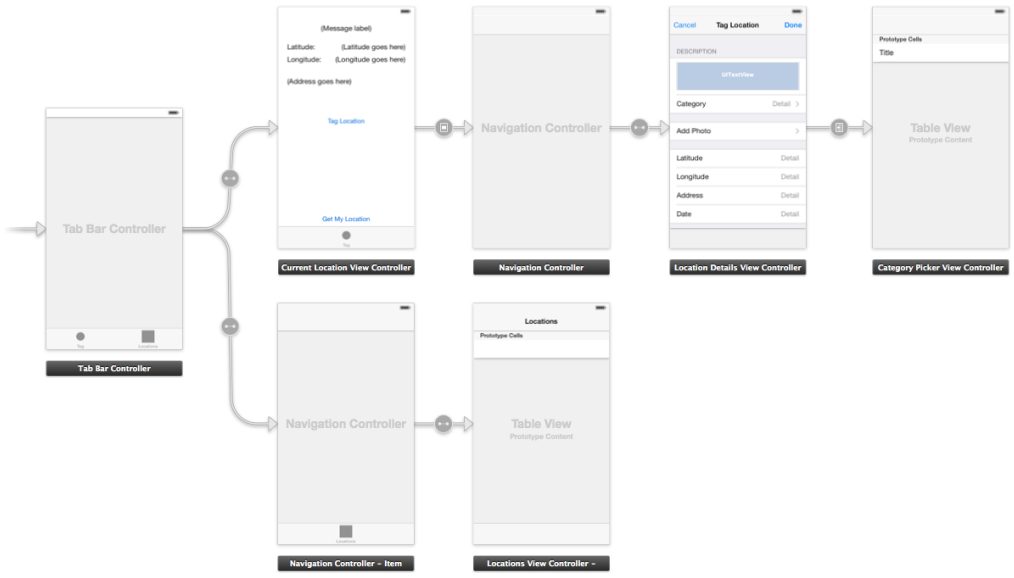
➤ **Ctrl-drag** from the Tab Bar Controller to this new Navigation Controller and select **Relationship Segue - view controllers**. This adds the navigation controller to the tab bar.

➤ The Navigation Controller now has a **Tab Bar Item** that is named "Item". Rename it to **Locations**.

➤ Double-click the navigation bar of the Root View Controller (the one attached to the new Navigation Controller) and change the title to **Locations**.

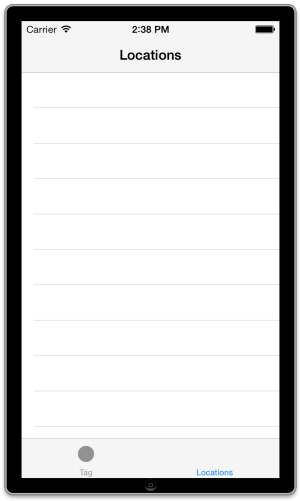
➤ In the **Identity inspector**, change the **Class** of the table view controller to **LocationsViewController**. This class doesn't exist yet but you'll make it in a minute.

The storyboard now looks like this:



The storyboard after adding the Locations screen

If you run the app it doesn't look very interesting yet:



The Locations screen in the second tab

Before you can show any data in the table, you first have to design the prototype cell.

› Set the prototype cell's Reuse Identifier to **Location**.

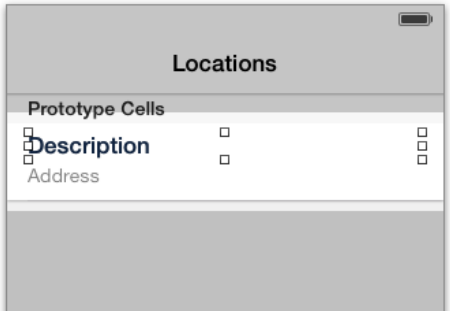
› In the **Size inspector**, change **Row Height** to 57.

› Drag two **Labels** into the cell. Give the top one the text **Description** and the bottom one the text **Address**. This is just so you know what they are for.

› Set the font of the Description label to **System Bold**, **17**. Give this label a tag of 100.

➤ Set the font of the Address label to **System**, **14**. Set the Text color to black with 50% opacity (so its looks like a medium gray). Give it a tag of 101.

The cell will look something like this:

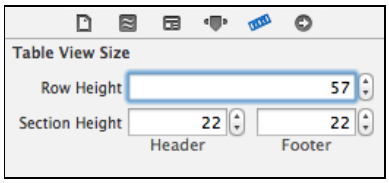


The prototype cell

Make sure that the labels are wide enough to span the entire cell.

Just changing the Row Height of the prototype cell isn't enough; you also have to tell the table view about the height of its rows.

➤ Select the table view and go to the **Size inspector**. Set the **Row Height** field to 57:



Setting the row height on the table view

Let's write the code for the view controller. You've seen table view controllers several times now, so this should be easy. You're going to fake the content first, because it's good to make sure that the prototype cell works before you have to deal with Core Data.

➤ Add a new table view controller subclass file to the project and name it **LocationsViewController**.

➤ Change the contents of **LocationsViewController.h** to:

```
@interface LocationsViewController : UITableViewController

@property (nonatomic, strong) NSManagedObjectContext
                                        *managedObjectContext;

@end
```

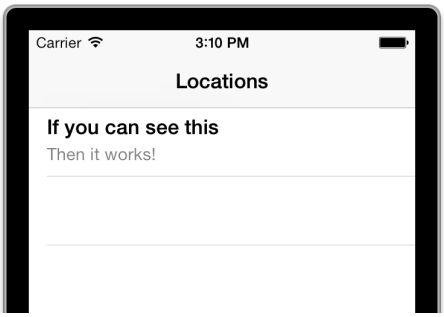
You're already giving this class an `NSManagedObjectContext` property even though it won't be using it yet.

➤ Change the contents of **LocationsViewController.m** to:

```
#import "LocationsViewController.h"

@implementation LocationsViewController

#pragma mark - UITableViewDataSource

- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
  return 1;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
                 dequeueReusableCellWithIdentifier:@"Location"];

  UILabel *descriptionLabel = (UILabel *)[cell viewWithTag:100];
  descriptionLabel.text = @"If you can see this";

  UILabel *addressLabel = (UILabel *)[cell viewWithTag:101];
  addressLabel.text = @"Then it works!";

  return cell;
}

@end
```

You're faking a single row with some placeholder text in the labels.

➤ Run the app to make sure the table view works.



The table view with fake data

Excellent. Now it's time to fill up the table with the `Location` objects from the data store.

➤ Run the app and tag a handful of locations. If there is no data in the data store, then the app doesn't have much to show...

This part of the app doesn't know anything yet about the `Location` objects that you have added to the data store. In order to display them in the table view, you need to obtain references to these objects somehow. You can do that by asking the data store. This is called **fetching**.

➤ First, add a new instance variable to **LocationsViewController.m**:

```
@implementation LocationsViewController
{
  NSArray *_locations;
}
```

This array contains the list of `Location` objects.

➤ Add the `viewDidLoad` method in **LocationsViewController.m**:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  // 1
  NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];

  // 2
  NSEntityDescription *entity = [NSEntityDescription
              entityForName:@"Location"
              inManagedObjectContext:self.managedObjectContext];
  [fetchRequest setEntity:entity];

  // 3
  NSSortDescriptor *sortDescriptor = [NSSortDescriptor
```

```
                    sortDescriptorWithKey:@"date" ascending:YES];
  [fetchRequest setSortDescriptors:@[sortDescriptor]];

  // 4
  NSError *error;
  NSArray *foundObjects = [self.managedObjectContext
                executeFetchRequest:fetchRequest error:&error];
  if (foundObjects == nil) {
    FATAL_CORE_DATA_ERROR(error);
    return;
  }

  // 5
  _locations = foundObjects;
}
```

This may look daunting but it's actually quite simple. You're going to ask the managed object context for a list of all `Location` objects in the data store, sorted by date.

1. The `NSFetchRequest` is the object that describes which objects you're going to fetch from the data store. To retrieve an object that you previously saved to the data store, you create a fetch request that describes the search parameters of the object – or multiple objects – that you're looking for.
2. The `NSEntityDescription` tells the fetch request you're looking for `Location` entities.
3. The `NSSortDescriptor` tells the fetch request to sort on the `date` attribute, in ascending order. In order words, the `Location` objects that the user added first will be at the top of the list. You can sort on any attribute here (later in this tutorial you'll sort on the `Location`'s category as well).

That completes the fetch request. It took a few lines of code, but basically you said: "Get all `Location` objects from the data store and sort them by date."

4. Now that you have the fetch request, you can tell the context to execute it. The `executeFetchRequest` method returns an `NSArray` with the sorted objects, or `nil` in case of an error. Since those errors shouldn't really happen, you use the special macro to handle that situation.
5. If everything went well, you assign the contents of the `foundObjects` array to the `_locations` instance variable.

Now that you've loaded the list of `Location` objects into an instance variable, you can change the table view's data source methods.

▸ First, add an import to the top of the source file:

```
#import "Location.h"
```

➤ Change the data source methods to:

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
  return [_locations count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
                 dequeueReusableCellWithIdentifier:@"Location"];

  Location *location = _locations[indexPath.row];

  UILabel *descriptionLabel = (UILabel *)[cell viewWithTag:100];
  descriptionLabel.text = location.locationDescription;

  UILabel *addressLabel = (UILabel *)[cell viewWithTag:101];
  addressLabel.text = [NSString stringWithFormat:@"%@ %@, %@",
    location.placemark.subThoroughfare,
    location.placemark.thoroughfare,
    location.placemark.locality];

  return cell;
}
```

➤ Run the app. Now switch to the Locations tab and... crap! It crashes.

The text in the debug area says something like:

```
*** Terminating app due to uncaught exception 'NSInternalInconsistency
Exception', reason: '+entityForName: could not locate an
NSManagedObjectModel for entity name 'Location''
```

That's a little strange, as you certainly have added an entity named Location to the data model.

Exercise. What did you forget? □

Answer: You added a `managedObjectContext` property to `LocationsViewController`, but never gave this property a value. Therefore, there is nothing to fetch Location objects from.

➤ In **AppDelegate.m**, first add a new import:

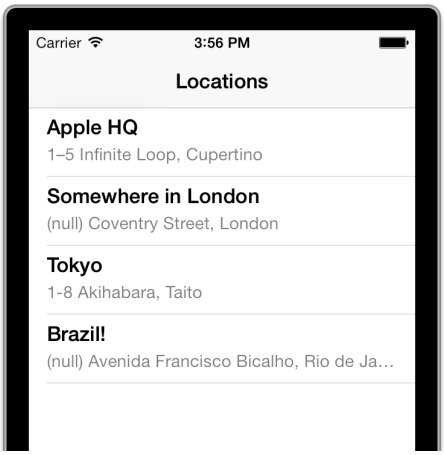
```
#import "LocationsViewController.h"
```

➤ Then, in `didFinishLaunchingWithOptions`, add the following:

```
UINavigationController *navigationController =
  (UINavigationController *)tabBarController.viewControllers[1];

LocationsViewController *locationsViewController =
        (LocationsViewController *)
                        navigationController.viewControllers[0];

locationsViewController.managedObjectContext =
                                        self.managedObjectContext;
```

This looks up the `LocationsViewController` in the storyboard and gives it a reference to the managed object context.

➤ Run the app again and switch to the Locations tab. Core Data properly fetches the objects and shows them on the screen:



The list of Locations

Note that the list doesn't update yet if you tag a new location. You have to restart the app to see the new `Location` object appear. You'll solve this later in the tutorial.

Creating a custom Table View Cell subclass

Using `[cell viewWithTag:xxx]` to find the labels from the table view cell works, but it doesn't look very object-oriented to me. It would be much nicer if you could make your own `UITableViewCell` subclass and give it outlets for the labels. Fortunately, you can and it's pretty easy!

➤ Add a new file to the project, choose **Objective-C class**. Name it **LocationCell** and make it a subclass of `UITableViewCell`.

➤ Replace the contents of **LocationCell.h** with the following:

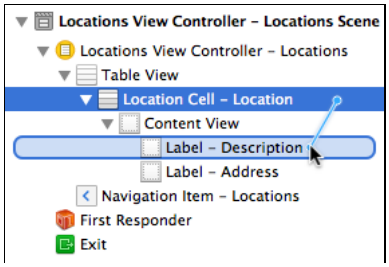
```
@interface LocationCell : UITableViewCell
```

```
@property (nonatomic, weak) IBOutlet UILabel *descriptionLabel;
@property (nonatomic, weak) IBOutlet UILabel *addressLabel;

@end
```

You can leave the rest of **LocationCell.m** as is.

▸ Open the storyboard and select the prototype cell that you made earlier. In the **Identity inspector**, set **Class** to **LocationCell**.

▸ Now you can connect the two labels to the two outlets. This time the outlets are not on the view controller but on the cell, so **Ctrl-drag** from the cell to the labels:



Connect the outlets to the cell

That is all you need to do to make the table view use your own table view cell class. You do need to update `LocationsViewController` to make use of it.

▸ In **LocationsViewController.m**, add an import for `LocationCell`:

```
#import "LocationCell.h"
```

▸ Replace `cellForRowAtIndexPath` with the following:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
                dequeueReusableCellWithIdentifier:@"Location"];
  [self configureCell:cell atIndexPath:indexPath];
  return cell;
}
```

As before, this asks for a cell using `dequeueReusableCellWithIdentifier`, but now this will always be a `LocationCell` object instead of a regular `UITableViewCell`.

Note that `@"Location"` is the re-use identifier from the placeholder cell, but `LocationCell` is the class of the actual cell object that you're getting. And `LocationCell` is still a `UITableViewCell` but with extra properties. I hope that's not too confusing.

➤ Add the new `configureCell:atIndexPath:` method:

```
- (void)configureCell:(UITableViewCell *)cell
          atIndexPath:(NSIndexPath *)indexPath
{
  LocationCell *locationCell = (LocationCell *)cell;
  Location *location = _locations[indexPath.row];

  if ([location.locationDescription length] > 0) {
    locationCell.descriptionLabel.text =
                                    location.locationDescription;
  } else {
    locationCell.descriptionLabel.text = @"(No Description)";
  }

  if (location.placemark != nil) {
    locationCell.addressLabel.text =
        [NSString stringWithFormat:@"%@ %@, %@",
            location.placemark.subThoroughfare,
            location.placemark.thoroughfare,
            location.placemark.locality];
  } else {
    locationCell.addressLabel.text = [NSString stringWithFormat:
      @"Lat: %.8f, Long: %.8f",
      [location.latitude doubleValue],
      [location.longitude doubleValue]];
  }
}
```

Instead of using `viewWithTag` to find the description and address labels, you now simply use the `descriptionLabel` and `addressLabel` properties of the cell. You first have to cast the `cell` variable to a `LocationCell` because the `UITableViewCell` superclass doesn't know anything about these properties.

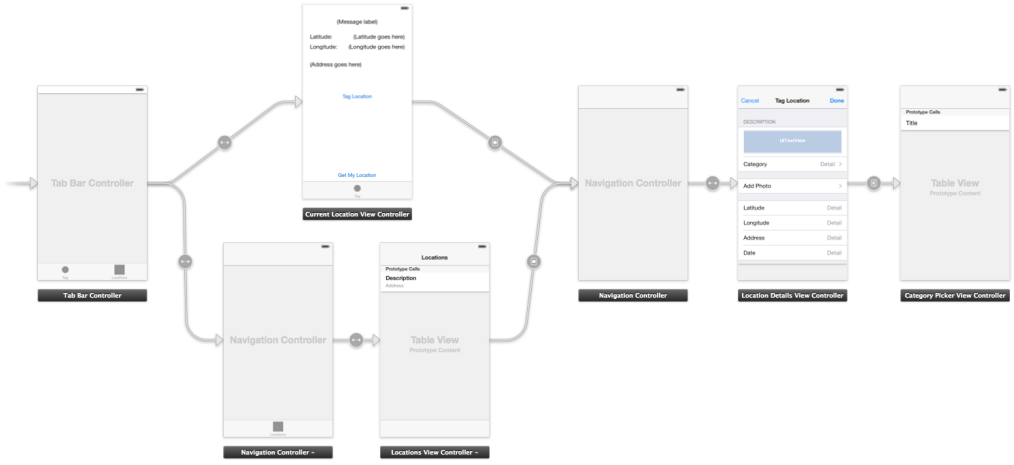
Using a custom subclass for your table view cells, there is no limit to how complex you can make them.

Editing the locations

You will now connect the `LocationsViewController` to the Location Details screen, so that when you tap a row in the table, it opens a screen that lets you edit that location. You'll be re-using the `LocationDetailsViewController` for that, but have it edit an existing `Location` object rather than add a new one.

➤ Go to the storyboard. Select the prototype cell and **ctrl-drag** to the Navigation Controller that is connected to the Location Details screen. Add a **modal** segue and name it **EditLocation**.

After a bit of tidying up, the storyboard looks like this:



The Location Details screen is now also connected to the Locations screen

There are now two segues from two different screens going to the same view controller. This is the reason why you should build your view controllers to be as independent of their "calling" controllers as possible, so that you can easily re-use them somewhere else in your app. Soon you will be calling this same screen from yet another place. In total there will be three segues to it.

▸ First, add an import to the top of **LocationsViewController.m**:

```
#import "LocationDetailsViewController.h"
```

▸ Also add the `prepareForSegue` method:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"EditLocation"]) {
    UINavigationController *navigationController =
                                segue.destinationViewController;

    LocationDetailsViewController *controller =
              (LocationDetailsViewController *)
                          navigationController.topViewController;

    controller.managedObjectContext = self.managedObjectContext;

    NSIndexPath *indexPath = [self.tableView
                                        indexPathForCell:sender];
    Location *location = _locations[indexPath.row];
    controller.locationToEdit = location;
  }
```

```
}
```

This method is invoked when the user taps a row in the Locations screen. It figures out which `Location` object belongs to this row and then puts it in the new `locationToEdit` property of `LocationDetailsViewController`. This property doesn't exist yet but you'll add it in a moment.

When editing an existing `Location` object, you have to do a few things differently in the `LocationDetailsViewController`. The title of the screen shouldn't be "Tag Location" but "Edit Location". You also must put the values from the existing `Location` object into the various cells.

The value of the new `locationToEdit` property determines whether the screen operates in "adding" mode or in "editing" mode.

➤ Add this property to **LocationDetailsViewController.h**:

```
@class Location;

@interface LocationDetailsViewController : UITableViewController

. . .

@property (nonatomic, strong) Location *locationToEdit;

@end
```

The `@class` statement is necessary otherwise the compiler won't know what the `Location` symbol means. Now it knows it's a class.

➤ In **LocationDetailsViewController.m**, expand `viewDidLoad` to check whether `locationToEdit` is set:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  if (self.locationToEdit != nil) {
    self.title = @"Edit Location";
  }

  . . .
}
```

If `locationToEdit` is not `nil`, you're editing an existing `Location` object. In that case, the title of the screen becomes "Edit Location".

➤ The real magic happens in the `setLocationToEdit:` method. Add this as well:

```
- (void)setLocationToEdit:(Location *)newLocationToEdit
{
  if (_locationToEdit != newLocationToEdit) {
    _locationToEdit = newLocationToEdit;

    _descriptionText = _locationToEdit.locationDescription;
    _categoryName = _locationToEdit.category;
    _date = _locationToEdit.date;

    self.coordinate = CLLocationCoordinate2DMake(
                        [_locationToEdit.latitude doubleValue],
                        [_locationToEdit.longitude doubleValue]);

    self.placemark = _locationToEdit.placemark;
  }
}
```

The `setLocationToEdit:` method is a so-called **setter**. Recall that a property always has two methods – a getter and a setter – in addition to its instance variable. The getter returns the value of the instance variable and the setter assigns it a new value.

The `locationToEdit` property has a getter method named simply `locationToEdit` and a setter method named `setLocationToEdit:`. When you do,

```
controller.locationToEdit = someLocation;
```

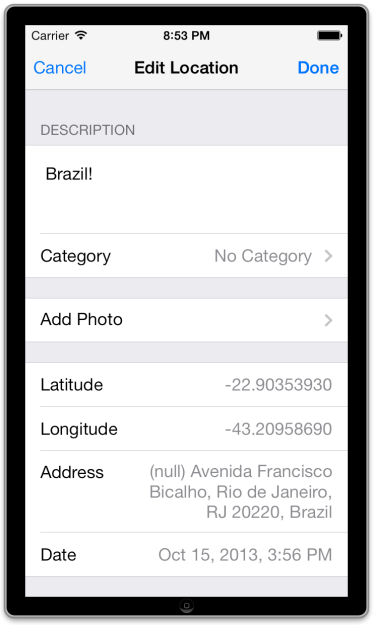
then behind the scenes the `setLocationToEdit:` method is called to make the actual assignment.

The compiler automatically generates the getter and setter methods for your properties (unless you used the `@dynamic` directive), but you can override them to do your own stuff when the property is being accessed. Here, you take the opportunity to fill in the instance variables and properties with the `Location` object's values.

Because `prepareForSegue` – and therefore `setLocationToEdit:` – is called before `viewDidLoad`, this puts the right values on the screen when it becomes visible.

> **Note:** You use the function `CLLocationCoordinate2DMake()` to turn the separate latitude and longitude values from the `Location` object into a single `CLLocationCoordinate` value. Remember this is a struct, not an object, so alloc+init aren't necessary.

➤ Run the app, go to the Locations tab and tap on a row. The Edit Location screen should now appear with the data from the selected location:



Editing an existing location

› Change the description of the location and press Done.

Nothing happened?! Well, that's not quite true. Stop the app and run it again. You will see that a new location has been added with the changed description, but the old one is still there as well.

There are two problems to solve: 1) When editing an existing location you must not insert a new one, and 2) The screen doesn't update to reflect any changes in the data model.

The first fix is easy.

› Still in **LocationDetailsViewController.m**, change the top part of `done:` to:

```
- (IBAction)done:(id)sender
{
  HudView *hudView = [HudView
         hudInView:self.navigationController.view animated:YES];

  Location *location = nil;
  if (self.locationToEdit != nil) {
    hudView.text = @"Updated";
    location = self.locationToEdit;
  } else {
    hudView.text = @"Tagged";
    location = [NSEntityDescription
              insertNewObjectForEntityForName:@"Location"
              inManagedObjectContext:self.managedObjectContext];
```

```
    }

    location.locationDescription = _descriptionText;
    . . .
```

The change is very straightforward: you only ask Core Data for a new `Location` object if you don't already have one. You also make the text in the HUD say "Updated" when the user is editing an existing `Location`.

➤ Run the app again and edit a location. Now the HUD should say updated. Stop the app and run it again to verify that the object was indeed properly changed. (You can also look at it directly in the SQLite database, of course.)

Exercise. Why do you think the table view isn't being updated after you change a `Location` object? Tip: The table view also doesn't update when you tag new locations. □

Answer: You only fetch the `Location` objects in `viewDidLoad`. But `viewDidLoad` is only performed once, in this case when the app starts. After the initial load of the Locations screen, its contents are never refreshed.

In the Checklists app, you solved this by using a delegate, and that would be a valid solution here too. The `LocationDetailsViewController` could tell you through delegate methods that a location has been added or changed. But since you're using Core Data, there is a much cooler way to do this, and that's the topic of the next section.

You can find the project files for the app up to this point under **05 - Locations Tab** in the tutorial's Source Code folder.

Using NSFetchedResultsController

As you are no doubt aware by now, table views are everywhere in iOS apps. A lot of the time when you're working with Core Data, you want to fetch objects from the data store and show them in a table view. And when those objects change, you want to update the table view in response to show the changes to the user.

So far you've filled up the table view by manually fetching the results, but then you also need to manually check for changes and perform the fetch again to update the table. Thanks to `NSFetchedResultsController`, that suddenly becomes a lot easier.

It works like this: you give `NSFetchedResultsController` a fetch request, just like the `NSFetchRequest` you've made earlier, and tell it to go fetch the objects. So far nothing new. But you don't put the results from that fetch into your own array. Instead, you read them straight from the fetched results controller. In addition, you make the view controller the delegate for the `NSFetchedResultsController`. Through

this delegate the view controller is informed that objects have been changed, added or deleted. In response it should update the table.

➤ In **LocationsViewController.m**, replace the `_locations` instance variable with the new `_fetchedResultsController` variable:

```
@implementation LocationsViewController
{
  NSFetchedResultsController *_fetchedResultsController;
}
```

➤ Add the following method above `viewDidLoad`:

```
- (NSFetchedResultsController *)fetchedResultsController
{
  if (_fetchedResultsController == nil) {

    NSFetchRequest *fetchRequest =
                                  [[NSFetchRequest alloc] init];

    NSEntityDescription *entity = [NSEntityDescription
              entityForName:@"Location"
              inManagedObjectContext:self.managedObjectContext];
    [fetchRequest setEntity:entity];

    NSSortDescriptor *sortDescriptor = [NSSortDescriptor
                   sortDescriptorWithKey:@"date" ascending:YES];
    [fetchRequest setSortDescriptors:@[sortDescriptor]];

    [fetchRequest setFetchBatchSize:20];

    _fetchedResultsController =
        [[NSFetchedResultsController alloc]
            initWithFetchRequest:fetchRequest
            managedObjectContext:self.managedObjectContext
            sectionNameKeyPath:nil
            cacheName:@"Locations"];

    _fetchedResultsController.delegate = self;
  }
  return _fetchedResultsController;
}
```

You should recognize the general pattern of this method as lazy loading:

```
- (NSFetchedResultsController *)fetchedResultsController
{
```

```
  if (_fetchedResultsController == nil) {

    // create the fetchedResultsController

  }
  return _fetchedResultsController;
}
```

It's good to get into the habit of lazily loading objects. You don't allocate them until you first use them. This makes your apps quicker to start and it saves memory.

> **Note:** To use the `NSFetchedResultsController` you should never access the `_fetchedResultsController` instance variable directly. You always need to write `self.fetchedResultsController` so that this getter method gets called and the `NSFetchedResultsController` is instantiated when necessary.

Inside the `if`, where you create the `NSFetchedResultsController` instance, you do the same thing that you used to do in `viewDidLoad`: make an `NSFetchRequest` and give it an entity description and a sort descriptor.

This is new:

```
    [fetchRequest setFetchBatchSize:20];
```

If you have a huge table with hundreds of objects then it requires a lot of memory to keep all of these objects around, even though you can only see a handful of them at a time. The `NSFetchedResultsController` is pretty smart about this and will only fetch the objects that you can actually see, which cuts down on memory usage. This is all done in the background without you having to worry about it. The fetch batch size setting allows you to tweak how many objects will be fetched at a time.

Once the fetch request is set up, you can create the star of the show:

```
    _fetchedResultsController =
        [[NSFetchedResultsController alloc]
            initWithFetchRequest:fetchRequest
            managedObjectContext:self.managedObjectContext
            sectionNameKeyPath:nil
            cacheName:@"Locations"];

    _fetchedResultsController.delegate = self;
```

The `cacheName` needs to be a unique name that `NSFetchedResultsController` uses to cache the search results. It keeps this cache around even after your app quits, so the next time it starts up the fetch request is lightning fast, as the

`NSFetchedResultsController` doesn't have to make a round-trip to the database but can simply read from the cache.

We'll talk about the `sectionNameKeyPath` parameter shortly.

➤ Add a class extension at the top of the file to make `LocationsViewController` conform to the `NSFetchedResultsControllerDelegate` protocol:

```
@interface LocationsViewController ()
                              <NSFetchedResultsControllerDelegate>

@end
```

Now that you have a fetched results controller, you can simplify `viewDidLoad`.

➤ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  [self performFetch];
}

- (void)performFetch
{
  NSError *error;
  if (![self.fetchedResultsController performFetch:&error]) {
    FATAL_CORE_DATA_ERROR(error);
    return;
  }
}
```

You still perform the initial fetch in `viewDidLoad`, using the new `performFetch` helper method, but if any `Location` objects change after that, then the `NSFetchedResultsController`'s delegate methods are called. I'll show you in a second how that works.

It's always a good idea to explicitly set the delegate to `nil` when you no longer need the `NSFetchedResultsController`, just so you don't get any more notifications that were still pending.

➤ For that reason, add a `dealloc` method:

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

The `dealloc` method is invoked when this view controller is destroyed. It may not strictly be necessary to `nil` out the delegate here, but it's a bit of defensive programming that won't hurt. (Note that in this app the `LocationsViewController` will never actually be deallocated because it's one of the top-level view controllers in the tab bar.)

Because you removed the `_locations` array, you should also change the table's data source methods.

▸ Change `numberOfRowsInSection` to:

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
  id <NSFetchedResultsSectionInfo> sectionInfo =
              [self.fetchedResultsController sections][section];

  return [sectionInfo numberOfObjects];
}
```

You simply ask the fetched results controller for the number of rows and return it. You'll learn more about this `NSFetchedResultsSectionInfo` object later on.

▸ Change the top of the `configureCell` method to:

```
- (void)configureCell:(UITableViewCell *)cell
          atIndexPath:(NSIndexPath *)indexPath
{
  LocationCell *locationCell = (LocationCell *)cell;
  Location *location = [self.fetchedResultsController
                                    objectAtIndexPath:indexPath];

  . . .
```

Instead of looking into the `_locations` array, you now ask the `fetchedResultsController` for the object at the requested index-path. Because it is designed to work closely together with table views, `NSFetchedResultsController` knows how to deal with index-paths, so that's very convenient.

Note: Even though you have not defined a property for it, you access the fetched results controller through `self.fetchedResultsController`. That is necessary for the lazy loading to work. Using `self` will always call the getter method, which instantiates the `NSFetchedResultsController` object if it doesn't exist yet. If instead you directly used the instance variable `_fetchedResultsController`, the fetcher object would never be created.

> Remember, when you use lazy loading, you always have to access the object through its getter method.

There is one more error in the class that prevents it from compiling.

➤ Change the offending line in `prepareForSegue` to:

```
Location *location = [self.fetchedResultsController
                                    objectAtIndexPath:indexPath];
```

➤ Run the app and switch to the Locations tab. All your locations should still be there.

If you edit a location or add a new one, the table still doesn't update. That's because you haven't implemented the delegate methods for `NSFetchedResultsController` yet.

➤ Add the following code to the bottom of the source file:

```
#pragma mark - NSFetchedResultsControllerDelegate

- (void)controllerWillChangeContent:
            (NSFetchedResultsController *)controller
{
  NSLog(@"*** controllerWillChangeContent");
  [self.tableView beginUpdates];
}

- (void)controller:(NSFetchedResultsController *)controller
    didChangeObject:(id)anObject
    atIndexPath:(NSIndexPath *)indexPath
    forChangeType:(NSFetchedResultsChangeType)type
    newIndexPath:(NSIndexPath *)newIndexPath
{
  switch (type) {
    case NSFetchedResultsChangeInsert:
      NSLog(@"*** NSFetchedResultsChangeInsert (object)");
      [self.tableView insertRowsAtIndexPaths:@[newIndexPath]
                    withRowAnimation:UITableViewRowAnimationFade];
      break;

    case NSFetchedResultsChangeDelete:
      NSLog(@"*** NSFetchedResultsChangeDelete (object)");
      [self.tableView deleteRowsAtIndexPaths:@[indexPath]
                    withRowAnimation:UITableViewRowAnimationFade];
      break;
```

```
    case NSFetchedResultsChangeUpdate:
      NSLog(@"*** NSFetchedResultsChangeUpdate (object)");
      [self configureCell:[self.tableView
        cellForRowAtIndexPath:indexPath] atIndexPath:indexPath];
      break;

    case NSFetchedResultsChangeMove:
      NSLog(@"*** NSFetchedResultsChangeMove (object)");
      [self.tableView deleteRowsAtIndexPaths:@[indexPath]
                  withRowAnimation:UITableViewRowAnimationFade];
      [self.tableView insertRowsAtIndexPaths:@[newIndexPath]
                  withRowAnimation:UITableViewRowAnimationFade];
      break;
  }
}

- (void)controller:(NSFetchedResultsController *)controller
  didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
  atIndex:(NSUInteger)sectionIndex
  forChangeType:(NSFetchedResultsChangeType)type
{
  switch (type) {
    case NSFetchedResultsChangeInsert:
      NSLog(@"*** NSFetchedResultsChangeInsert (section)");
      [self.tableView insertSections:[NSIndexSet
             indexSetWithIndex:sectionIndex]
                  withRowAnimation:UITableViewRowAnimationFade];
      break;

    case NSFetchedResultsChangeDelete:
      NSLog(@"*** NSFetchedResultsChangeDelete (section)");
      [self.tableView deleteSections:[NSIndexSet
             indexSetWithIndex:sectionIndex]
                  withRowAnimation:UITableViewRowAnimationFade];
      break;
  }
}

- (void)controllerDidChangeContent:
                    (NSFetchedResultsController *)controller
{
  NSLog(@"*** controllerDidChangeContent");
  [self.tableView endUpdates];
}
```

Don't let this freak you out! This is the standard way of implementing these delegate methods. For many apps, this exact code will suffice and you can simply copy it over.

`NSFetchedResultsController` will call these methods to let you know that certain objects were inserted, removed, or just updated. In response, you call the corresponding methods on the `UITableView` to insert, remove or update rows. That's all there is to it.

I put `NSLog()` statements in these methods so you can follow along in the debug area with what is happening. Also note that you're using the `switch` statement here. A series of `if`'s would have worked just as well but `switch` reads better.

➤ Run the app. Edit an existing location and press the Done button.

The debug area now shows:

```
MyLocations[35636:a0b] *** controllerWillChangeContent
MyLocations[35636:a0b] *** NSFetchedResultsChangeUpdate (object)
MyLocations[35636:a0b] *** controllerDidChangeContent
```

`NSFetchedResultsController` noticed that an existing object was updated and, through updating the table, called your `configureCell` method to redraw the contents of the cell. By the time the Edit Location screen has disappeared from sight, the table view is updated and your change will be visible.

This also works for adding new locations.

➤ Tag a new location and press the Done button.

The debug area says:

```
MyLocations[35636:a0b] *** controllerWillChangeContent
MyLocations[35636:a0b] *** NSFetchedResultsChangeInsert (object)
MyLocations[35636:a0b] *** controllerDidChangeContent
```

This time it's an "insert" notification. The delegate methods told the table view to do `insertRowsAtIndexPaths` in response and the new `Location` object is inserted in the table.

That's how easy it is. You make a new `NSFetchedResultsController` object with a fetch request and implement the delegate methods. The fetched results controller keeps an eye on any changes that you make to the data store and notifies its delegate. It doesn't matter where in the code you make these changes – the fetched results controller picks up on them right away. For most apps you can simply copy-paste the delegate methods above.

Note: There is a nasty bug with Core Data in iOS 7.0. Here is how you can reproduce it:

1. Quit the app.
2. Run the app again and tag a new location.
3. Switch to the Locations tab.

You'd expect the new location to appear in the Locations tab, but it doesn't. Instead, the app crashes as soon as you switch tabs. The error message is:

```
CoreData: FATAL ERROR: The persistent cache of section information does
not match the current configuration.  You have illegally mutated the
NSFetchedResultsController's fetch request, its predicate, or its sort
descriptor without either disabling caching or using
+deleteCacheWithName:
```

This does not happen when you switch to the Locations tab before you tag a new location. It also does not happen on iOS 6. So far the only fix I've found is to clear out the cache of the `NSFetchedResultsController`. If this problem also affects you, then add the following line to `viewDidLoad`, before the call to `performFetch`:

```
[NSFetchedResultsController deleteCacheWithName:@"Locations"];
```

Deleting Locations

Everyone makes mistakes so it's likely that users will want to delete locations from their list at some point. This is a very easy feature to add: you just have to remove the `Location` object from the data store and the `NSFetchedResultsController` will make sure it gets dropped from the table (again, through its delegate methods).

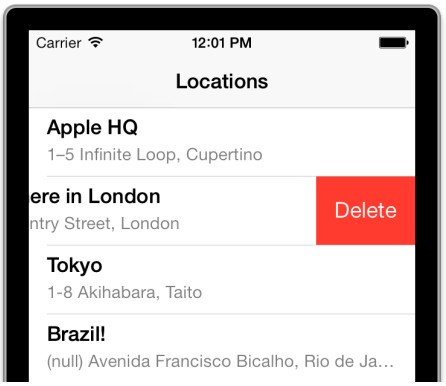
➤ Add the following method to **LocationsViewController.m**:

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
     forRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (editingStyle == UITableViewCellEditingStyleDelete) {
    Location *location = [self.fetchedResultsController
                                    objectAtIndexPath:indexPath];

    [self.managedObjectContext deleteObject:location];

    NSError *error;
    if (![self.managedObjectContext save:&error]) {
      FATAL_CORE_DATA_ERROR(error);
      return;
    }
  }
}
```

You've seen `commitEditingStyle` before. It's part of the table view's data source protocol. As soon as you implement this method in your view controller, you enable swipe-to-delete.

This method gets the `Location` object from the selected row and then tells the context to delete that object. This will trigger the `NSFetchedResultsController` to send a notification to the delegate (`NSFetchedResultsChangeDelete`), which then removes the corresponding row from the table. That's all you need to do!

➤ Run the app and remove a location using swipe-to-delete. The `Location` object is dropped from the database and its row disappears from the screen with a brief animation.



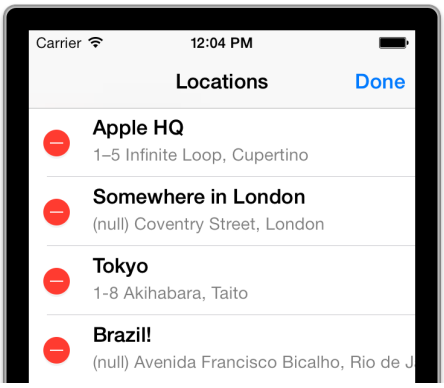
Swipe to delete rows from the table

Many apps have an Edit button in the navigation bar that triggers a mode that also lets you delete (and sometimes move) rows. This is extremely easy to add.

➤ Add the following line to `viewDidLoad`:

```
self.navigationItem.rightBarButtonItem = self.editButtonItem;
```

That's all there is to it. Every view controller has a built-in Edit button that can be accessed through the `editButtonItem` property. Tapping that button puts the table in editing mode:



Putting the screen into Edit mode

➤ Run the app and verify that you can now also delete rows by pressing the Edit button.

Pretty cool, huh. There's more cool stuff that `NSFetchedResultsController` makes really easy, such as splitting up the rows into sections.

Table View Sections

The `Location` objects have a `category` field. It would be nice to group the locations by category in the table. The table view supports organizing rows into sections and each of those sections can have its own header. Putting your rows into sections is a lot of work if you're doing it by hand, but `NSFetchedResultsController` practically gives you section support for free.

➤ Change the creation of the sort descriptors in the `fetchedResultsController` method:

```
- (NSFetchedResultsController *)fetchedResultsController
{
  if (_fetchedResultsController == nil) {

    . . .

    NSSortDescriptor *sortDescriptor1 = [NSSortDescriptor
              sortDescriptorWithKey:@"category" ascending:YES];
    NSSortDescriptor *sortDescriptor2 = [NSSortDescriptor
                  sortDescriptorWithKey:@"date" ascending:YES];
    [fetchRequest setSortDescriptors:
                         @[sortDescriptor1, sortDescriptor2]];

    . . .
```

Instead of one sort descriptor object, you now have two. First this sorts the `Location` objects by `category` and inside each of these groups it sorts by `date`.

Note: When you write `@[ object1, object2, object3 ]`, the objects in between the `@[` and `]` brackets are added to a new array object. This notation is a recent addition to the language. In older source code you may find this written as:

```
[NSArray arrayWithObjects:object1, object2, object3, nil]
```

That does the exact same thing. The form `[NSArray arrayWithObjects:...]` requires a `nil` at the end of the list to indicate to the compiler that there are no more items following, but the new `@[ ... ]` syntax doesn't.

➤ Also change the initialization of the `NSFetchedResultsController` object:

```
. . .

_fetchedResultsController = [[NSFetchedResultsController alloc]
      initWithFetchRequest:fetchRequest
      managedObjectContext:self.managedObjectContext
      sectionNameKeyPath:@"category"
      cacheName:@"Locations"];

. . .
```

The only difference here is that the `sectionNameKeyPath` parameter is now `@"category"`, which means the fetched results controller will group the search results based on the value of the `category` attribute.

You're not done yet. The table view's data source also has methods for sections. So far you've only used the methods for rows, but now that you're adding sections to the table you need to implement a few additional methods.

➤ Add the following methods to the data source:

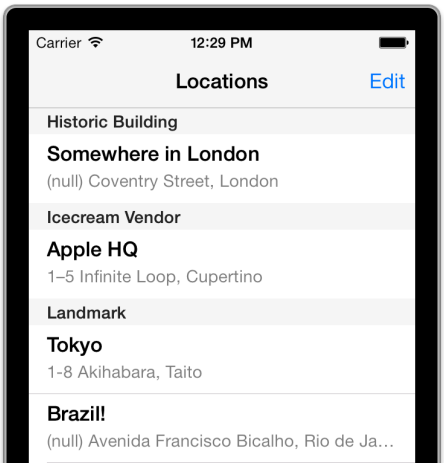
```
- (NSInteger)numberOfSectionsInTableView:
                              (UITableView *)tableView
{
  return [[self.fetchedResultsController sections] count];
}

- (NSString *)tableView:(UITableView *)tableView
             titleForHeaderInSection:(NSInteger)section
{
  id <NSFetchedResultsSectionInfo> sectionInfo =
             [self.fetchedResultsController sections][section];

  return [sectionInfo name];
```

```
}
```

Because you let `NSFetchedResultsController` do all the work already, the implementation of these methods is very simple. You ask the fetcher object for a list of the sections, which is an `NSArray` of `NSFetchedResultsSectionInfo` objects, and then look inside that array to find out how many sections there are and what their names are.

> **Note:** Don't let the notation `id <NSFetchedResultsSectionInfo>` confuse you. This just means that Core Data gives you an object that conforms to the `NSFetchedResultsSectionInfo` protocol. That protocol contains methods for obtaining the name of the section and the list of objects that belong to that section. You don't need to care about the actual datatype of the `sectionInfo` variable, only that you can treat it as a `NSFetchedResultsSectionInfo` object.

▸ Run the app. Play with the categories on the locations and notice how the table view automatically updates. All thanks to `NSFetchedResultsController`!



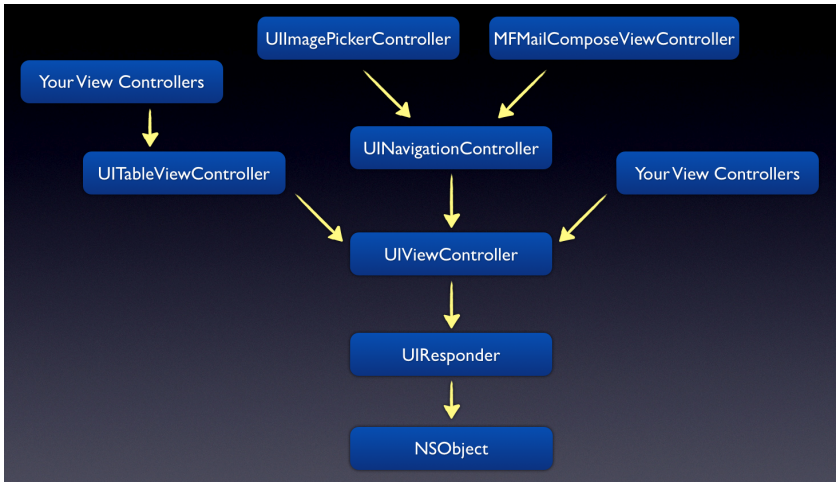
The locations are now grouped in sections

You can find the project files for this section under **06 - NSFetchedResultsController** in the tutorial's Source Code folder.

Hierarchies

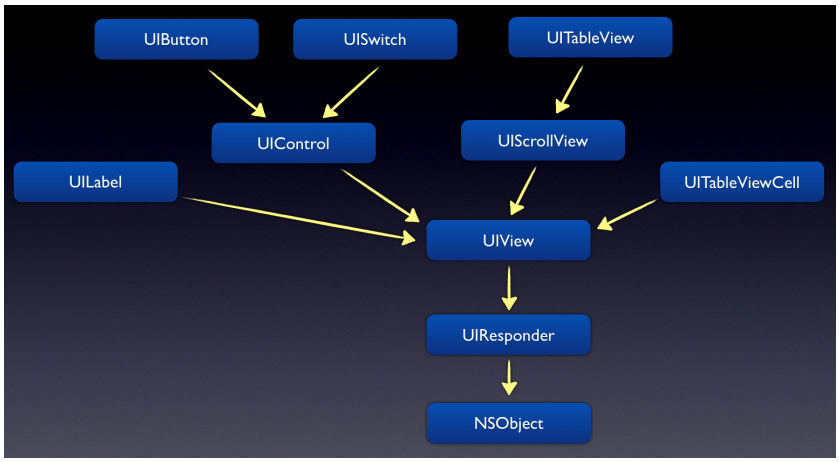
Programmers love to organize stuff into hierarchies – or **tree structures** as they like to call them. You have to admit that the following pictures do look at bit like trees. The thing at the bottom is often called the root, all the other items are branches. Keeping with the tree analogy, the items at the very ends are known as the leaves.

There is the inheritance hierarchy of view controller classes:



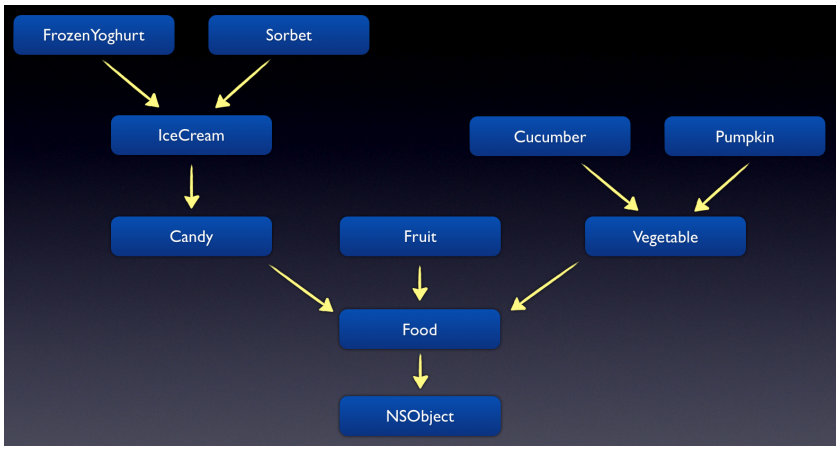
UIViewController inheritance tree (partial)

There is also an inheritance hierarchy of view classes:



UIView and some of its subclasses

Often your data model classes will also have superclasses and subclasses:



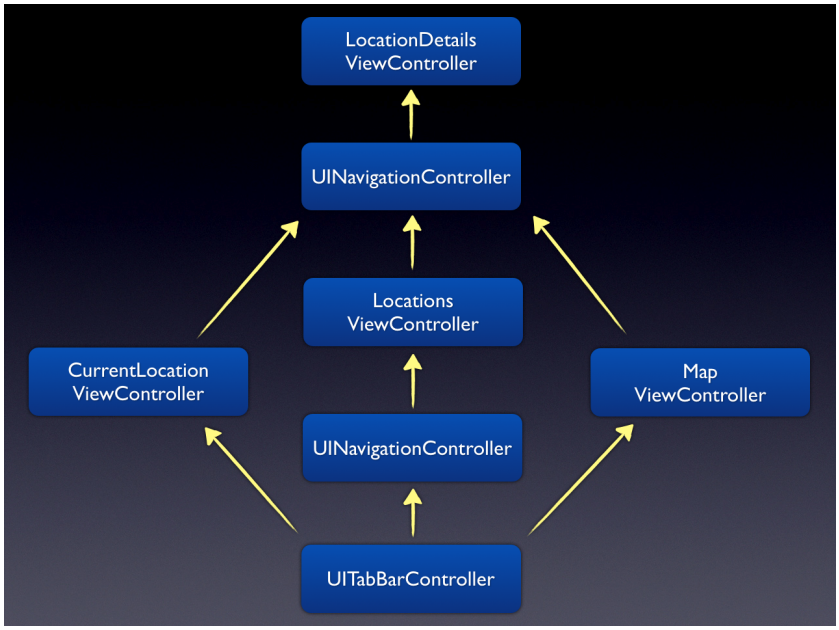
Data model class hierarchy for a grocery store

The arrows in these diagrams all represent **is-a** relationships between the classes. A `UITableViewController` *is* a `UIViewController`, but with extra stuff. Likewise, a `UIButton` is a `UIView` that also knows how to respond to touches and can initiate actions when tapped.

Often people use the terms **parent** and **child** when they talk about such hierarchies. `UIViewController` is the parent of `UITableViewController`, while `UIButton` is a child of `UIView` and `UIControl`.

The above hierarchies are between datatypes. The object graph, on the other hand, is a hierarchy between actual object instances. The type hierarchy is used only during the construction of your programs and is fixed, but the object graph can change dynamically while your app is running as new relationships are forged and old ones are broken.

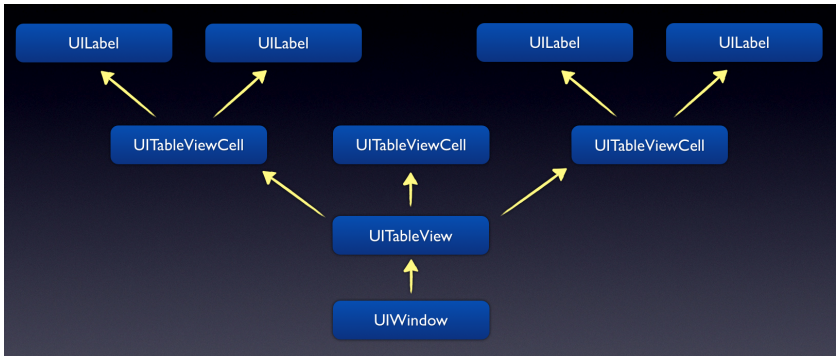
The flow of screens in the MyLocations app can be expressed as a hierarchy between view controllers, with the `UITabBarController` as the root at the bottom:



The flow between the screens in the app is also a hierarchy

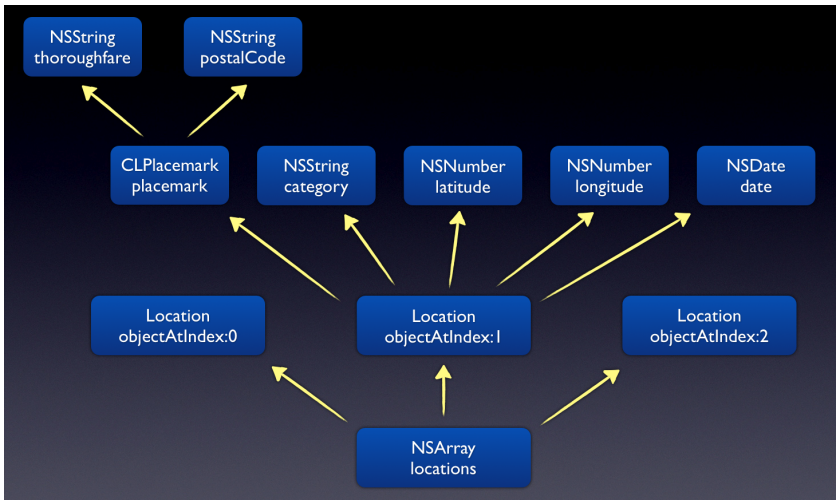
This is essentially what your storyboard represents. Here the terms parent and child are used also: the `UITabBarController` is the parent of the view controllers for the Tag, Locations and Map screens.

Views on the screen also have a hierarchy of subviews. The view controller's main view provides the backdrop for the screen, and it has many view objects layered on top of it:



The view and subview objects in a screen with a table view

The graph of data model objects can also form a hierarchy:



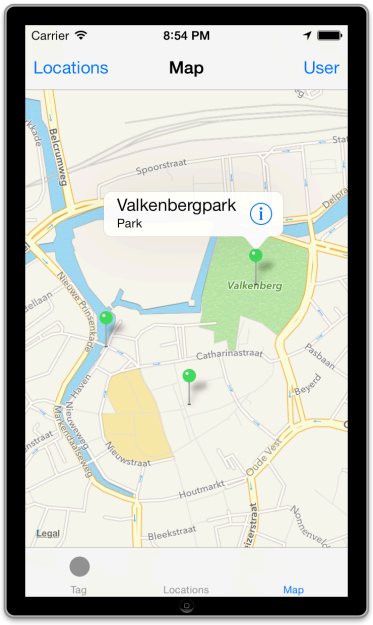
The object graph of data model objects

These are all known as **has-a** relationships. The `NSArray` instance *has* one or more `Location` objects (its children). Each `Location` object has `latitude`, `longitude`, `category` and `placemark` objects. And the `CLPlacemark` instance has a bunch of string objects of its own.

The world of computing is rife with examples of hierarchies and tree structures. For example, the contents of a plist or XML file, the organization of your source files into groups in the Xcode project, the file system itself with its infinitely nested directories, get-rich-quick Ponzi schemes, you name it.

Pins on a map

Showing the locations in a table view is useful but not very visually appealing. The iOS SDK comes with an awesome map view control and it would be a shame not to use it. In this section you will add third tab to the app that looks like this:



The Map screen with some locations in my home town

› First visit: the storyboard. From the Objects Library, drag a regular **View Controller** into the canvas. **Ctrl-drag** from the Tab Bar Controller to this new View Controller to add it to the tabs.

› The new view controller now has a **Tab Bar Item**. Rename it to say **Map**.

› Select the view controller and in the **Identity inspector** set its **Class** to **MapViewController**. You haven't made this class yet but you will do so soon.

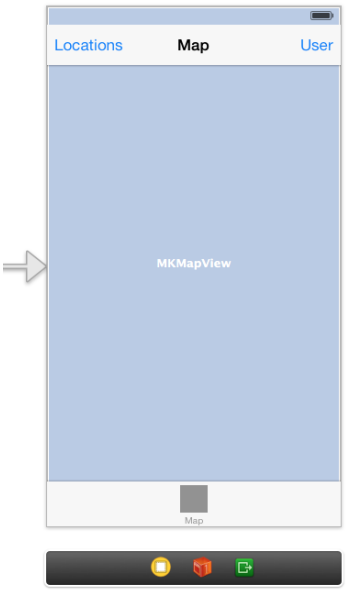
› Drag a **Map View** into the view controller. Make it cover the entire area of the screen, so that it sits below the tab bar. (The size of the Map View should be 320 × 568 points.)

› In the **Attributes inspector** for the Map View, enable **Shows User Location**. That will put a blue dot on the map at the user's current coordinates.

› Drag a **Navigation Bar** into the Map View Controller. Place it against the bottom of the status bar, at Y = 20. Change its title to **Map**. The navigation bar partially overlaps the Map View.

› Drag a **Bar Button Item** into the left-hand slot of the navigation bar and give it the title **Locations**. Drag another into the right-hand slot but name it **User**. Later on you'll use nice icons for these buttons but for now these labels will do.

This part of the storyboard should look like this:

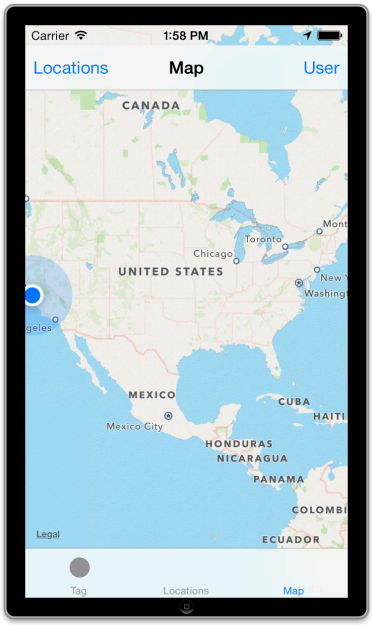


The design of the Map screen

The app won't run as-is. `MKMapView` is not part of UIKit but of MapKit, so you need to add the MapKit framework first. (The app will compile without problems but crashes when you switch to the Map screen.)

› In the **Linked Frameworks and Libraries** section, add **MapKit.framework**.

› Run the app. Choose a location in the Simulator's Debug menu and switch to the Map. The screen should look something like this:



The map shows the user's location

> **Note:** There is a gap between the top of the screen and the navigation bar. That is because on iOS 7 the status bar is no longer a separate area but is directly drawn on top of the view controller. You will fix this later in the tutorial.

Next, you're going to show the user's location in a little more detail because the blue dot could be almost anywhere in California!

▸ Add the following import to the **MyLocations-Prefix.pch** file, so that MapKit can be used anywhere inside the project:

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

▸ Add a new `UIViewController` subclass file to the project. Name it **MapViewController**.

▸ Add a property for the managed object context to **MapViewController.h**:

```
@interface MapViewController : UIViewController

@property (nonatomic, strong) NSManagedObjectContext
                                        *managedObjectContext;

@end
```

▸ Replace the contents of **MapViewController.m** with the following:

```
#import "MapViewController.h"

@interface MapViewController () <MKMapViewDelegate>

@property (nonatomic, weak) IBOutlet MKMapView *mapView;

@end

@implementation MapViewController

- (IBAction)showUser
{
  MKCoordinateRegion region =
          MKCoordinateRegionMakeWithDistance(
              self.mapView.userLocation.coordinate, 1000, 1000);

  [self.mapView setRegion:[self.mapView regionThatFits:region]
```

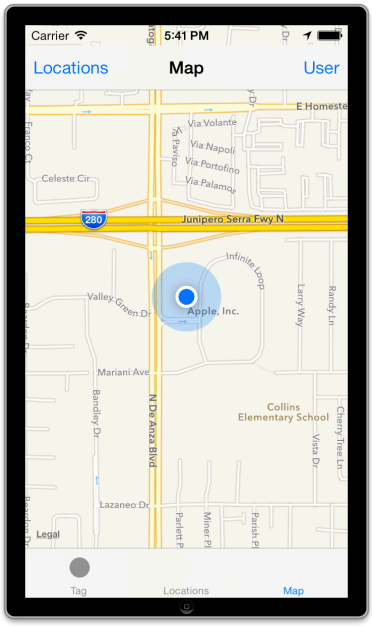
```
                    animated:YES];
}

- (IBAction)showLocations
{
}

@end
```

This has a private outlet property for the map view and two action methods that will be connected to the buttons in the navigation bar. The view controller is also the delegate of the map view.

➤ In the Map screen in the storyboard editor, connect the Locations bar button item with the `showLocations` action and the User button with the `showUser` action.

➤ Connect the Map View with the `mapView` outlet, and its delegate with the view controller.

Currently the view controller only implements the `showUser` action. When you press the **User** button, it zooms in the map to a region that is 1000 by 1000 meters (a little more than half a mile in both directions) around the user's position:



Pressing the User button zooms in to the user's location

The other button, Locations, will show the region that contains all the user's saved locations. Before you can do that, you first have to fetch those locations from the data store.

Even though this screen doesn't have a table view, you could still use an `NSFetchedResultsController` object to handle all the fetching and automatic change detection. But this time I want to make it hard on you, so you're going to do the fetching by hand.

▸ Add a new array to **MapViewController.m**:

```
@implementation MapViewController
{
  NSArray *_locations;
}
```

▸ Add the `updateLocations` method:

```
- (void)updateLocations
{
  NSEntityDescription *entity = [NSEntityDescription
              entityForName:@"Location"
              inManagedObjectContext:self.managedObjectContext];

  NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
  [fetchRequest setEntity:entity];

  NSError *error;
  NSArray *foundObjects = [self.managedObjectContext
                executeFetchRequest:fetchRequest error:&error];
  if (foundObjects == nil) {
    FATAL_CORE_DATA_ERROR(error);
    return;
  }

  if (_locations != nil) {
    [self.mapView removeAnnotations:_locations];
  }

  _locations = foundObjects;
  [self.mapView addAnnotations:_locations];
}
```

The fetch request is nothing new, except this time you're not sorting the `Location` objects. The order of the `Location` objects in the array doesn't really matter to the map view, only their latitude and longitude coordinates.

Once you've obtained the `Location` objects, you call `addAnnotations` on the map view to add a pin for each location on the map.

The idea is that `updateLocations` will be executed every time there is a change in the data store. How you'll do that is of later concern, but the point is that the

`_locations` array may already exist and may contain `Location` objects. If so, you first tell the map view to remove the pins for these old objects.

➤ Add the `viewDidLoad` method below `updateLocations`:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  [self updateLocations];
}
```

This fetches the `Location` objects and shows them on the map when the view loads. Nothing special here.

Before this class can use the `managedObjectContext`, you have to give it a pointer to that object first. As before, that happens in `AppDelegate`.

➤ In **AppDelegate.m**, add an import for `MapViewController`:

```
#import "MapViewController.h"
```

➤ And extend `didFinishLaunchingWithOptions` to pass the context object to the `MapViewController` as well:

```
MapViewController *mapViewController =
       (MapViewController *)tabBarController.viewControllers[2];
mapViewController.managedObjectContext =
                                      self.managedObjectContext;
```

You're not quite done yet. The app builds without problems but crashes when you switch to the Map tab.

In `updateLocations` you tell the map view to add the `Location` objects as annotations (an annotation is a pin on the map). But `MKMapView` expects an array of `MKAnnotation` objects, not your own `Location` class. Luckily, `MKAnnotation` is a protocol, so you can turn the `Location` objects into map annotations by making the class conform to that protocol.

➤ Change the `@interface` line from **Location.h** to:

```
@interface Location : NSManagedObject <MKAnnotation>
```

Just because `Location` is an object that is managed by Core Data doesn't mean you can't add your own stuff to it. It's still an object!

The `MKAnnotation` protocol requires that the class implements the getters for three properties: `coordinate`, `title` and `subtitle`. It obviously needs to know the coordinate in order to place the pin in the correct place on the map. The title and subtitle are used for the "call-out" that appears when you tap on the pin.

▸ Add the following getter methods to **Location.m**:

```
- (CLLocationCoordinate2D)coordinate
{
  return CLLocationCoordinate2DMake(
      [self.latitude doubleValue], [self.longitude doubleValue]);
}

- (NSString *)title
{
  if ([self.locationDescription length] > 0) {
    return self.locationDescription;
  } else {
    return @"(No Description)";
  }
}

- (NSString *)subtitle
{
  return self.category;
}
```

▸ Run the app and switch to the Map screen. It should now show pins for the saved locations. If you tap on a pin, the callout shows the chosen description and category.

The map shows pins for the saved locations

When you press the User button, the app zooms into the user's location but the same thing doesn't happen yet for the Locations button and the location pins. By looking at the highest and lowest values for the latitude and longitude of all the `Location` objects, you can calculate a region and then tell the map view to zoom to that region.

➤ In **MapViewController.m**, change the `showLocations` method to:

```
- (IBAction)showLocations
{
  MKCoordinateRegion region = [self
                               regionForAnnotations:_locations];

  [self.mapView setRegion:region animated:YES];
}
```

This calls a new method, `regionForAnnotations`, to calculate a reasonable region that fits all `Location` objects and then sets that region on the map view. You'll add `regionForAnnotations` in a second.

➤ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  [self updateLocations];

  if ([_locations count] > 0) {
    [self showLocations];
  }
}
```

It's a good idea to show the user's locations by default the first time you switch to the Map tab, so `viewDidLoad` calls `showLocations` too.

➤ Add the `regionForAnnotations` method below `showLocations`:

```
- (MKCoordinateRegion)regionForAnnotations:
                                    (NSArray *)annotations
{
  MKCoordinateRegion region;

  if ([annotations count] == 0) {
    region = MKCoordinateRegionMakeWithDistance(
              self.mapView.userLocation.coordinate, 1000, 1000);

  } else if ([annotations count] == 1) {
    id <MKAnnotation> annotation = [annotations lastObject];
```

```
    region = MKCoordinateRegionMakeWithDistance(
                              annotation.coordinate, 1000, 1000);

  } else {
    CLLocationCoordinate2D topLeftCoord;
    topLeftCoord.latitude = -90;
    topLeftCoord.longitude = 180;

    CLLocationCoordinate2D bottomRightCoord;
    bottomRightCoord.latitude = 90;
    bottomRightCoord.longitude = -180;

    for (id <MKAnnotation> annotation in annotations) {

      topLeftCoord.latitude = fmax(topLeftCoord.latitude,
                                   annotation.coordinate.latitude);

      topLeftCoord.longitude = fmin(topLeftCoord.longitude,
                                  annotation.coordinate.longitude);

      bottomRightCoord.latitude = fmin(
                                   bottomRightCoord.latitude,
                                   annotation.coordinate.latitude);

      bottomRightCoord.longitude = fmax(
                                  bottomRightCoord.longitude,
                                  annotation.coordinate.longitude);
    }

    const double extraSpace = 1.1;

    region.center.latitude = topLeftCoord.latitude -
                (topLeftCoord.latitude -
                                   bottomRightCoord.latitude) / 2.0;

    region.center.longitude = topLeftCoord.longitude -
                (topLeftCoord.longitude -
                                  bottomRightCoord.longitude) / 2.0;

    region.span.latitudeDelta = fabs(topLeftCoord.latitude -
                         bottomRightCoord.latitude) * extraSpace;

    region.span.longitudeDelta = fabs(topLeftCoord.longitude -
                        bottomRightCoord.longitude) * extraSpace;
  }
```

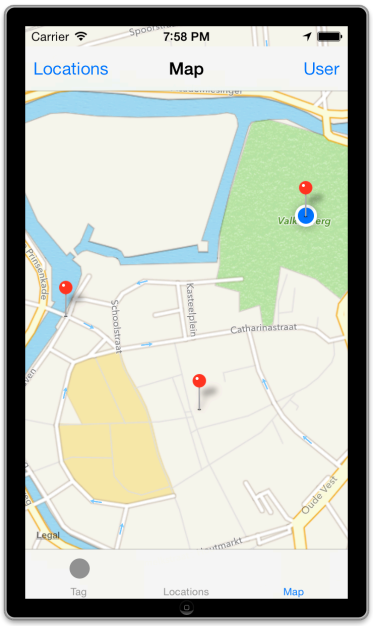
```
    return [self.mapView regionThatFits:region];
}
```

`regionForAnnotations` has three situations to handle:

1. There are no annotations. In that case you'll center the map on the user's current position.
2. There is only one annotation. You'll center the map on that one annotation.
3. There are two or more annotations. You'll calculate the extent of their reach and add a little padding.

Note that this method does not use `Location` objects for anything. It assumes that all the objects in the array conform to the `MKAnnotation` protocol and it only looks at that part of the objects. As far as `regionForAnnotations` is concerned, what it deals with are annotations. That means you can use this method in any app that uses Map Kit, without modifications. Pretty neat.

➤ Run the app and press the Locations button. The map view should now zoom in on your saved locations. (This only works well if the locations aren't too far apart, of course.)



The map view zooms in to fit all your saved locations

You made the `MapViewController` conform to the `MKMapViewDelegate` protocol but so far you haven't done anything with that. This delegate is useful for creating your own annotation views. Currently a default pin and callout are being used, but you can change this to anything you like.

➤ First add an import at the top of **MapViewController.m**:

```
#import "Location.h"
```

➤ Add the following code to the bottom of **MapViewController.m**:

```
#pragma mark - MKMapViewDelegate

- (MKAnnotationView *)mapView:(MKMapView *)mapView
            viewForAnnotation:(id <MKAnnotation>)annotation
{
  // 1
  if ([annotation isKindOfClass:[Location class]]) {

    // 2
    static NSString *identifier = @"Location";
    MKPinAnnotationView *annotationView =
      (MKPinAnnotationView *)[self.mapView
        dequeueReusableAnnotationViewWithIdentifier:identifier];

    if (annotationView == nil) {
      annotationView = [[MKPinAnnotationView alloc]
                                  initWithAnnotation:annotation
                                     reuseIdentifier:identifier];
      // 3
      annotationView.enabled = YES;
      annotationView.canShowCallout = YES;
      annotationView.animatesDrop = NO;
      annotationView.pinColor = MKPinAnnotationColorGreen;

      // 4
      UIButton *rightButton = [UIButton
                   buttonWithType:UIButtonTypeDetailDisclosure];
      [rightButton addTarget:self
                      action:@selector(showLocationDetails:)
            forControlEvents:UIControlEventTouchUpInside];
      annotationView.rightCalloutAccessoryView = rightButton;

    } else {
      annotationView.annotation = annotation;
    }

    // 5
    UIButton *button =
           (UIButton *)annotationView.rightCalloutAccessoryView;
    button.tag = [_locations indexOfObject:
                                        (Location *)annotation];
```

```
    return annotationView;
  }

  return nil;
}
```

This is very similar to what a table view does with `cellForRowAtIndexPath`, except that you're not dealing with table view cells here but with `MKAnnotationView` objects. Step-by-step this is what happens:

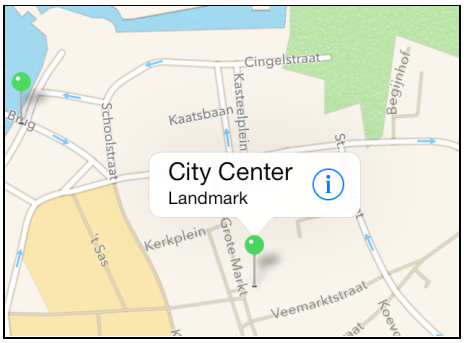
1. Because `MKAnnotation` is a protocol, there may be other objects than the `Location` object that want to be annotations on the map. An example is the blue dot that represents the user's current location. You should leave such annotations alone, so you use the `isKindOfClass:` method to determine whether the annotation is really a `Location` object. If so, you continue.
2. This should look very familiar to creating a table view cell. You ask the map view to re-use an annotation view object. If it cannot find a recyclable annotation view, then you create a new one. Note that you're not limited to `MKPinAnnotationView`. This is the standard annotation view class, but you can also create your own `MKAnnotationView` subclass and make it look like anything you want. Pins are only one option.
3. This just sets some properties to configure the look and feel of the annotation view. Previously the pins were red, but you make them green here.
4. This is the interesting bit. You create a new `UIButton` object that looks like a detail disclosure button (a blue circled **i**). You use the target-action pattern to hook up the button's "Touch Up Inside" event with the `showLocationDetails:` method, and add the button to the annotation view's accessory view.
5. Once the annotation view is constructed and configured, you obtain a reference to that detail disclosure button again and set its `tag` to the index of the `Location` object in the `_locations` array. That way you can find the `Location` object later in the `showLocationDetails:` method when the button is pressed.

➤ Add the `showLocationDetails:` method:

```
- (void)showLocationDetails:(UIButton *)button
{
}
```

Leave it empty for now. If you hadn't added this method and you'd press the button on the annotation's callout then the app would crash with an "unrecognized selector sent to instance" error.

➤ Run the app. The pins are now green and the callout has a custom button. (If the pins stay red, then make sure you connected the view controller as the delegate of the map view.)

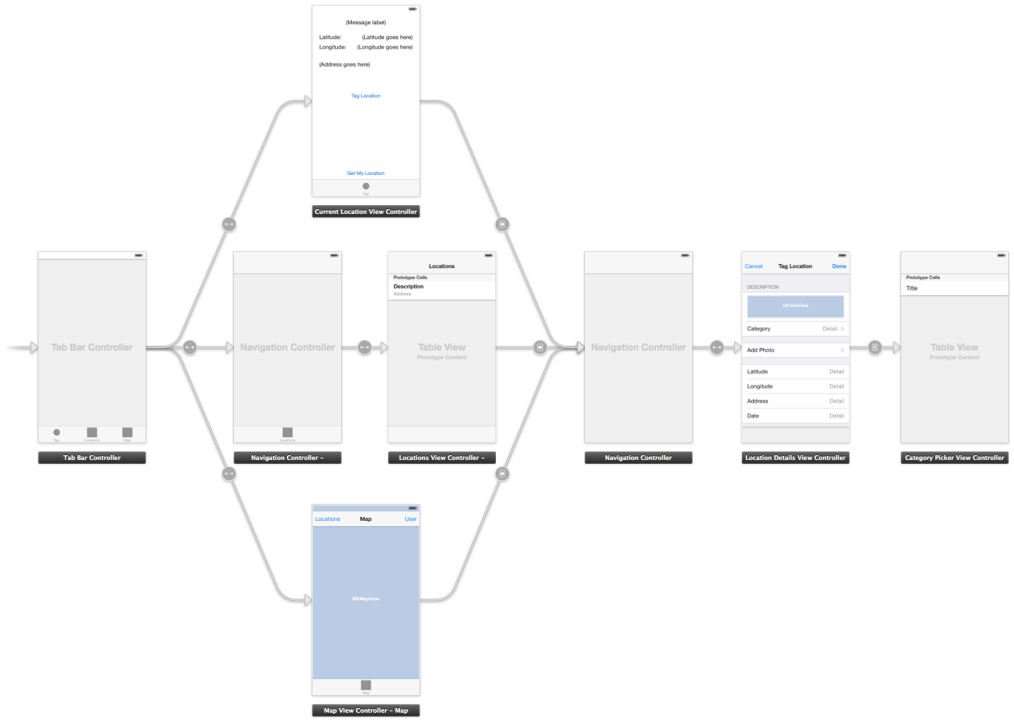


The annotations use your own view

What should this button do? Show the Edit Location screen, of course!

› Open the storyboard. Select the Map View Controller (the actual view controller, not some view inside it) and **ctrl-drag** to the Navigation Controller that contains the Location Details View Controller. Add a new **modal** segue named **EditLocation**.

The storyboard now looks like this:



The Location Details screen is connected to all three screens

I had to zoom out the Storyboard in order to make the screen capture. It's not very readable at this level but you can see that there are now three segues going to the Location Details screen.

› Add the ever-important import at the top of **MapViewController.m**:

```
#import "LocationDetailsViewController.h"
```

➤ Change the `showLocationDetails:` method to:

```
- (void)showLocationDetails:(UIButton *)button
{
  [self performSegueWithIdentifier:@"EditLocation"
                            sender:button];
}
```

Because the segue isn't connected to any particular control in the view controller, you have to trigger it manually. You send along the button object as the `sender`, so you can read its `tag` property in `prepareForSegue`.

➤ Add the `prepareForSegue` method:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"EditLocation"]) {

    UINavigationController *navigationController =
                                segue.destinationViewController;

    LocationDetailsViewController *controller =
           (LocationDetailsViewController *)
                        navigationController.topViewController;

    controller.managedObjectContext = self.managedObjectContext;

    UIButton *button = (UIButton *)sender;
    Location *location = _locations[button.tag];
    controller.locationToEdit = location;
  }
}
```

This is very similar to what you did in the Locations screen, except that now you get the `Location` object to edit from the `_locations` array, using the `tag` property of the button as the index in that array.

➤ Run the app, tap on a pin and edit the location.

It works, except that the annotation on the map doesn't change until you tap the pin again. (The cells in the Locations screen changed immediately because you're using the `NSFetchedResultsController` there.) Likewise, changes on the other screens, such as adding or deleting a location, have no effect on the map.

This is the same problem you had earlier with the Locations screen: because the list of `Location` objects is only fetched once in `viewDidLoad`, any changes are overlooked.

The way you're going to fix this for the Map screen is by using notifications. Recall that you have already put `NSNotificationCenter` to use for dealing with Core Data save errors. As it happens, Core Data also sends out a bunch of notifications when certain changes happen to the data store. You can subscribe to these notifications and update the map view when you receive them.

▸ Add an `initWithCoder:` method to **MapViewController.m**:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder]))
  {
    [[NSNotificationCenter defaultCenter] addObserver:self
      selector:@selector(contextDidChange:)
      name:NSManagedObjectContextObjectsDidChangeNotification
      object:self.managedObjectContext];
  }
  return self;
}
```

This tells the `NSNotificationCenter` to add `self`, i.e. this view controller, as an observer for the `NSManagedObjectContextObjectsDidChangeNotification`. This notification with the very long name is sent out by the `managedObjectContext` whenever the data store changes. In response you would like your `contextDidChange:` method to be called.

▸ Add the `contextDidChange:` method:

```
- (void)contextDidChange:(NSNotification *)notification
{
  if ([self isViewLoaded]) {
    [self updateLocations];
  }
}
```

This couldn't be simpler: you just call `updateLocations` to fetch all the `Location` objects again. This throws away all the old pins and it makes new pins for all the newly fetched `Location` objects. Granted, it's not a very efficient method if there are hundreds of annotation objects, but for now it gets the job done.

Note: You only call `updateLocations` when the Maps screen's view is loaded. Because this screen sits in a tab, its view may not have been loaded yet when

the user tags a new location. (The view does not get loaded until the user switches to the Map tab.) In that case it makes no sense to call `updateLocations`, and it could even crash the app!

You should tell the `NSNotificationCenter` to stop sending these notifications when the view controller is destroyed. You don't want `NSNotificationCenter` to send notifications to an object that no longer exists, that's just asking for trouble!

➤ Add a `dealloc` method that unregisters the notification:

```
- (void)dealloc
{
  [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

➤ Run the app. First go to the Map screen to make sure it is loaded. Then tag a new location. The map should have added a new pin for it, although you may have to press the Locations bar button to make the new pin appear if it's outside the visible range.

Exercise. The `contextDidChange:` method is given an `NSNotification` parameter. This object has a `userInfo` dictionary. From that dictionary it is possible to figure out which objects were inserted/deleted/updated. Try to make the reloading of the locations more efficient by not re-fetching the entire list of `Location` objects, but by only inserting or deleting those that have changed. Good luck! (You can find the solutions from other readers on the raywenderlich.com forums.) □

That's it for the Map screen. Oh, one more thing. To fix the issue with the status bar you need to make the view controller the delegate for the navigation bar.

➤ Add the delegate protocol to the class extension:

```
@interface MapViewController () <MKMapViewDelegate,
                                 UINavigationBarDelegate>
```

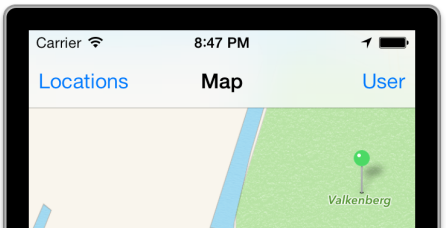
➤ And add the following method:

```
#pragma mark - UINavigationBarDelegate

- (UIBarPosition)positionForBar:(id <UIBarPositioning>)bar
{
  return UIBarPositionTopAttached;
}
```

This tells the navigation bar to extend under the status bar area.

➤ Finally, **ctrl-drag** from the navigation bar to the view controller inside the storyboard to make the delegate connection.

Now the gap between the navigation bar and the top of the screen is gone:



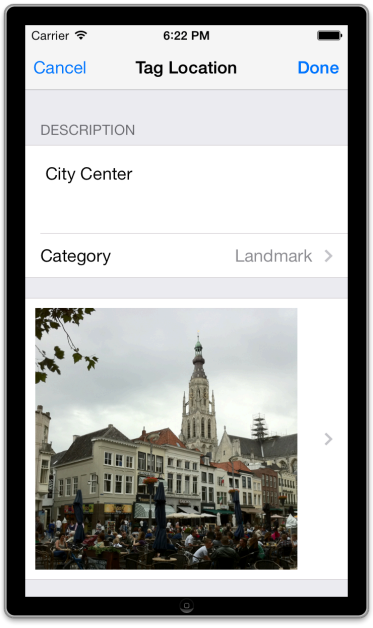
The navigation bar extends to the top of the screen

You can find the project files for the app up to this point under **07 - Map View** in the tutorial's Source Code folder.

The photo picker

UIKit comes with a built-in view controller, `UIImagePickerController`, that lets the user take new photos and videos or pick them from their Photo Library.

You're going to make the Add Photo button from the Tag/Edit Location screen work and you'll save the photo along with the location so the user has a nice picture to look at.



A photo in the Tag Location screen

› In **LocationDetailsViewController.m**, change `didSelectRowAtIndexPath:`

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
```

```
{
  if (indexPath.section == 0 && indexPath.row == 0) {
    [self.descriptionTextView becomeFirstResponder];
  }
  else if (indexPath.section == 1 && indexPath.row == 0) {
    [self takePhoto];
  }
}
```

Add Photo is the first row in the second section. When it's tapped, you call the new `takePhoto` method.

➤ Add the `takePhoto` method to the class. Put it somewhere above the table view delegate methods:

```
- (void)takePhoto
{
  UIImagePickerController *imagePicker =
                          [[UIImagePickerController alloc] init];
  imagePicker.sourceType =
                         UIImagePickerControllerSourceTypeCamera;
  imagePicker.delegate = self;
  imagePicker.allowsEditing = YES;
  [self presentViewController:imagePicker animated:YES
                   completion:nil];
}
```

The `UIImagePickerController` is a view controller like any other, but it comes with UIKit and it takes care of the entire process of taking new photos and picking them from the user's photo library. All you need to do is create a `UIImagePickerController` instance, set its properties to configure the picker, set its delegate, and then present it. When the user closes the image picker screen, the delegate methods will let you know what happened. That's exactly how you've been designing your own view controllers. (You don't add the `UIImagePickerController` to the storyboard, though.)

➤ Add the following methods to the bottom of the source file:

```
#pragma mark - UIImagePickerControllerDelegate

- (void)imagePickerController:(UIImagePickerController *)picker
        didFinishPickingMediaWithInfo:(NSDictionary *)info
{
  [self dismissViewControllerAnimated:YES completion:nil];
}
```

```
- (void)imagePickerControllerDidCancel:
                                (UIImagePickerController *)picker
{
  [self dismissViewControllerAnimated:YES completion:nil];
}
```

Currently these delegate methods simply remove the image picker from the screen. Soon you'll be taking the image the user picked and add it to the `Location` object, but for now you just want to see that you can make the image picker show up.

▸ Put the delegate protocol in the class extensions's `@interface` line:

```
@interface LocationDetailsViewController () <UITextViewDelegate,
UIImagePickerControllerDelegate, UINavigationControllerDelegate>
```

The view controller must conform to both `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate` for this to work, but you don't have to implement any of the `UINavigationControllerDelegate` methods.

▸ Run the app and press Add Photo.

If you're running the app in the Simulator, then bam! It crashes. The error message is this:

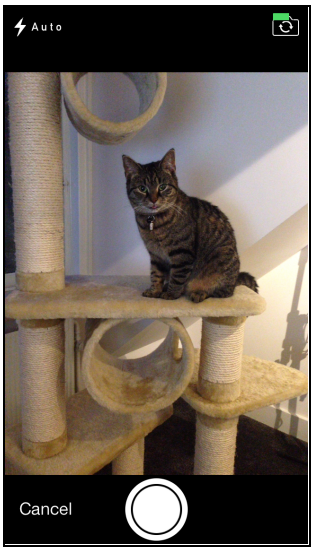
```
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: 'Source type 1 not available'
```

The culprit for the crash is the line:

```
imagePicker.sourceType =
                    UIImagePickerControllerSourceTypeCamera;
```

Not all devices have a camera, and neither does the Simulator. If you try to use the `UIImagePickerController` with a `sourceType` that is not supported by the device or the Simulator, the app crashes.

If you run the app on your device – and if it has a camera (which it probably does if it's a recent model) – then you should see this:



The camera interface

That is very similar to what you see when you take pictures using the iPhone's Camera app. (MyLocations doesn't let you record video, but you can certainly enable this feature in your own apps.)

You can still test the image picker on the Simulator, but instead of using the camera you have to use the photo library.

➤ Add the `choosePhotoFromLibrary` method below `takePhoto`:

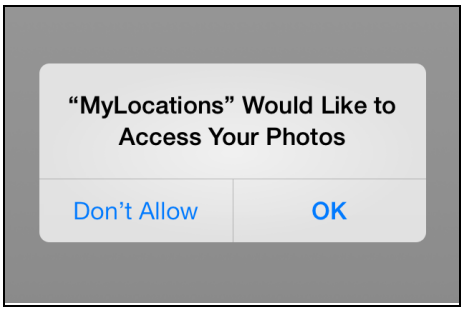
```
- (void)choosePhotoFromLibrary
{
  UIImagePickerController *imagePicker =
                          [[UIImagePickerController alloc] init];
  imagePicker.sourceType =
                  UIImagePickerControllerSourceTypePhotoLibrary;
  imagePicker.delegate = self;
  imagePicker.allowsEditing = YES;
  [self presentViewController:imagePicker animated:YES
        completion:nil];
}
```

These two methods do essentially the same, except now you set the `sourceType` to `UIImagePickerControllerSourceTypePhotoLibrary`.

➤ Change `didSelectedRowAtIndexPath` to call `choosePhotoFromLibrary` instead of `takePhoto`.

➤ Run the app in the Simulator and press Add Photo.

First, the user needs to give the app permission to access his photo library:



The user needs to allow the app to access the photo library

If they tap Don't Allow, the photo picker screen remains empty. (Users can undo this choice in the Settings app, under Privacy → Photos.)

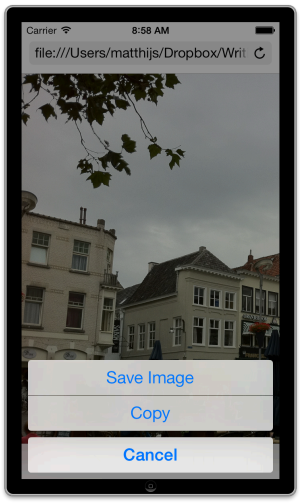
➤ Choose **OK**. At this point you'll probably see this:

There are no photos in the library

Older versions of the iOS SDK used to come with a handful of sample photos in the Photo Library, but these days you'll have to add your own.

➤ Stop the app and click on the built-in **Photos** app in the Simulator. This gives you the same "No Photos or Videos" screen. Exit the Photos app. (I found this step to be necessary, otherwise the next steps won't work.)

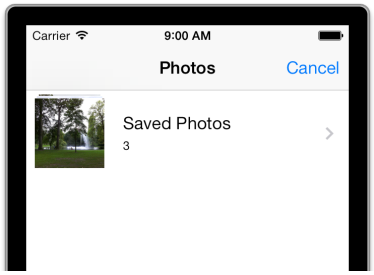
There are two ways you can add new photos to the Simulator. You can go into **Safari** (on the Simulator) and search the internet for an image. Then press down on the image until a menu appears and choose Save Image:



Adding images to the Simulator

Instead of surfing the internet for images, you can also simply drop the image file on top of the Simulator window. This opens the picture in Safari and you can add it to the library in the same way. The images end up in your Photo Library in the Saved Photos album.

- Run the app again. Now you can choose a photo from the Photo Library:



The photos in the library

- Choose one of the photos. The screen now changes to:

The user can tweak the photo

This happens because you set the image picker's `allowsEditing` property to `YES`. With this setting enabled, the user can do some quick editing on the photo before making his final choice.

So there are two types of image pickers you can use, the camera and the Photo Library, but the camera won't work everywhere. It's a bit limiting to restrict the app to just picking photos from the library, though. You'll have to make the app a bit smarter and allow the user to choose the camera when it's present.

First you check whether the camera is available. When it is, you show an **action sheet** to let the user choose between the camera and the Photo Library.

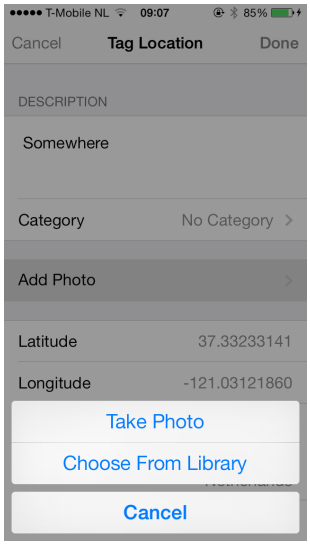
➤ Add the `showPhotoMenu` method to **LocationDetailsViewController.m**:

```
- (void)showPhotoMenu
{
  if ([UIImagePickerController isSourceTypeAvailable:
                    UIImagePickerControllerSourceTypeCamera]) {

    UIActionSheet *actionSheet = [[UIActionSheet alloc]
      initWithTitle:nil
      delegate:self
      cancelButtonTitle:@"Cancel"
      destructiveButtonTitle:nil
      otherButtonTitles:@"Take Photo", @"Choose From Library",
      nil];

    [actionSheet showInView:self.view];
  } else {
    [self choosePhotoFromLibrary];
  }
```

```
}
```

You use `UIImagePickerController`'s `isSourceTypeAvailable` method to check whether there's a camera present. If not, you call `choosePhotoFromLibrary` as that is the only option then. But when the device does have a camera you show a `UIActionSheet` on the screen. An action sheet works very much like an alert view, except that it slides in from the bottom of the screen.

- In `didSelectRowAtIndexPath`, change the call to `choosePhotoFromLibrary` to `showPhotoMenu` instead.
- Add `UIActionSheetDelegate` to the class extension.
- Run the app on your device to see the action sheet in action:



The action sheet that lets you choose between camera and photo library

Tapping any of the buttons in the action sheet simply dismisses it but doesn't do anything yet because you haven't implemented the delegate method for the action sheet.

By the way, if you want to test this action sheet on the Simulator, then you can fake the availability of the camera by replacing the line,

```
  if ([UIImagePickerController isSourceTypeAvailable:
                       UIImagePickerControllerSourceTypeCamera]) {
```

with:

```
if (YES || [UIImagePickerController isSourceTypeAvailable:
                       UIImagePickerControllerSourceTypeCamera]) {
```

That will always show the action sheet because the condition is now always true.

Also notice that the Add Photo cell remains selected (dark gray background). Normally when you show another screen in response to a tap on a cell, using a segue for example, the table view controller automatically deselects it. Here there is no segue so you should deselect the row yourself.

▸ Change `didSelectRowAtIndexPath` to:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (indexPath.section == 0 && indexPath.row == 0) {
    [self.descriptionTextView becomeFirstResponder];
  }
  else if (indexPath.section == 1 && indexPath.row == 0) {
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    [self showPhotoMenu];
  }
}
```

Before calling `showPhotoMenu`, you first deselect the Add Photo row. Try it out, it looks better this way. The cell background quickly fades from gray back to white as the action sheet slides into the screen.

▸ Add the following method to the bottom of the file:

```
#pragma mark - UIActionSheetDelegate

- (void)actionSheet:(UIActionSheet *)actionSheet
        didDismissWithButtonIndex:(NSInteger)buttonIndex
{
  if (buttonIndex == 0) {
    [self takePhoto];
  } else if (buttonIndex == 1) {
    [self choosePhotoFromLibrary];
  }
}
```

That's all you need to do to handle the action sheet. The button at index 0 is the Take Photo button and the button at index 1 is the Choose from Library button. There may be a small delay between pressing any of these buttons before the image picker appears but that's because it's a big component and iOS needs a few seconds to load it up.

By the way, if you still have the Core Data debug output enabled, then you should see a whole bunch of output in the Debug Area when the image picker is active. Apparently the `UIImagePickerController` uses Core Data as well!

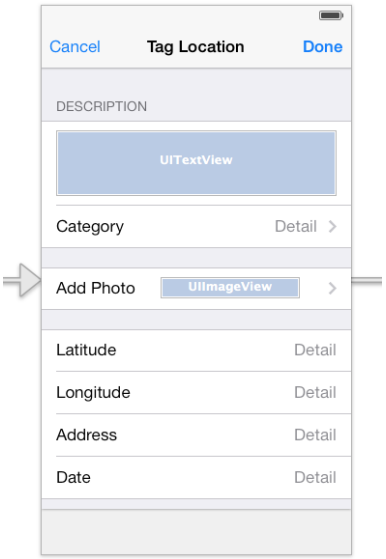
Showing the image

Now that the user can pick a photo, you should display it somewhere (otherwise, what's the point?). After picking a photo, you'll change the Add Photo cell to hold the photo. The cell will grow to fit the photo and the Add Photo label is gone.

› Add two new outlets to **LocationDetailsViewController.m**, in the class extension:

```
@property (nonatomic, weak) IBOutlet UIImageView *imageView;
@property (nonatomic, weak) IBOutlet UILabel *photoLabel;
```

› In the storyboard, drag an **Image View** into the Add Photo cell. It doesn't really matter how big it is or where you put it. You'll programmatically move it to the proper place later. (This is the reason you made this a custom cell way back when, so you could add this image view into it.)



Adding an Image View to the Add Photo cell

› Connect the Image View to the view controller's `imageView` outlet. Connect the Add Photo label to the `photoLabel` outlet.

› Select the Image View. In the **Attributes inspector**, check its **Hidden** attribute (in the Drawing section). This makes the image view initially invisible, until you have a photo to give it.

Now that you have an image view, let's make it display something.

› Add a new instance variable to **LocationDetailsViewController.m**:

```
UIImage *_image;
```

› Add the `showImage:` method to the class, below `viewDidLoad:`

```
- (void)showImage:(UIImage *)image
{
  self.imageView.image = image;
  self.imageView.hidden = NO;
  self.imageView.frame = CGRectMake(10, 10, 260, 260);
  self.photoLabel.hidden = YES;
}
```

This puts the image into the image view, makes the image view visible and gives it the proper dimensions. It also hides the Add Photo label because you don't want it to overlap the image view.

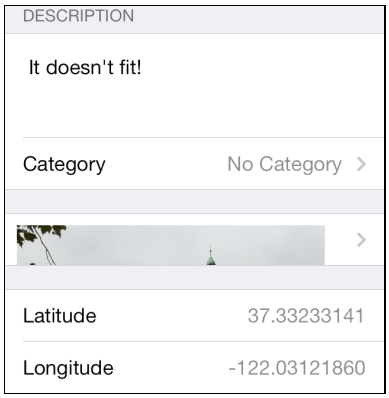
▸ Change the `didFinishPickingMediaWithInfo` method to the following:

```
- (void)imagePickerController:(UIImagePickerController *)picker
        didFinishPickingMediaWithInfo:(NSDictionary *)info
{
  _image = info[UIImagePickerControllerEditedImage];

  [self showImage:_image];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

This is the method that gets called when the user has selected a photo in the image picker. The `info` dictionary contains a variety of data describing the image that the user picked. You use the `UIImagePickerControllerEditedImage` key to retrieve a `UIImage` object that contains the image from after the Move and Scale operation. (You can also get the original image if you wish.)

Once you have the image, the call to `showImage` puts it in the Add Photo cell. Note that you also store the image in the `_image` instance variable so you can use it later.

▸ Run the app and choose a photo. Whoops, it looks like you have a small problem here:



The photo doesn't fit in the table view cell

The `showImage` method made the image view 260-by-260 points tall but the table view cell doesn't automatically resize to fit that image view. You'll have to add some logic to the `heightForRowAtIndexPath` method to accomplish that.

➤ Change the `heightForRowAtIndexPath` method:

```
- (CGFloat)tableView:(UITableView *)tableView
            heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (indexPath.section == 0 && indexPath.row == 0) {
    return 88;
  } else if (indexPath.section == 1) {
    if (self.imageView.hidden) {
      return 44;
    } else {
      return 280;
    }
  } else if (indexPath.section == 2 && indexPath.row == 2) {
    . . .
```

If there is no image, then the height for the Add Photo cell is 44 points just like a regular cell. But if there is an image, then it's a lot higher: 280 points. That is 260 points for the image view plus 10 points margin on the top and bottom.

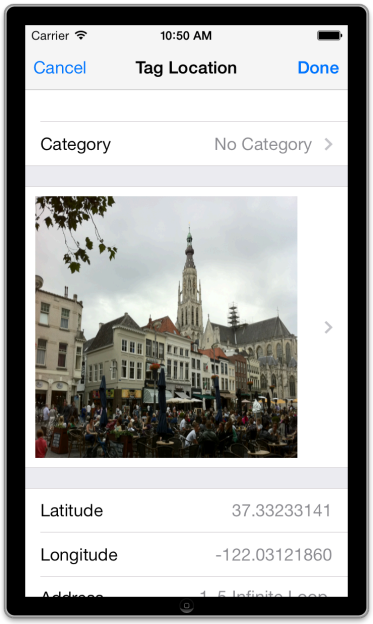
➤ Update `didFinishPickingMediaWithInfo` to the following:

```
- (void)imagePickerController:(UIImagePickerController *)picker
        didFinishPickingMediaWithInfo:(NSDictionary *)info
{
  _image = info[UIImagePickerControllerEditedImage];

  [self showImage:_image];
  [self.tableView reloadData];
```

```
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

The call to `[self.tableView reloadData]` refreshes the table view and sets the photo row to the proper height.

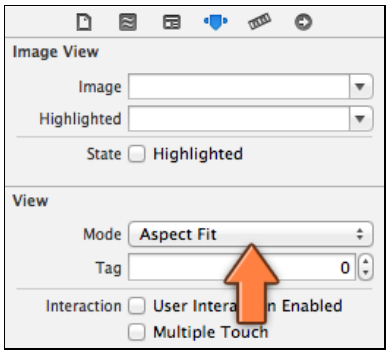
› Try it out. The row now resizes and is big enough for the whole photo. The image does appear to be stretched out a little, though.



The photo is stretched out a bit

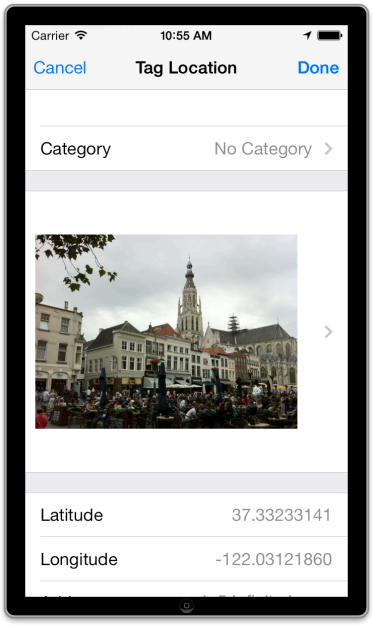
The image view is square but most photos won't be. By default, an image view will stretch the image to fit the entire content area. That's probably not what you want for this app.

› Go to the storyboard and select the Image View (it may be hard to see on account of it being hidden, but you can still find it in the outline pane). In the **Attributes inspector**, set its **Mode** to **Aspect Fit**.



Changing the image view's content mode

This will keep the image's aspect ratio intact as it's resized to fit into the image view. Play a bit with the other content modes to see what they do. (Aspect Fill is similar to Aspect Fit, except that it tries to fill up the entire view.)



The aspect ratio of the photo is kept intact

That looks a bit better, although there are now larger margins at the top and bottom of the image.

Exercise. Make the height of the photo table view cell dynamic, depending on the aspect ratio of the image. This is a tough one! You can keep the width of the image view at 260 points. This should correspond to the width of the `UIImage` object. You can get the aspect ratio of the image by doing `image.size.width / image.size.height`. With this ratio you can calculate what the height of the image view and the cell should be. Good luck!

Going to the background

The user can take or pick a photo now but the app doesn't save it yet in the data store. Before you get to that, there are still a few improvements to make with the image picker.

Apple recommends that apps that show an alert view or action sheet remove these from the screen when the user presses the Home button to put the app in the background. The user may return to the app hours or days later and they'll have forgotten what they were going to do. The presence of the alert view or action sheet is confusing and users will be thinking, "What's that thing doing here?!"

To prevent this from happening, you'll make the Tag Location screen a little more attentive. When the app goes to the background, it will dismiss the action sheet if that is currently showing. You'll do the same for the image picker.

You've seen in the Checklists tutorial that the `AppDelegate` is notified by the operating system when the app is about to go into the background, through its `applicationDidEnterBackground` method. View controllers don't have such a method, but fortunately iOS sends out notifications through `NSNotificationCenter` that you can make the view controller listen to. Earlier you used the notification center to observe the notifications from Core Data. This time you'll listen for the `UIApplicationDidEnterBackgroundNotification`.

➤ In **LocationDetailsViewController.m**, expand `initWithCoder:` to:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    _descriptionText = @"";
    _categoryName = @"No Category";
    _date = [NSDate date];

    [[NSNotificationCenter defaultCenter] addObserver:self
      selector:@selector(applicationDidEnterBackground)
      name:UIApplicationDidEnterBackgroundNotification
      object:nil];
  }
  return self;
}
```

This adds `self`, i.e. the view controller, as the observer for `UIApplicationDidEnterBackgroundNotification`. When this notification is received, `NSNotificationCenter` will call the `applicationDidEnterBackground` method.

You should stop listening to these notifications when the Tag Location screen closes. Usually when you set up something in your init method, you tear it down again in the `dealloc` method.

➤ Add the `dealloc` method below `initWithCoder:`:

```
- (void)dealloc
{
  [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

➤ Add the `applicationDidEnterBackground` method to respond to the notification:

```
- (void)applicationDidEnterBackground
{
  if (_imagePicker != nil) {
    [self dismissViewControllerAnimated:NO completion:nil];
    _imagePicker = nil;
  }
```

```
  if (_actionSheet != nil) {
    [_actionSheet dismissWithClickedButtonIndex:
                  _actionSheet.cancelButtonIndex animated:NO];
    _actionSheet = nil;
  }

  [self.descriptionTextView resignFirstResponder];
}
```

If there is an active image picker or action sheet, you dismiss it. This assumes you store references to those objects in instance variables.

▸ Add the following instance variables:

```
UIActionSheet *_actionSheet;
UIImagePickerController *_imagePicker;
```

In `showPhotoMenu`, `takePhoto` and `choosePhotoFromLibrary`, you currently store the action sheet and image picker objects in local variables. You have to change these methods to use instance variables instead, so that `applicationDidEnterBackground` can access these objects.

▸ Remove the local variable declarations from these three methods:

```
- (void)takePhoto
{
  _imagePicker = [[UIImagePickerController alloc] init];
  _imagePicker.sourceType =
                        UIImagePickerControllerSourceTypeCamera;
  _imagePicker.delegate = self;
  _imagePicker.allowsEditing = YES;
  [self presentViewController:_imagePicker animated:YES
                   completion:nil];
}
```

```
- (void)choosePhotoFromLibrary
{
  _imagePicker = [[UIImagePickerController alloc] init];
  _imagePicker.sourceType =
                  UIImagePickerControllerSourceTypePhotoLibrary;
  _imagePicker.delegate = self;
  _imagePicker.allowsEditing = YES;
  [self presentViewController:_imagePicker animated:YES
                   completion:nil];
```

```
}
```

```
- (void)showPhotoMenu
{
  if ([UIImagePickerController isSourceTypeAvailable:
                    UIImagePickerControllerSourceTypeCamera]) {

    _actionSheet = [[UIActionSheet alloc]
      initWithTitle:nil
      delegate:self
      cancelButtonTitle:@"Cancel"
      destructiveButtonTitle:nil
      otherButtonTitles:@"Take Photo", @"Choose From Library",
      nil];

    [_actionSheet showInView:self.view];
  } else {
    [self choosePhotoFromLibrary];
  }
}
```

In the action sheet delegate you must set the `_actionSheet` variable to `nil` when the action sheet is dismissed. Otherwise you'll be keeping a reference to an action sheet that is no longer in use.

```
- (void)actionSheet:(UIActionSheet *)actionSheet
        didDismissWithButtonIndex:(NSInteger)buttonIndex
{
  . . .

  _actionSheet = nil;
}
```

The same thing goes for the image picker delegate methods:

```
- (void)imagePickerController:(UIImagePickerController *)picker
        didFinishPickingMediaWithInfo:(NSDictionary *)info
{
  . . .

  _imagePicker = nil;
}
```

```
- (void)imagePickerControllerDidCancel:
                                (UIImagePickerController *)picker
{
  . . .
  _imagePicker = nil;
}
```

That should do it.

› Try it out. Open the image picker (or the action sheet if you're on a device that has a camera) and tap the Home button to put the app to sleep. Then activate the app again and you should be back on the Tag Location screen.

You can find the project files for the app up to this point under **08 - Photo Picker** in the tutorial's Source Code folder.

Saving photos

The ability to pick photos is rather useless if the app doesn't also save them, so that's what you'll do here. It is possible to store images inside the Core Data store as "blobs" (Binary Large OBjects), but that is not recommended. Large blocks of data are better off stored as regular files in the app's Documents directory.

When the image picker gives you a `UIImage` object with a photo, that photo only lives in the iPhone's working memory. The photo may also be stored as a file somewhere if the user picked it from the photo library, but that's not the case if she just snapped a new picture. The user may also have resized the photo. So you have to save that `UIImage` to a file of your own if you want to keep it. The photos from this app will be saved in the JPEG format.

You also need a way to associate a `Location` object with that image file. The obvious solution is to store the filename in the `Location` object. You won't store the entire filename, just an ID, which is a positive number. The file itself will be named **Photo-XXX.jpg**, where XXX is the numeric ID.

› Open the Data Model editor. Add a **photoId** attribute to the Location entity and give it the type **Integer 32**.

› Add a property for this new attribute to **Location.h**, as well as a few new methods:

```
@property (nonatomic, retain) NSNumber * photoId;

+ (NSInteger)nextPhotoId;

- (BOOL)hasPhoto;
- (NSString *)photoPath;
- (UIImage *)photoImage;
```

Remember that in an object that is managed by Core Data, you have to declare the property as `@dynamic`.

➤ Add the `@dynamic` statement for the new property to **Location.m**:

```
@dynamic photoId;
```

The methods you're adding to the `Location` object are there to make working with the photo file a little easier.

The `hasPhoto` method determines whether this `Location` object has a photo associated with it or not. You set the `photoId` property to -1 if it doesn't, and to any positive integer if it does.

➤ Add the `hasPhoto` method:

```
- (BOOL)hasPhoto
{
  return (self.photoId != nil) &&
                      ([self.photoId integerValue] != -1);
}
```

The `photoPath` method returns the full path to the JPEG file for the photo. You'll save these files inside the app's Documents directory.

➤ Add the `documentsDirectory` and `photoPath` methods:

```
- (NSString *)documentsDirectory
{
  NSArray *paths = NSSearchPathForDirectoriesInDomains(
                    NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths lastObject];
  return documentsDirectory;
}

- (NSString *)photoPath
{
  NSString *filename = [NSString stringWithFormat:
                 @"Photo-%d.jpg", [self.photoId integerValue]];

  return [[self documentsDirectory]
                       stringByAppendingPathComponent:filename];
}
```

Note that `documentsDirectory` is only used inside this class, so it is not listed in the public `@interface` in the Location.h header file.

➤ Add `photoImage`. This method returns a `UIImage` object by loading the image file from the app's Documents directory. You'll need this later to show the photos for existing `Location` objects.

```
- (UIImage *)photoImage
{
  NSAssert(self.photoId != nil, @"No photo ID set");
  NSAssert([self.photoId integerValue] != -1,
                                          @"Photo ID is -1");

  return [UIImage imageWithContentsOfFile:[self photoPath]];
}
```

Notice the use of `NSAssert()`. An **assertion** is a check that makes sure that what you're doing is valid. It's a form of defensive programming. (Most of the crashes you've seen so far were actually caused by assertions in UIKit.)

If you were to ask a `Location` object for its `photoImage` without having given that `Location` a valid `photoId` earlier, the app will crash with the message "No photo ID set". That means there is a bug in the code somewhere because this is not supposed to happen. Internal consistency checks like this can be very useful.

Assertions are usually enabled only while you're developing and testing your app and disabled when you upload the final build of your app to the App Store. By then there should be no more bugs in your app (or so you would hope!). It's a good idea to use `NSAssert()` in strategic places to catch yourself making programming errors.

The final method is `nextPhotoId`. This is a class method – because it starts with a plus instead of a minus sign – meaning that you don't need to have a `Location` object to call it. You can call this method anytime from anywhere.

➤ Add the `nextPhotoId` method:

```
+ (NSInteger)nextPhotoId
{
  NSUserDefaults *defaults = [NSUserDefaults
                                        standardUserDefaults];
  NSInteger photoId = [defaults integerForKey:@"PhotoID"];
  [defaults setInteger:photoId+1 forKey:@"PhotoID"];
  [defaults synchronize];
  return photoId;
}
```

You need to have some way to generate a unique ID for each `Location` object. All `NSManagedObjects` have an `objectID` method, but that returns something unreadable such as:

```
<x-coredata://C26CC559-959C-49F6-BEF0-F221D6F3F04A/Location/p1>
```

You can't really use that in a filename. So instead, you're going to put a simple integer in `NSUserDefaults` and update it every time you ask for a new ID using `nextPhotoId`. This is similar to what you did in the last tutorial for making `ChecklistItem` IDs.

It may seem a little silly to use `NSUserDefaults` for this when you're already using Core Data as the data store, but with `NSUserDefaults` the `nextPhotoId` method is only five lines. You've seen how verbose the code is for fetching something from Core Data and then saving it again. This is just as easy. (As an exercise, you could try to implement these IDs using Core Data.)

That does it for `Location`. Now you have to save the photo in the Tag Location screen and fill in the `Location` object's `photoId` field. This happens in the Location Details View Controller's `done:` action.

➤ Change the `done:` method in **LocationDetailsViewController.m** to the following:

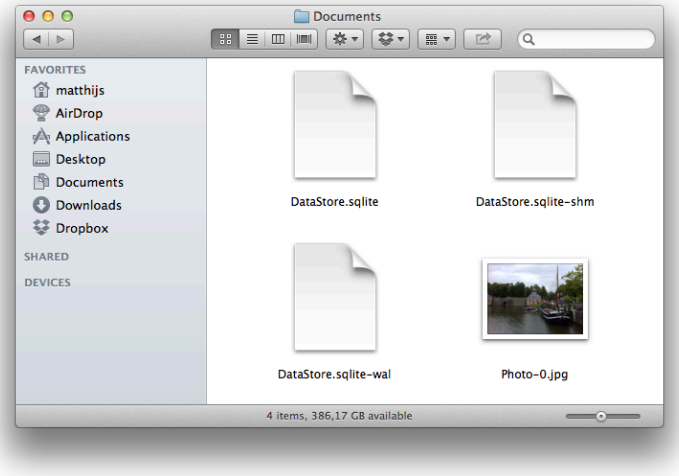
```
- (IBAction)done:(id)sender
{
  . . .
  location.placemark = self.placemark;

  if (_image != nil) {

    // 1
    if (![location hasPhoto]) {
      location.photoId = @([Location nextPhotoId]);
    }

    // 2
    NSData *data = UIImageJPEGRepresentation(_image, 0.5);
    NSError *error;
    if (![data writeToFile:[location photoPath]
                   options:NSDataWritingAtomic error:&error]) {
      NSLog(@"Error writing file: %@", error);
    }
  }

  NSError *error;
  if (![self.managedObjectContext save:&error]) {
    . . .
```

You've inserted new code in between where you set the properties of the `Location` object and where you save the managed object context. This code is only performed if `image` is not `nil`, in other words, when the user has picked a photo.

1. You need to get a new ID and assign it to the `Location`'s `photoId` property, but only if you're adding a photo to a `Location` that didn't already have one. If a photo existed, you keep the same ID and overwrite the existing JPEG file. The `@()` is necessary to convert the ID, which is an `NSInteger`, into an `NSNumber` object.
2. Here you write the `UIImage` object to a JPEG file. The `UIImageJPEGRepresentation()` function converts the `UIImage` into the JPEG format, and you save that to the path given by the `photoPath` method. (Also notice the use of the `NSError` pattern again.)

▸ Before you run the app, first remove the old **DataStore.sqlite** file (or simply remove the app from the Simulator or device). You have added a new attribute to the data model (photoId), so the data source is out of sync.

▸ Run the app, tag a location, choose a photo, and press Done to exit the screen. Now the photo you picked should be saved in the app's Documents directory as a regular PNG file.



The photo is saved in the app's Documents folder

▸ Tag another location and add a photo to it. Hmm... if you look into the Documents directory, this seems to have overwritten the previous photo.

Exercise. Try to debug this one on your own. What is going wrong here? □

Answer: The default value of integer variables is 0. That means each `Location` initially has a `photoId` of 0. That should really be -1, which means "no photo".

▸ In **LocationDetailsViewController.m**, change the top of the `done:` method to:

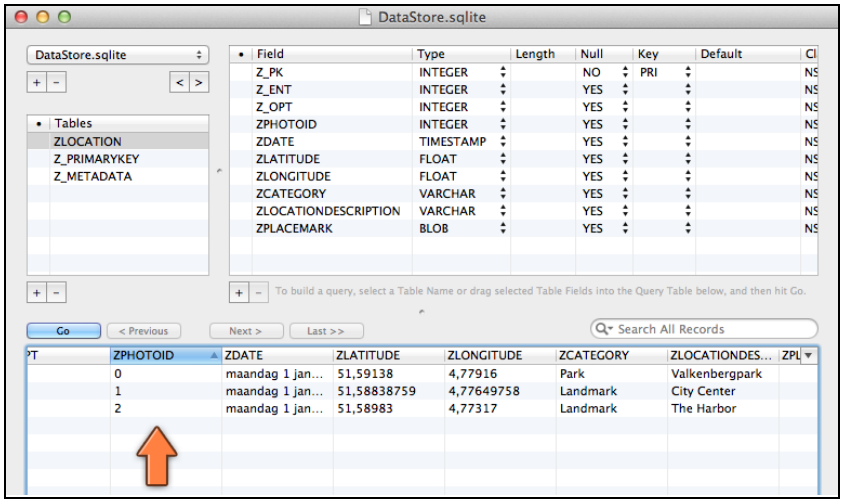
```
- (IBAction)done:(id)sender
{
  HudView *hudView = [HudView hudInView:
                      self.navigationController.view animated:YES];
```

```
  Location *location = nil;
  if (self.locationToEdit != nil) {
    hudView.text = @"Updated";
    location = self.locationToEdit;
  } else {
    hudView.text = @"Tagged";
    location = [NSEntityDescription
                insertNewObjectForEntityForName:@"Location"
                inManagedObjectContext:self.managedObjectContext];
    location.photoId = @-1;
  }
  . . .
```

You now give new `Location` objects a `photoId` of -1 so that the `hasPhoto` method correctly recognizes that these `Locations` do not have a photo yet.

▸ Run the app again and tag multiple locations with photos. Verify that now each photo is saved individually.

If you have Liya or another SQLite inspection tool, you can verify that each `Location` object has been given a unique `photoId` value (in the ZPHOTOID column):



The Location objects with unique photoId values in Liya

Editing photos

So far all the changes you've made were for the Tag Location screen and adding new locations. Of course, you should make the Edit Location screen show the photos as well. The change to `LocationDetailsViewController` is quite simple.

▸ Change `viewDidLoad` in **LocationDetailsViewController.m** to:

```
- (void)viewDidLoad
{
```

```
  [super viewDidLoad];

  if (self.locationToEdit != nil) {
    self.title = @"Edit Location";

    if ([self.locationToEdit hasPhoto]) {
      UIImage *existingImage = [self.locationToEdit photoImage];
      if (existingImage != nil) {
        [self showImage:existingImage];
      }
    }
  }
  . . .
```

If the `Location` that you're editing has a photo, then this calls `showImage:` to display it in the photo cell.

The call to `if (existingImage != nil)` is a bit of defensive programming. If `hasPhoto` is `YES`, then there should always be a valid image file present. But it's possible to imagine a scenario where there isn't, even though that should never happen. (The JPEG file could have been erased or corrupted.)

Note also what you **don't** do here: the `Location`'s image is not assigned to the `_image` instance variable. If the user doesn't change the photo, then you don't need to write it to a file again – it's already in that file and doing perfectly fine, thank you. If you were to put the photo in the `_image` variable, then the `done:` action would overwrite that existing file with the exact same data, which is a little silly. Therefore, the `_image` instance variable will only be set when the user picks a new photo.

➤ Run the app and take a peek at the existing locations from the Locations or Map tabs. The Edit Location screen should now show the photos for the locations you're editing. Verify that you can also change the photo and that the JPEG file in the app's Documents directory gets overwritten when you press the Done button.

There's another editing operation the user can perform on a location: deletion. What happens to the image file when the location is deleted? At the moment, nothing. That photo stays forever in the app's Documents directory. Let's do something about that and remove the file when the `Location` object is deleted.

Deleting locations happens in `LocationsViewController`'s `commitEditingStyle` method.

➤ Change `commitEditingStyle` in **LocationsViewController.m** to the following:

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
```

```
{
  if (editingStyle == UITableViewCellEditingStyleDelete) {
    Location *location = [self.fetchedResultsController
                                    objectAtIndexPath:indexPath];
    [location removePhotoFile];
    [self.managedObjectContext deleteObject:location];
    . . .
```

The new line calls `removePhotoFile` on the `Location` object. Let's add that method to the `Location` class.

➤ Add the method signature to **Location.h**:

```
- (void)removePhotoFile;
```

➤ And the method body to **Location.m**:

```
- (void)removePhotoFile
{
  NSString *path = [self photoPath];
  NSFileManager *fileManager = [NSFileManager defaultManager];
  if ([fileManager fileExistsAtPath:path]) {
    NSError *error;
    if (![fileManager removeItemAtPath:path error:&error]) {
      NSLog(@"Error removing file: %@", error);
    }
  }
}
```

This is a code snippet that you can use to remove any file or folder. The `NSFileManager` class has all kinds of useful methods for dealing with the file system.

➤ Try it out. Add a new location and give it a photo. You should see the JPEG file in the Documents directory. Then delete it from the Locations screen and look in the Documents directory to make sure the JPEG file is truly gone.

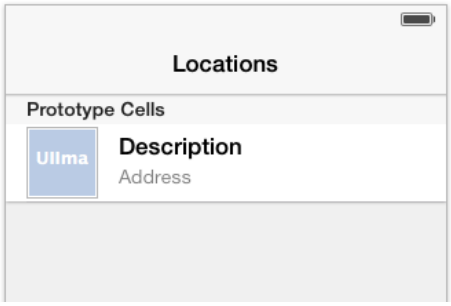
Thumbnails of the photos

Now that locations can have photos, you might as well show thumbnails for these photos in the Locations tab. That will liven up this screen a little... a table view with just a bunch of text isn't particularly exciting.

➤ Go to the storyboard editor. In the prototype cell on the Locations View Controller, move the two labels to X = 82. Make them 238 points wide.

➤ Drag a new **Image View** into the cell. Place it at the top-left corner of the cell. Give it the following position: X = 15, Y = 2. Make it 52 by 52 points big.

➤ Connect the image view to a new outlet on `LocationCell`, named **photoImageView**. (Tip: you should connect the image view to the cell, not to the view controller.)



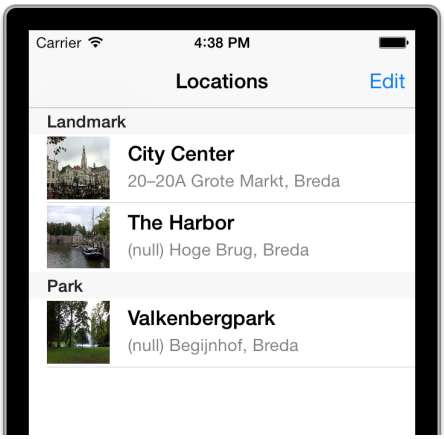
The table view cell has an image view

Now you can put any image into the table view cell simply by placing it inside the image view from `LocationCell`'s `photoImageView` property.

➤ Add the following code to the bottom of `configureCell:atIndexPath:` in **LocationsViewController.m**:

```
- (void)configureCell:(UITableViewCell *)cell
          atIndexPath:(NSIndexPath *)indexPath
{
  . . .

  UIImage *image = nil;
  if ([location hasPhoto]) {
    image = [location photoImage];
  }
  locationCell.photoImageView.image = image;
}
```

➤ Try it out. The Locations tab should now look something like this:



Images in the Locations table view

The images look a little squashed again – because you didn't set the Aspect Fit content mode on the image view – but there's a bigger problem here. These photos are potentially huge (2592 by 1936 pixels or more), even though the image view is only 52 pixels square. The image view needs to scale down the images by a lot (which is also why they look a little "gritty").

What if you have tens or even hundreds of locations? That is going to require a ton of memory and processing speed just to display these tiny thumbnails. A better solution is to scale down the images before you put them into the table view cell.

You're going to use something new for that: an Objective-C **category**.

Categories

Not to be confused with the `category` property on the `Location` object, a **category** in Objective-C is a way to extend a class without having to subclass it. Using a category you can add methods to existing classes, even if you didn't write those classes yourself. That includes classes from iOS, such as `NSString` and `UIImage`.

If you ever catch yourself thinking, "Gee, I wish class X had such-or-so method" then you can probably give that class that method by making a category for it. For example, suppose you want `NSString` to have a method for adding random words to the string, you could add the `addRandomWord` method to `NSString` as follows:

You create a new .h and .m file. The convention for category file names is **NSString+RandomWord.h** and **NSString+RandomWord.m**. It's the name of the class you're extending, followed by a plus sign and the name of your extension (RandomWord).

The .h file would look like this:

```
@interface NSString (RandomWord)
- (NSString *)addRandomWord;
@end
```

By adding `(RandomWord)` to the `@interface` line, you're creating a new category on the `NSString` class.

The .m file would look like this:

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

@implementation NSString (RandomWord)

- (NSString *)addRandomWord
{
  int value = arc4random_uniform(3);

  NSString *word;
  if (value == 0) {
    word = @"rabbit";
  } else if (value == 1) {
    word = @"banana";
  } else if (value == 2) {
    word = @"boat";
  }

  return [NSString stringWithFormat:"%@%@", self, word];
}

@end
```

As you can see, creating a new category is very much like creating a new class. Anywhere in your code where you import the .h file for this category you can now call `addRandomWord` on any `NSString` object:

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

NSString *someString = @"Hello, ";
NSLog(@"The queen says: %@", [someString addRandomWord]);
```

Categories are pretty cool because they make it simple to add new functionality into an existing class. In other programming languages you would have to make a subclass and put your new methods in there, but categories are often a cleaner solution.

A category can only be used to add methods, not instance variables. There is a special form of a category that you've seen before: the class extension. This is a category with no name, just a pair of parentheses, but that gives it special powers. In a class extension you can add new properties that are backed by instance variables (something you cannot do in a regular category), but you

can't use them to extend classes that you didn't write and don't have the source code for.

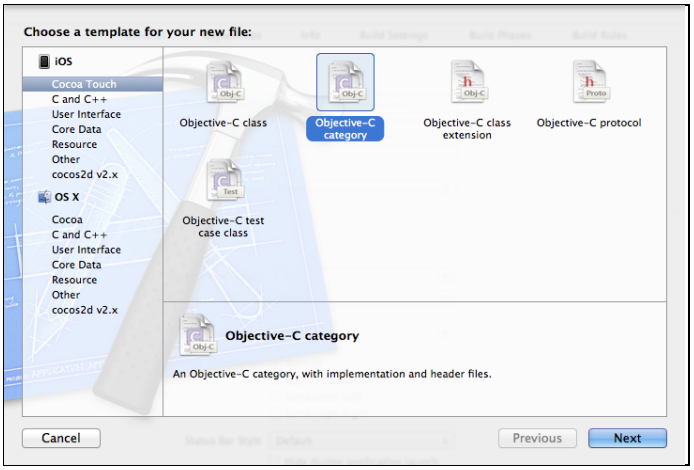
You are going to add a category to `UIImage` that lets you resize the image. You'll use it as follows:

```
image = [image resizedImageWithBounds:CGSizeMake(52, 52)];
```

The `resizedImageWithBounds:` method is new. The "bounds" is the size of the rectangle (or square in this case) that encloses the image. If the image itself is not square, then the resized image may actually be smaller than the bounds.

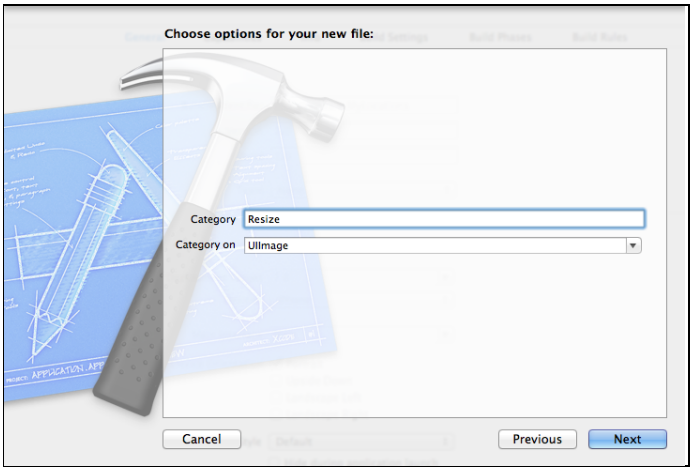
Let's write the category.

› Add a new file to the project and choose the **Objective-C category** template:



Choosing the Objective-C category template

› Name the category **Resize**. It is made on **UIImage**:



Options for the category template

Xcode will add two files to the project, **UIImage+Resize.h** and **.m**.

➤ Add the following method signature to **UIImage+Resize.h**:

```
@interface UIImage (Resize)

- (UIImage *)resizedImageWithBounds:(CGSize)bounds;

@end
```

➤ Add the method body in **UIImage+Resize.m**:

```
- (UIImage *)resizedImageWithBounds:(CGSize)bounds
{
  CGFloat horizontalRatio = bounds.width / self.size.width;
  CGFloat verticalRatio = bounds.height / self.size.height;
  CGFloat ratio = MIN(horizontalRatio, verticalRatio);
  CGSize newSize = CGSizeMake(self.size.width * ratio,
                              self.size.height * ratio);

  UIGraphicsBeginImageContextWithOptions(newSize, YES, 0);
  [self drawInRect:CGRectMake(0, 0, newSize.width,
                                              newSize.height)];
  UIImage *newImage =
                    UIGraphicsGetImageFromCurrentImageContext();
  UIGraphicsEndImageContext();

  return newImage;
}
```

This method first calculates how big the image can be in order to fit inside the bounds rectangle. It uses the "aspect fit" approach to keep the aspect ratio intact. Then it creates a new image context and draws the image into that. We haven't really dealt with graphics contexts before, but they are an important concept in Core Graphics (it has nothing to do with the managed object context from Core Data).

To put this category in action, you just need to add an import for the proper header to **LocationsViewController.m**, or it won't be able the find the `resizedImageWithBounds:` method.

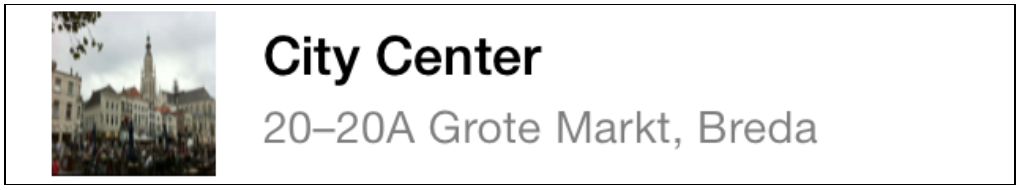
➤ Add the import to **LocationsViewController.m**:

```
#import "UIImage+Resize.h"
```

➤ And add this code to `configureCell:atIndexPath:`

```
- (void)configureCell:(UITableViewCell *)cell
          atIndexPath:(NSIndexPath *)indexPath
{
  . . .

  UIImage *image = nil;
  if ([location hasPhoto]) {
    image = [location photoImage];
    if (image != nil) {
      image = [image resizedImageWithBounds:CGSizeMake(52, 52)];
    }
  }
  locationCell.photoImageView.image = image;
}
```

➤ Run the app. The thumbnails look like this:

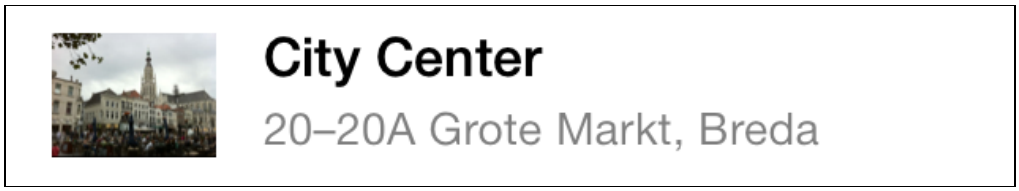


The photos are shrunk to the size of the thumbnails

The images are a little blurry, and they still seem to be stretched out. This is because the content mode on the image view is still wrong. Previously it shrunk the big photos to 52 by 52 points, but now the thumbnails may actually be smaller than 52 points (unless the photo was perfectly square) and they get scaled up to fill the entire image view rectangle.

➤ Go to the storyboard and set the **Mode** of the image view to **Center**.

➤ Run the app again and now the photos look A-OK:



The thumbnails now have the correct aspect ratio

Exercise. Change the resize function in the `UIImage` category to resize using the "Aspect Fill" rules instead of the "Aspect Fit" rules. Both keep the aspect ratio intact but Aspect Fit keeps the entire image visible while Aspect Fill fills up the entire rectangle and may cut off parts of the sides. In other words, Aspect Fit scales to the longest side but Aspect Fill scales to the shortest side.

Aspect Fit vs. Aspect Fill

Handling low-memory situations

The `UIImagePickerController` is very memory-hungry. Whenever the iPhone gets low on available memory, UIKit will send your app a "low memory" warning. When that happens you should reclaim as much memory as possible, or iOS might be forced to terminate your app. And that's something to avoid – users generally don't like apps that suddenly quit on them!

Chances are that your app gets one or more low-memory warnings while the image picker is open, especially when you run it on a device that has other apps suspended in the background. Photos take up a lot of space – especially when your camera is 5 or more megapixels – so it's no wonder that memory fills up quickly.

You can handle low-memory warnings by overriding the `didReceiveMemoryWarning` method in your view controllers to free up any memory you no longer need. An example of that are things that you can easily recalculate later, such as thumbnails or other cached objects.

UIKit is already pretty smart about low memory situations and it will do everything it can to release memory, including the thumbnail images of rows that are not (or no longer) visible in your table view. So for MyLocations there's not much that you can or need to do to free up additional memory; you can rely on UIKit to automatically take care of it. But in your own apps you might want to take extra measures, depending on the sort of cached data that you have.

By the way, on the Simulator you can trigger a low memory warning using the Hardware → Simulate Memory Warning menu item. It's smart to test your apps under low memory conditions, because that's what they are going to encounter out in the wild once they're running on real users' devices.

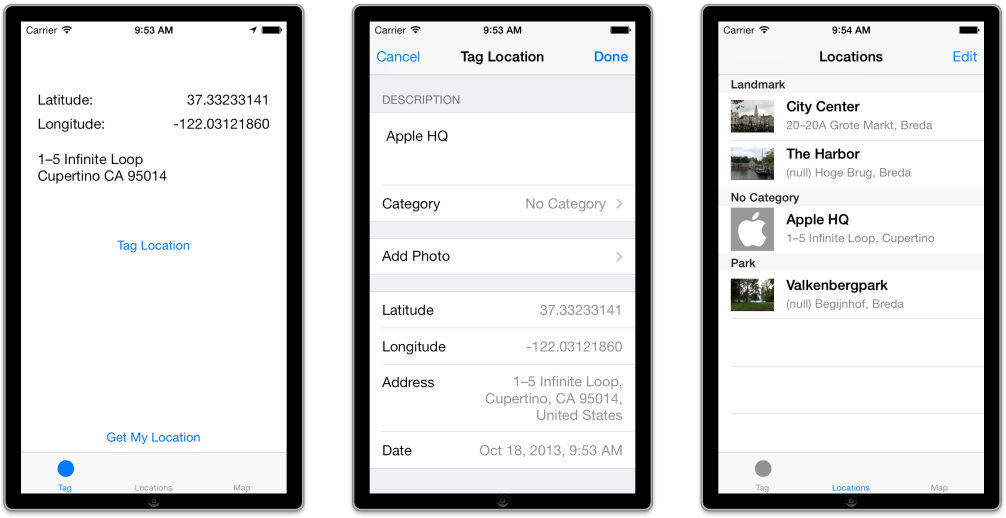
Great, that concludes all the functionality for this app. Now it's time to fine-tune its looks.

You can find the project files for the app up to this point under **09 - Saving Photos** in the tutorial's Source Code folder.

Making it look good

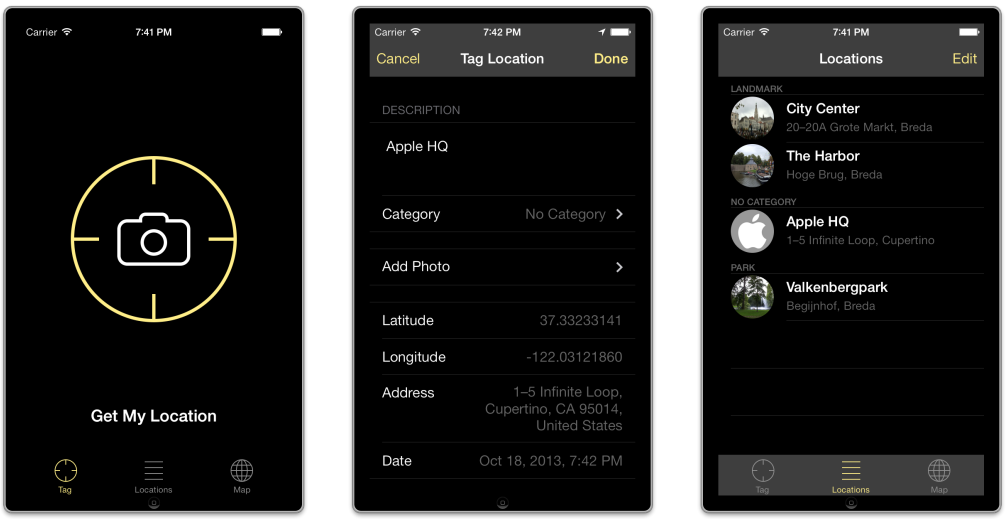
Apps with appealing visuals sell better than ugly ones. Usually I don't wait with the special sauce until the end of a project, but for these tutorials it's clearer if you first get all the functionality in before you improve the looks. Now that the app works as it should, let's make it look good!

You're going to go from this,



Before

to this:



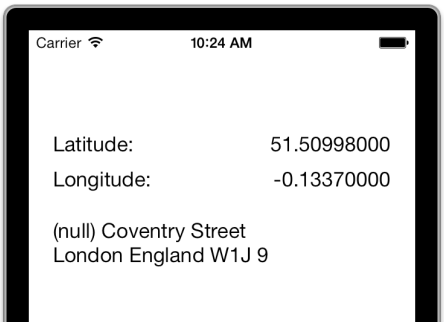
After

The main screen gets the biggest makeover, but you'll also tweak the others a little.

No more (null) in placemarks

You may have noticed that the reverse geocoded street address often contains the text "(null)". That means one or more of its fields do not have a value. Whenever you convert a `nil` object into a string, it says "(null)". That's not very user-friendly, so you'll have to make the logic for displaying placemarks a little smarter.

If you haven't noticed any "(null)" before, then launch the app in the Simulator and set the simulated location to London, England. Let it run for a bit and you'll see missing house numbers and street names:



No street name was found for this location

> **Note:** For some reason, UK postal codes are truncated to 5 characters on iOS 7. The above address should really be London England W1J 9HP. This appears to be a bug in Core Location.

There are three places where you convert `CLPlacemark` objects into strings:

- `CurrentLocationViewController`, the main screen
- `LocationDetailsViewController`, the Tag/Edit Location screen
- `LocationsViewController`, the list of saved locations

Let's start with the main screen. `CurrentLocationViewController` has a method named `stringFromPlacemark` where this conversion happens. It currently does this:

```
return [NSString stringWithFormat:@"%@ %@\n%@ %@ %@",
    thePlacemark.subThoroughfare, thePlacemark.thoroughfare,
    thePlacemark.locality, thePlacemark.administrativeArea,
    thePlacemark.postalCode];
```

It may be a little hard to read with all those `@` symbols, but this returns a string that looks like:

```
subThoroughfare thoroughfare
locality administrativeArea postalCode
```

This string goes into a `UILabel` that has room for two lines, so you use the `\n` character sequence to create a line-break between the thoroughfare and locality.

The problem is that any of these fields may be `nil`. Currently they print `(null)` when that happens but now you'll make the code smarter so that it skips these empty fields. You'll use the `NSMutableString` class to build the final string. Unlike `NSString`, which cannot be changed once it's created, `NSMutableString` is a string object that you can add text to.

➤ Change `stringFromPlacemark:` to the following:

```
- (NSString *)stringFromPlacemark:(CLPlacemark *)thePlacemark
{
  // 1
  NSMutableString *line1 = [NSMutableString
                                      stringWithCapacity:100];

  // 2
  if (thePlacemark.subThoroughfare != nil) {
    [line1 appendString:thePlacemark.subThoroughfare];
  }

  // 3
  if (thePlacemark.thoroughfare != nil) {
    if ([line1 length] > 0) {
      [line1 appendString:@" "];
    }
```

```
    [line1 appendString:thePlacemark.thoroughfare];
  }

  // 4
  NSMutableString *line2 = [NSMutableString
                                        stringWithCapacity:100];

  if (thePlacemark.locality != nil) {
    [line2 appendString:thePlacemark.locality];
  }

  if (thePlacemark.administrativeArea != nil) {
    if ([line2 length] > 0) {
      [line2 appendString:@" "];
    }
    [line2 appendString:thePlacemark.administrativeArea];
  }

  if (thePlacemark.postalCode != nil) {
    if ([line2 length] > 0) {
      [line2 appendString:@" "];
    }
    [line2 appendString:thePlacemark.postalCode];
  }

  // 5
  [line1 appendString:@"\n"];
  [line1 appendString:line2];
  return line1;
}
```

Let's look at this in detail:

1. Create a mutable string object with room for 100 characters, initially. The string will expand to make more room if necessary.
2. `NSMutableString` is a subclass of `NSString`, so all the regular `NSString` methods can be used here too, but `appendString` is new (there is also an `appendFormat`). If the placemark has a `subThoroughfare`, you add it to the string.
3. Adding the `thoroughfare` is done similarly, but you also put a space between it and `subThoroughfare` so they don't get glued together. If there was no `subThoroughfare` in the placemark, then you don't want to add that space.
4. The same logic goes for the second line. This adds the locality, administrative area, and postal code, with spaces between them where appropriate.
5. Finally, the two lines are concatenated (added together) with a newline character in between.

This works – try it out, there should no longer be any "(null)" in the address label – but there's a lot of repetition going on in this method. You can refactor this.

Exercise. Try to make this method simpler by moving the common logic into a new method.

Answer: Here is how I did it.

➤ Add the following method below `stringFromPlacemark`:

```
- (void)addText:(NSString *)text toLine:(NSMutableString *)line
        withSeparator:(NSString *)separator
{
  if (text != nil) {
    if ([line length] > 0) {
      [line appendString:separator];
    }
    [line appendString:text];
  }
}
```

This adds `text` (a regular string) to a mutable string, with an optional separator such as a space or comma that is only used if the mutable string isn't empty.

➤ Now you can rewrite `stringFromPlacemark:` as follows:

```
- (NSString *)stringFromPlacemark:(CLPlacemark *)thePlacemark
{
  NSMutableString *line1 = [NSMutableString
                                        stringWithCapacity:100];
  [self addText:thePlacemark.subThoroughfare toLine:line1
        withSeparator:@""];
  [self addText:thePlacemark.thoroughfare toLine:line1
        withSeparator:@" "];

  NSMutableString *line2 = [NSMutableString
                                        stringWithCapacity:100];
  [self addText:thePlacemark.locality toLine:line2
        withSeparator:@""];
  [self addText:thePlacemark.administrativeArea toLine:line2
        withSeparator:@" "];
  [self addText:thePlacemark.postalCode toLine:line2
        withSeparator:@" "];

  if ([line1 length] == 0) {
    [line2 appendString:@"\n "];
    return line2;
  } else {
```

```
    [line1 appendString:@"\n"];
    [line1 appendString:line2];
    return line1;
  }
}
```

That is a lot cleaner.

But what's up with the if-statement at the bottom? `UILabel` always centers its text vertically. You made the label big enough to fit two lines of text, but if there's only one line's worth then that text will be positioned in the middle of the label, not at the top. For some apps that might be fine, but here I want to always align the text at the top.

If there is no text in the `line1` string, then you add a newline and a space to the end of `line2`. This will force the `UILabel` to always draw two lines of text, even if the second one looks empty (it only has a space).

That settles `CurrentLocationViewController`. What about the other two controllers? Well, they probably need to do something very similar: make an `NSMutableString` and add text to it. You could either copy the `addText:toLine:withSeparator:` method into the other two view controllers, which causes code duplication, or... you can turn it into a category on `NSMutableString`. I bet you can guess which one we're going to pick.

➤ Add a new file to the project using the **Objective-C category** template. Choose **AddText** for the category name and apply it to `NSMutableString`.

➤ Replace the contents of **NSMutableString+AddText.h** with:

```
@interface NSMutableString (AddText)

- (void)addText:(NSString *)text withSeparator:
                                          (NSString *)separator;

@end
```

➤ Replace the contents of **NSMutableString+AddText.m** with:

```
#import "NSMutableString+AddText.h"

@implementation NSMutableString (AddText)

- (void)addText:(NSString *)text withSeparator:
                                           (NSString *)separator
{
  if (text != nil) {
    if ([self length] > 0) {
```

```
      [self appendString:separator];
    }
    [self appendString:text];
  }
}

@end
```

You've simply moved that `addText` method into this new category. The only difference is that the `toLine:` parameter is no longer necessary because this method now always works on the string object that it belongs to.

➤ Add an import for this category to **CurrentLocationViewController.m**:

```
#import "NSMutableString+AddText.h"
```

➤ And change the `stringFromPlacemark:` method to:

```
- (NSString *)stringFromPlacemark:(CLPlacemark *)thePlacemark
{
  NSMutableString *line1 = [NSMutableString
                                        stringWithCapacity:100];
  [line1 addText:thePlacemark.subThoroughfare
         withSeparator:@""];
  [line1 addText:thePlacemark.thoroughfare withSeparator:@" "];

  NSMutableString *line2 = [NSMutableString
                                        stringWithCapacity:100];
  [line2 addText:thePlacemark.locality withSeparator:@""];
  [line2 addText:thePlacemark.administrativeArea
         withSeparator:@" "];
  [line2 addText:thePlacemark.postalCode withSeparator:@" "];

  if ([line1 length] == 0) {
    [line2 appendString:@"\n "];
    return line2;
  } else {
    [line1 appendString:@"\n"];
    [line1 appendString:line2];
    return line1;
  }
}
```

➤ Remove the `addText:toLine:withSeparator:` method; it's no longer used for anything.

Now you have a pretty clean solution that you can re-use in the other two view controllers.

➤ Add the import to **LocationDetailsViewController.m**:

```
#import "NSMutableString+AddText.h"
```

➤ Change its `stringFromPlacemark:` method to:

```
- (NSString *)stringFromPlacemark:(CLPlacemark *)placemark
{
  NSMutableString *line = [NSMutableString
                                        stringWithCapacity:100];

  [line addText:placemark.subThoroughfare withSeparator:@""];
  [line addText:placemark.thoroughfare withSeparator:@" "];
  [line addText:placemark.locality withSeparator:@", "];
  [line addText:placemark.administrativeArea withSeparator:
                                                    @", "];
  [line addText:placemark.postalCode withSeparator:@" "];
  [line addText:placemark.country withSeparator:@", "];

  return line;
}
```

It's slightly different from how the main screen does it. Here you make a string that looks like: `@"subThoroughfare thoroughfare, locality, administrativeArea postalCode, country"`. There are no `\n` newline characters and some of the elements are separated by commas instead of just spaces. Newlines aren't necessary here because the label will word-wrap.

The final place to change is `LocationsViewController`.

➤ Add the import to **LocationsViewController.m**:

```
#import "NSMutableString+AddText.h"
```

This class doesn't have a `stringFromPlacemark:` method. You only show the street and the city so the conversion is simpler. The string will be `@"subThoroughfare thoroughfare, locality"`.

➤ Change the relevant part of `configureCell` method to:

```
- (void)configureCell:(UITableViewCell *)cell
        atIndexPath:(NSIndexPath *)indexPath
{
  . . .
```

```
  if (location.placemark != nil) {

    NSMutableString *string = [NSMutableString
                                        stringWithCapacity:100];

    [string addText:location.placemark.subThoroughfare
            withSeparator:@""];
    [string addText:location.placemark.thoroughfare
            withSeparator:@" "];
    [string addText:location.placemark.locality
            withSeparator:@", "];

    locationCell.addressLabel.text = string;

  } else {
    . . .
```

If there is now a bit of missing data, such as the `subThoroughfare` which often is `nil`, then the app simply won't display it. That's a lot nicer than showing the text "(null)" to the user.

And that's it for placemarks.

Back to black

Right now the app looks like a typical standard iOS 7 app: lots of white, gray tab bar, blue tint color. Let's go for a radically different look and paint the whole thing black.

➤ Open the storyboard and go to the Current Location View Controller. Select the top-level view and change its **Background Color** to **Black Color**.

➤ Select all the labels (probably easiest from the outline pane since they are now invisible) and set their **Color** to **White Color**.

➤ Change the **Font** of the **(Latitude/Longitude goes here)** labels to **System Bold 17**.

➤ Select the two buttons and change their **Font** to **System Bold 20**, to make them slightly larger. You may need to resize their frames to make the text fit (remember, **⌘=** is the magic keyboard shortcut).

➤ In the **File inspector**, change **Global Tint** to the color **Red: 255, Green: 238, Blue: 136**. That makes the buttons and other interactive elements yellow, which stands out nicely against the black background.

➤ Select the Get My Location button and change its **Text Color** to **White Color**. This provides some contrast between the two buttons.

The storyboard should look like this:

The new yellow-on-black design

When you run the app, there are two obvious problems: the status bar text has become invisible (it is black text on a black background) and the grey tab bar sticks out like a sore thumb (also, the yellow tint color doesn't work very well on light grey).

To fix this, you can use the `UIAppearance` API. This is a set of methods that lets you customize the look of the standard UIKit controls. You can do this on a per-control basis, or you can use the "appearance proxy" to change the look of all of the controls of a particular type at once. That's what you're going to do here.

▸ Add the following method to **AppDelegate.m**:

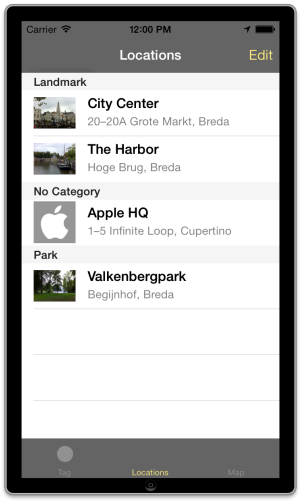
```
- (void)customizeAppearance
{
  [[UINavigationBar appearance] setBarTintColor:
                                        [UIColor blackColor]];

  [[UINavigationBar appearance] setTitleTextAttributes:@{
    NSForegroundColorAttributeName : [UIColor whiteColor],
    }];

  [[UITabBar appearance] setBarTintColor:[UIColor blackColor]];
}
```

This changes the "bar tint" or background color of all navigation bars and tab bars in the app to black in one fell swoop. It also sets the color of the navigation bar's title label to white.

▸ Call this method from `didFinishLaunchingWithOptions:`

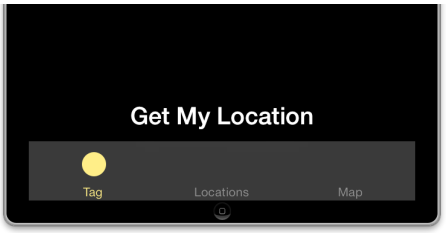
```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  [self customizeAppearance];
  . . .
```

This looks a bit better already. On the Locations and Map screens you can clearly see that the bars now have a dark tint:



The navigation and tab bars appear in a dark color

Keep in mind that the bar tint is not the true background color. The bars are still translucent, which is why they appear as a medium gray rather than pure black. That's a bit of a problem on the main screen, where the dark gray tab bar still stands out too much:



It would be nice to have a fully black tab bar here

It is possible to make the tab bar on the main screen completely black, while it remains translucent on the other screens. Warning: This is a bit of a hack (but it's little things such as these that add spice to life).

› Change the class extension interface in **CurrentLocationViewController.m** to:

```
@interface CurrentLocationViewController ()
                                  <UITabBarControllerDelegate>
```

➤ And add the following method:

```
#pragma mark - UITabBarControllerDelegate

- (BOOL)tabBarController:(UITabBarController *)tabBarController
  shouldSelectViewController:(UIViewController *)viewController
{
  tabBarController.tabBar.translucent =
                                  (viewController != self);
  return YES;
}
```

This method gets called when the user switches tabs. It sets the tab bar's `translucent` property to YES if the newly selected view controller is not `self`, in other words if the Locations or Map screen becomes active. However, if the Current Location screen is the active tab, `translucent` becomes NO, making the tab bar pitch black.

➤ You still need to tell the tab bar that the view controller is its delegate, so add the following lines to `viewDidLoad`:

```
self.tabBarController.delegate = self;
self.tabBarController.tabBar.translucent = NO;
```

➤ Run the app. Now the tab bar is completely black on the first screen and translucent on the others:

The tab bar is totally black

Tab bar icons

The icons in the tab bar could also do with some improvement. The Xcode Tabbed Application template put a bunch of cruft in the app that you're no longer using, so let's get rid of it.

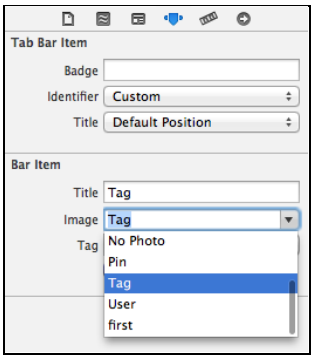
➤ Remove the **SecondViewController.h** and **.m** files from the project.

➤ Remove the **first** and **second** images from the asset catalog.

Tab bar images should be basic grayscale images of up to 30 × 30 pixels for low-resolution devices and 60 × 60 pixels for Retina. You don't have to tint the images; iOS will automatically draw them in the proper color.

➤ The resources for this tutorial include an **Images** directory. Add the files from this folder to the asset catalog. (Most of these images are in white, which makes them hard to see on the white background of the asset catalog, but trust me they are there.)

➤ Go to the storyboard. Select the **Tab Bar Item** of the Current Location screen. In the **Attributes inspector**, under **Image** choose **Tag**.

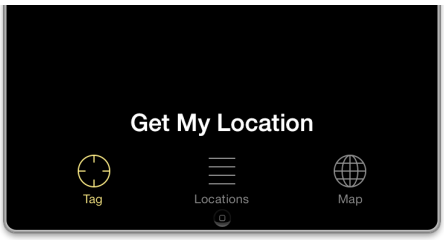


Choosing an image for a Tab Bar Item

➤ For the Tab Bar Item from the navigation controller attached to the Locations screen, choose the **Locations** image.

➤ For the Tab Bar Item from the Map View Controller, choose the **Map** image.

Now the tab bar looks a lot more appealing:



The tab bar with proper icons

The status bar

The status bar is currently invisible on the Tag screen and appears as black text on dark gray on the other two screens. It will look better if the status bar text is white instead.

To do this, you need to override the `preferredStatusBarStyle` method in your view controllers and make it return the value `UIStatusBarStyleLightContent`. For some reason that won't work for view controllers embedded in a Navigation Controller, such as the Locations tab and the Tag/Edit Location screens.

The simplest way to make the status bar white for all your view controllers in the entire app is to replace the `UITabBarController` with your own subclass.

➤ Add a new file to the project using the **Objective-C class** template. Name it **MyTabBarController**, subclass of `UITabBarController`.

➤ Replace the contents of **MyTabBarController.m** with:

```
#import "MyTabBarController.h"

@implementation MyTabBarController

- (UIStatusBarStyle)preferredStatusBarStyle
{
  return UIStatusBarStyleLightContent;
}

- (UIViewController *)childViewControllerForStatusBarStyle
{
  return nil;
}

@end
```

By returning `nil` from `childViewControllerForStatusBarStyle`, the tab bar controller will look at its own `preferredStatusBarStyle` method.

➤ In the storyboard, select the Tab Bar Controller and in the **Identity inspector** change its **Class** to **MyTabBarController**. This tells the storyboard that it should now create an instance of your subclass when the app starts up.

You also need to make a subclass for the Navigation Controller that embeds the Tag/Edit Location screen, because that is presented modally on top of the other screens and is therefore not part of the Tab Bar Controller hierarchy.

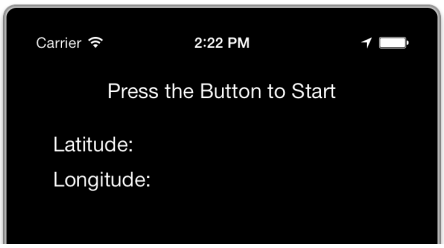
➤ Add a new file to the project using the **Objective-C class** template. Name it **MyNavigationController**, subclass of `UINavigationController`.

➤ Replace the contents of **MyNavigationController.m** with:

```
#import "MyNavigationController.h"

@implementation MyNavigationController

- (UIStatusBarStyle)preferredStatusBarStyle
{
  return UIStatusBarStyleLightContent;
}

@end
```

› In the storyboard, select the Navigation Controller that is connected to the Location Details View Controller. Change its **Class** to **MyNavigationController**.

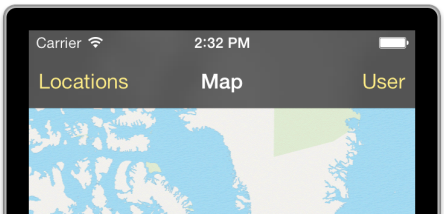
Now the status bar is white everywhere:



The status bar is visible again

The map screen

The Map screen currently has a somewhat busy navigation bar with three pieces of text in it: the title and the two buttons.



The bar button items have text labels

The design advice that Apple gives for iOS 7 apps is to prefer text to icons on buttons because icons tend to be harder to understand. The disadvantage of using text is that it makes your navigation bar more crowded. There are two possible solutions:

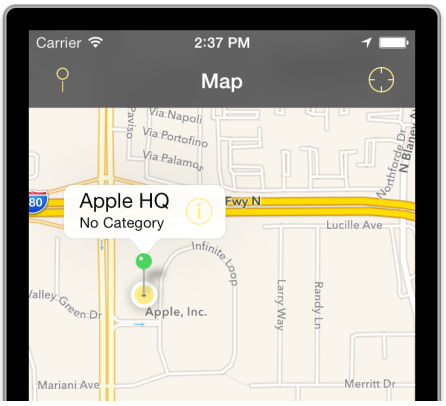
1. Remove the title. If the purpose of the screen is obvious, which it is in this case, then the title "Map" is superfluous. You might as well remove it.
2. Keep the title but replace the button labels with icons.

For this app you'll choose the second option.

› Select the **Locations** bar button item in the storyboard. In the **Attributes inspector**, under **Image** choose **Pin**. This will remove the text from the button.

› For the User bar button item, choose the **User** image.

The Map screen now looks like this:



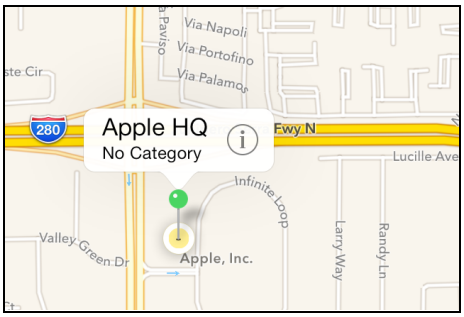
The Map screen with the button icons

Notice that the dot for the user's current location is drawn in the yellow tint color (it was a blue dot before). The **(i)** button on the map annotation also appears in yellow, making it hard to see on the white callout. Fortunately, you can override the tint color on a per-view basis.

› In **MapViewController.m**, in the method `mapView:viewForAnnotation:`, add this below the line that sets `annotationView.pinColor`:

```
annotationView.tintColor = [UIColor colorWithWhite:0.0f
                                             alpha:0.5f];
```

This sets the annotation's tint color to half-opaque black:



The callout button is now easier to see

Fixing up the table views

The app is starting to shape up but there are still some details to take care of. The table views, for example, are still very white. Unfortunately, what `UIAppearance` can do for table views is very limited, so you'll have to customize each of the table views individually.

› Add the following lines to `viewDidLoad` in **LocationsViewController.m**:

```
self.tableView.backgroundColor = [UIColor blackColor];
```

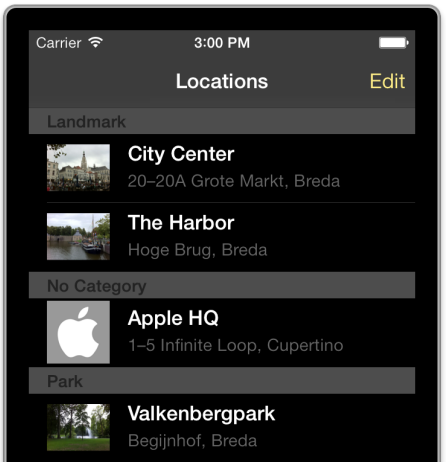
```
self.tableView.separatorColor = [UIColor colorWithWhite:1.0f
                                                  alpha:0.2f];
```

This makes the table view itself black but does not alter the cells.

➤ Add these lines to the bottom of `configureCell:atIndexPath:` to change the appearance of the actual cells:

```
locationCell.backgroundColor = [UIColor blackColor];
locationCell.descriptionLabel.textColor = [UIColor whiteColor];
locationCell.descriptionLabel.highlightedTextColor =
                        locationCell.descriptionLabel.textColor;
locationCell.addressLabel.textColor =
                       [UIColor colorWithWhite:1.0f alpha:0.4f];
locationCell.addressLabel.highlightedTextColor =
                            locationCell.addressLabel.textColor;
```

➤ Run the app. That's starting to look pretty good already:



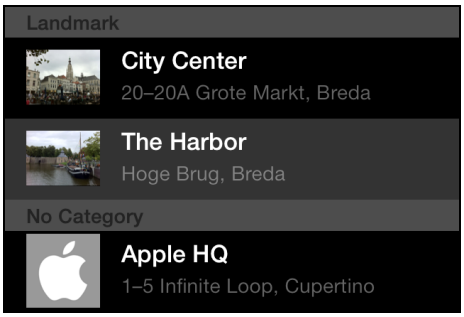
The table view cells are now white-on-black

There is a small issue. When you tap a cell it still lights up in a bright color, which is a little extreme. It would look better if the selection color was more subdued. There is no "selectionColor" property on `UITableViewCell`, but you can give it a different view to display when it is selected.

➤ Add the following to the bottom of `configureCell:atIndexPath:`

```
UIView *selectionView = [[UIView alloc]
                                    initWithFrame:CGRectZero];
selectionView.backgroundColor =
                       [UIColor colorWithWhite:1.0f alpha:0.2f];
locationCell.selectedBackgroundView = selectionView;
```

This creates a new `UIView` that is filled a dark gray color. This new view is placed on top of the cell's background when the user taps on the cell. It looks like this:



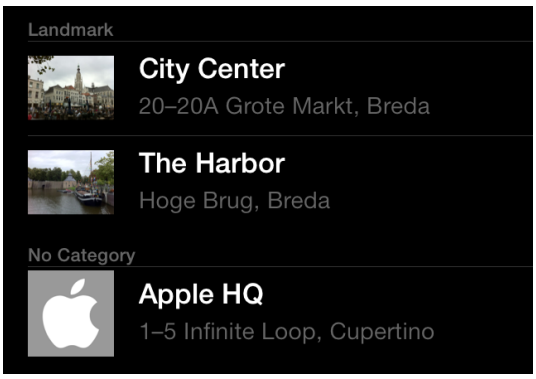
The selected cell (The Harbor) has a subtly different background color

I also think the section headers are on the heavy side. There is no easy way to customize the existing headers but you can replace them with a view of your own.

▸ Add the following method to the class:

```
- (UIView *)tableView:(UITableView *)tableView
            viewForHeaderInSection:(NSInteger)section
{
  UILabel *label = [[UILabel alloc] initWithFrame:
      CGRectMake(15.0f, tableView.sectionHeaderHeight - 14.0f,
                                                 300.0f, 14.0f)];
  label.font = [UIFont boldSystemFontOfSize:11.0f];
  label.text = [tableView.dataSource tableView:tableView
                                titleForHeaderInSection:section];
  label.textColor = [UIColor colorWithWhite:1.0f alpha:0.4f];
  label.backgroundColor = [UIColor clearColor];

  UIView *separator = [[UIView alloc] initWithFrame:
       CGRectMake(15.0f, tableView.sectionHeaderHeight - 0.5f,
                    tableView.bounds.size.width - 15.0f, 0.5f)];
  separator.backgroundColor = tableView.separatorColor;

  UIView *view = [[UIView alloc] initWithFrame:
         CGRectMake(0.0f, 0.0f, tableView.bounds.size.width,
                                  tableView.sectionHeaderHeight)];
  view.backgroundColor = [UIColor colorWithWhite:0.0f
                                           alpha:0.85f];
  [view addSubview:label];
  [view addSubview:separator];

  return view;
}
```

This is a `UITableView` delegate method. It gets called once for each section in the table view. Here you create a label for the section name, a 1-pixel high view that functions as a separator line, and a container view to hold these two subviews. It looks like this:



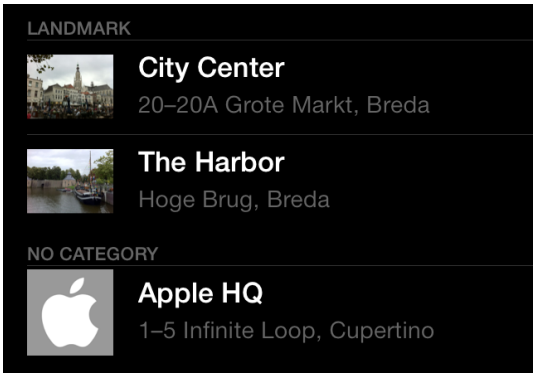
The section headers now draw much less attention to themselves

Another small improvement you can make here is to always put the section headers in uppercase.

➤ Change the `titleForHeaderInSection` data source method to:

```
- (NSString *)tableView:(UITableView *)tableView
             titleForHeaderInSection:(NSInteger)section
{
  id <NSFetchedResultsSectionInfo> sectionInfo =
           [self.fetchedResultsController sections][section];

  return [[sectionInfo name] uppercaseString];
}
```

Now the section headers look even better:



The section header text is in uppercase

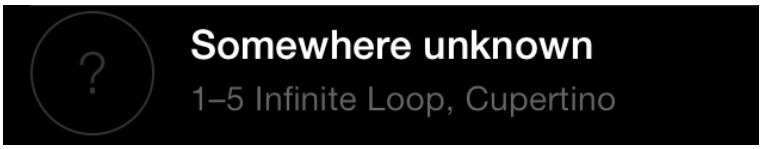
Currently if a location does not have a photo, there is a black gap where the thumbnail is supposed to be. That doesn't look very professional. It's better to show

a placeholder image. You already added one to the asset catalog when you imported the Images folder.

➤ In `configureCell:atIndexPath:`, just before the line that places `image` into the `photoImageView`, add these lines:

```
  if (image == nil) {
    image = [UIImage imageNamed:@"No Photo"];
  }
```

Now locations without photos appear like so:



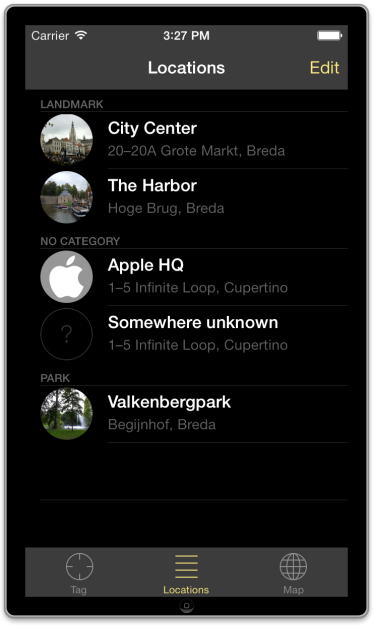
That makes it a lot clearer to the user that the photo is missing. (As opposed to, say, being a photo of a black hole.)

The placeholder image is round. That's the fashion for thumbnail images on iOS these days, and it's pretty easy to make the other thumbnails rounded too.

➤ Add the following lines to the bottom of `configureCell:atIndexPath:`

```
locationCell.photoImageView.layer.cornerRadius =
          locationCell.photoImageView.bounds.size.width / 2.0f;
locationCell.photoImageView.clipsToBounds = YES;
locationCell.separatorInset = UIEdgeInsetsMake(0, 82, 0, 0);
```

This gives the image view rounded corners with a radius that is equal to half the width of the image, which makes it a perfect circle. The `clipsToBounds` setting makes sure that the image view respects these rounded corners and does not draw outside them. The `separatorInset` moves the separator lines between the cells a bit to the right so there are no lines between the thumbnail images.



The thumbnails live inside little circles

> **Note:** The rounded thumbnails don't look very good if the original photo isn't square. You may want to change the Mode of the image view back to Aspect Fill or Scale to Fill so that the thumbnail always fills up the entire image view.

There are two other table views in the app and they get a similar treatment.

➤ Add these lines to `viewDidLoad` in **LocationDetailsViewController.m**:

```
self.tableView.backgroundColor = [UIColor blackColor];
self.tableView.separatorColor = [UIColor colorWithWhite:1.0f
                                                  alpha:0.2f];

self.descriptionTextView.textColor = [UIColor whiteColor];
self.descriptionTextView.backgroundColor = [UIColor blackColor];

self.photoLabel.textColor = [UIColor whiteColor];
self.photoLabel.highlightedTextColor =
                                    self.photoLabel.textColor;

self.addressLabel.textColor =
                        [UIColor colorWithWhite:1.0f alpha:0.4f];
self.addressLabel.highlightedTextColor =
                                  self.addressLabel.textColor;
```

This is similar to what you did before. It changes the colors of the table view (but not the cells) and some of the other controls. This table view controller has static

cells so there is no `cellForRowAtIndexPath` or `configureCell` method that you can use to change the colors of the cells and their labels. However, the table view delegate has a handy method that comes in useful here.

➤ Add the following method:

```
- (void)tableView:(UITableView *)tableView
        willDisplayCell:(UITableViewCell *)cell
        forRowAtIndexPath:(NSIndexPath *)indexPath
{
  cell.backgroundColor = [UIColor blackColor];

  cell.textLabel.textColor = [UIColor whiteColor];
  cell.textLabel.highlightedTextColor =
                                      cell.textLabel.textColor;
  cell.detailTextLabel.textColor =
                        [UIColor colorWithWhite:1.0f alpha:0.4f];
  cell.detailTextLabel.highlightedTextColor =
                                  cell.detailTextLabel.textColor;

  UIView *selectionView = [[UIView alloc]
                                       initWithFrame:CGRectZero];
  selectionView.backgroundColor =
                        [UIColor colorWithWhite:1.0f alpha:0.2f];
  cell.selectedBackgroundView = selectionView;
}
```

The `willDisplayCell` delegate method is called just before a cell becomes visible. Here you can do some last-minute customizations on the cell and its contents.

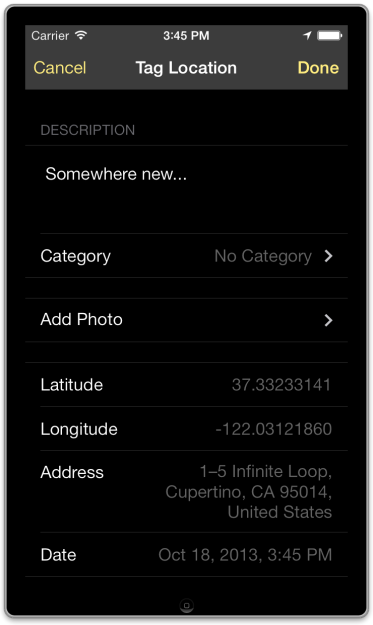
If you run the app now and tag a new location, you'll see that the label that used to say "Address" is not visible. That's because this cell does not use one of the built-in cell types so it does not have anything connected to its `textLabel` and `detailTextLabel` properties. You could make a new outlet for this label but you can also do it as follows:

➤ Set the **Tag** of the "Address" label to 100 in the storyboard.

➤ Add these lines to `willDisplayCell`:

```
if (indexPath.row == 2) {
  UILabel *addressLabel = (UILabel *)[cell viewWithTag:100];
  addressLabel.textColor = [UIColor whiteColor];
  addressLabel.highlightedTextColor = addressLabel.textColor;
}
```

The Tag Location screen now looks like this:



The Tag Location screen with styling applied

The final table view is the category picker.

➤ Add these two lines to `viewDidLoad` in **CategoryPickerViewController.m**:

```
self.tableView.backgroundColor = [UIColor blackColor];
self.tableView.separatorColor = [UIColor colorWithWhite:1.0f
                                                  alpha:0.2f];
```

➤ And give this class a `willDisplayCell` method too:

```
- (void)tableView:(UITableView *)tableView
        willDisplayCell:(UITableViewCell *)cell
        forRowAtIndexPath:(NSIndexPath *)indexPath
{
  cell.backgroundColor = [UIColor blackColor];

  cell.textLabel.textColor = [UIColor whiteColor];
  cell.textLabel.highlightedTextColor =
                                      cell.textLabel.textColor;

  UIView *selectionView = [[UIView alloc]
                                     initWithFrame:CGRectZero];
  selectionView.backgroundColor =
                        [UIColor colorWithWhite:1.0f alpha:0.2f];
  cell.selectedBackgroundView = selectionView;
}
```

Now the category picker is dressed in black as well. It's a bit of work to change the visuals of all these table views by hand, but it's worth it.

One tiny thing: the photo picker still uses the standard blue tint color. That makes its navigation bar buttons hard to read (blue text on a dark gray navigation bar). The fix is simple: set the tint color on the `UIImagePickerController` just before you present it.

➤ In **LocationDetailsViewController.m**, add the following line to `takePhoto` and `choosePhotoFromLibrary`:

```
_imagePicker.view.tintColor = self.view.tintColor;
```

Now the Cancel button appears in yellow instead of blue.

Polishing up the main screen

I'm pretty happy with all the other screens but the main screen needs a bit more work to be presentable.

Here's what you'll do:

- Show a logo when the app starts up. Normally such splash screens are bad for the user experience, but here you can get away with it.
- Make the logo disappear with an animation when the user taps Get My Location.
- While the app is fetching the coordinates, show an animated activity spinner to make it even clearer to the user that something is going on.
- Hide the Latitude: and Longitude: labels until the app has actually found coordinates.

First you need to move the labels into a new container subview.

➤ Open the storyboard and go to the Current Location View Controller. In the outline pane, select the six labels and the Tag Location button. With these seven views selected, choose **Editor → Embed In → View** from the Xcode menubar.

This creates a blank, white `UIView` and puts these labels and the button inside that new view.

➤ Change the **Background Color** of this new container view to **Clear Color**, so that everything becomes visible again.

The layout of the screen hasn't changed; you have simply reorganized the view hierarchy so that you can easily manipulate and animate this group of labels as a whole. Grouping views in a container view is a common technique for building complex layouts.

You will hide the **Latitude:** and **Longitude:** labels from the screen until the app actually has some coordinates to display. The only label that will be visible until

then is the one on the top and it will say "Searching..." or give some kind of error message. In order to do this, you must have outlets for these two labels.

➤ Add the following properties to **CurrentLocationViewController.h**:

```
@property (nonatomic, weak) IBOutlet UILabel *latitudeTextLabel;
@property (nonatomic, weak) IBOutlet UILabel
                                          *longitudeTextLabel;
@property (nonatomic, weak) IBOutlet UIView *containerView;
```

Note: If instead you want to add these outlets to the class extension in the .m file, like you've been doing with the other view controllers, then that's perfectly fine. There is no reason why other objects in this app should have access to them, so making these outlets private is good practice. (That includes the other outlets on this view controller.)

You'll put all the logic for updating the labels into one single place, the `updateLabels` method, so that hiding and showing these labels is pretty straightforward.

➤ Change the `updateLabels` method in **CurrentLocationViewController.m** to:

```
- (void)updateLabels
{
  if (_location != nil) {
    . . .

    self.latitudeTextLabel.hidden = NO;
    self.longitudeTextLabel.hidden = NO;

  } else {
    . . .

    self.latitudeTextLabel.hidden = YES;
    self.longitudeTextLabel.hidden = YES;
  }
}
```

➤ Connect the **Latitude:** and **Longitude:** labels in the storyboard editor to the `latitudeTextLabel` and `longitudeTextLabel` outlets. Connect the `UIView` that contains all these labels to the `containerView` outlet.

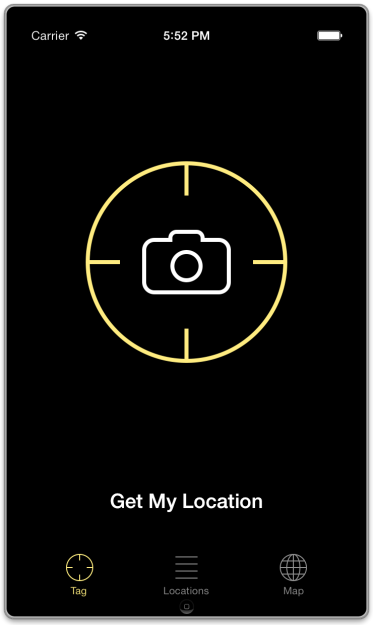
➤ Run the app and verify that the **Latitude:** and **Longitude:** labels only appear when you have obtained GPS coordinates.

The first impression

The main screen looks decent and is completely functional but it could do with more pizzazz. It lacks the "Wow!" factor. You want to impress users the first time they

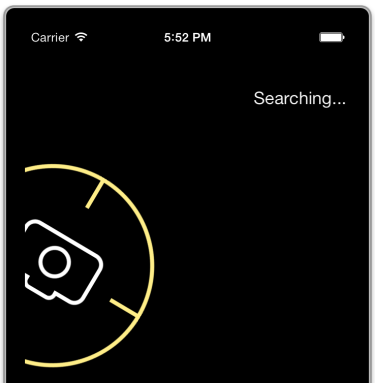
start your app, in order to keep them coming back. To pull this off, you'll add a logo and a cool animation.

When the user hasn't yet pressed the Get My Location button, there are no GPS coordinates and the Tag Location button is hidden. Instead of showing the upper panel with absolutely no information in it, you can show a large version of the app's icon:



The welcome screen of MyLocations

When the user taps the Get My Location button, the icon rolls out of the screen (it's round so that kinda makes sense) and the panel with the GPS status slides in:



The logo rolls out of the screen while the panel slides in

This is pretty easy to program thanks to the power of Core Animation and it makes the app a whole lot more impressive to first-time users.

➤ Add the following instance variables to **CurrentLocationViewController.m**:

```
UIButton *_logoButton;
BOOL _logoVisible;
```

The logo image is actually a button, so that you can tap the logo to get started. The app will show this button when it starts up, and when it doesn't have anything better to display (after you press Stop and there are no coordinates and no error). To orchestrate this, you'll use the boolean `_logoVisible`.

➤ Add the following method to the bottom of the class:

```
#pragma mark - Logo View

- (void)showLogoView
{
  if (_logoVisible) {
    return;
  }

  _logoVisible = YES;
  self.containerView.hidden = YES;

  _logoButton = [UIButton buttonWithType:UIButtonTypeCustom];
  [_logoButton setBackgroundImage:
    [UIImage imageNamed:@"Logo"] forState:UIControlStateNormal];
  [_logoButton sizeToFit];
  [_logoButton addTarget:self action:@selector(getLocation:)
                forControlEvents:UIControlEventTouchUpInside];
  _logoButton.center = CGPointMake(
          self.view.bounds.size.width / 2.0f,
          self.view.bounds.size.height / 2.0f - 49.0f);

  [self.view addSubview:_logoButton];
}
```

This hides the container view so the labels disappear, and creates the `_logoButton` object. This is a "custom" type `UIButton`, meaning that it has no title text or other frills. It draws the **Logo.png** image and calls the `getLocation:` method when tapped.

➤ In `updateLabels`, change the line that says,

```
statusMessage = @"Press the Button to Start";
```

into:

```
statusMessage = @"";
[self showLogoView];
```

This new logic makes the logo appear when there are no coordinates or error messages to display. That's also the state at startup time, so when you run the app now, you should be greeted by the logo.

➤ Run the app. In the Simulator's **Debug** menu, choose **Location → None**. Tap on Get My Location to make the logo disappear. Tap Stop to make the logo appear again. Notice that the logo is higher up on the screen than before.

That happens because `showLogoView` uses `self.view.bounds.size.height` to calculate the position of the `_logoButton`. This initially happens in `viewDidLoad`, which calls `updateLabels`, which calls `showLogoView`. At this point the definitive height of the view isn't known yet, so UIKit uses the height from the storyboard. But because the tab bar is not translucent on this screen, UIKit decides to make the view 49 points smaller (the height of the tab bar). This happens some time after `viewDidLoad` finishes. When `showLogoView` is called at a later point, for example after you press Stop, `self.view.bounds.size.height` is missing 49 points and the button's Y-position is smaller.

➤ To fix this, change `viewDidLoad` and add the `viewWillLayoutSubviews` method:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  self.tabBarController.delegate = self;
  self.tabBarController.tabBar.translucent = NO;
}

- (void)viewWillLayoutSubviews
{
  [super viewWillLayoutSubviews];
  [self updateLabels];
  [self configureGetButton];
}
```

By the time `viewWillLayoutSubviews` is called, the final height of the view is known, so this is a safe place to calculate the position for the `_logoButton`. The moral of this story: you cannot depend on the main view's size in `viewDidLoad`.

When you tap the logo (or Get My Location), it should disappear and the panel with the labels should show up.

➤ Add the following method:

```
- (void)hideLogoView
{
  _logoVisible = NO;
  self.containerView.hidden = NO;
```

```
    [_logoButton removeFromSuperview];
    _logoButton = nil;
  }
```

This is the counterpart to `showLogoView`. For now, it simply removes the button with the logo and un-hides the container view with the GPS coordinates.

▸ Add the following to the top of `getLocation:`

```
- (IBAction)getLocation:(id)sender
{
  if (_logoVisible) {
    [self hideLogoView];
  }

  . . .
```

Before it starts the location manager, this first removes the logo from the screen if it was visible. Currently there is no animation code to be seen. When doing complicated layout stuff such as this, I always first want to make sure the basics work. If they do, you can make it look fancy with an animation.

▸ Run the app. You should see the screen with the logo. Press the Get My Location button and the logo is replaced by the coordinate labels.

Great, now that works you can add the animation. The only method you have to change is `hideLogoView`.

▸ Replace `hideLogoView` with:

```
- (void)hideLogoView
{
  if (!_logoVisible) {
    return;
  }

  _logoVisible = NO;
  self.containerView.hidden = NO;

  self.containerView.center = CGPointMake(
          self.view.bounds.size.width * 2.0f,
          40.0f + self.containerView.bounds.size.height / 2.0f);

  CABasicAnimation *panelMover = [CABasicAnimation
                                  animationWithKeyPath:@"position"];
  panelMover.removedOnCompletion = NO;
  panelMover.fillMode = kCAFillModeForwards;
  panelMover.duration = 0.6;
```

```
  panelMover.fromValue = [NSValue
                    valueWithCGPoint:self.containerView.center];
  panelMover.toValue = [NSValue valueWithCGPoint:
              CGPointMake(160.0f, self.containerView.center.y)];
  panelMover.timingFunction = [CAMediaTimingFunction
                functionWithName:kCAMediaTimingFunctionEaseOut];
  panelMover.delegate = self;
  [self.containerView.layer addAnimation:panelMover
                                  forKey:@"panelMover"];

  CABasicAnimation *logoMover = [CABasicAnimation
                               animationWithKeyPath:@"position"];
  logoMover.removedOnCompletion = NO;
  logoMover.fillMode = kCAFillModeForwards;
  logoMover.duration = 0.5;
  logoMover.fromValue = [NSValue
                           valueWithCGPoint:_logoButton.center];
  logoMover.toValue = [NSValue valueWithCGPoint:
                    CGPointMake(-160.0f, _logoButton.center.y)];
  logoMover.timingFunction = [CAMediaTimingFunction
                 functionWithName:kCAMediaTimingFunctionEaseIn];
  [_logoButton.layer addAnimation:logoMover
                           forKey:@"logoMover"];

  CABasicAnimation *logoRotator = [CABasicAnimation
                  animationWithKeyPath:@"transform.rotation.z"];
  logoRotator.removedOnCompletion = NO;
  logoRotator.fillMode = kCAFillModeForwards;
  logoRotator.duration = 0.5;
  logoRotator.fromValue = @0.0f;
  logoRotator.toValue = @(-2.0f * M_PI);
  logoRotator.timingFunction = [CAMediaTimingFunction
                 functionWithName:kCAMediaTimingFunctionEaseIn];
  [_logoButton.layer addAnimation:logoRotator
                           forKey:@"logoRotator"];
}
```

This creates three animations that are played at the same time: 1) the `containerView` is placed outside the screen (somewhere on the right) and moved to the center, while 2) the logo image view slides out of the screen and 3) at the same time rotates around its center, giving the impression that it's rolling away.

Because the "panelMover" animation takes longest, you set a delegate on it so that you will be notified when the entire animation is over. The methods for this delegate are not declared in a protocol, so there is no need to add anything to your `@interface`.

➤ Add the following method below `hideLogoView`:

```
- (void)animationDidStop:(CAAnimation *)anim finished:(BOOL)flag
{
  [self.containerView.layer removeAllAnimations];
  self.containerView.center = CGPointMake(
                  self.view.bounds.size.width / 2.0f, 40.0f +
                  self.containerView.bounds.size.height / 2.0f);

  [_logoButton.layer removeAllAnimations];
  [_logoButton removeFromSuperview];
  _logoButton = nil;
}
```

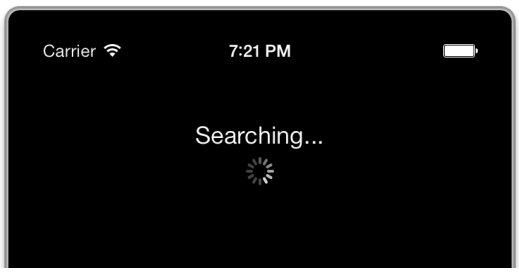
This cleans up after the animations and removes the logo button, as you no longer need it.

➤ Run the app. I think the animation looks pretty cool.

Apple says that good apps should "surprise and delight" and modest animations such as these really make your apps more interesting to use – as long as you don't overdo it!

Adding an activity indicator

When the user taps the Get My Location button, you change the button's text to Stop to indicate the change of state. You can make it even clearer to the user that something is going on by adding an animated activity "spinner". It will look like this:



The animated activity spinner shows that the app is busy

UIKit comes with a standard control for this, `UIActivityIndicatorView`. You're going to create this spinner control programmatically, because not everything has to be done in Interface Builder (even though you could).

➤ Add the following instance variable to **CurrentLocationViewController.m**:

```
UIActivityIndicatorView *_spinner;
```

The code to change the appearance of the Get My Location button sits in the `configureGetButton` method, so that's also a good place to show and hide the spinner.

➤ Change the `configureGetButton` method to the following:

```
- (void)configureGetButton
{
  if (_updatingLocation) {
    [self.getButton setTitle:@"Stop"
                    forState:UIControlStateNormal];

    if (_spinner == nil) {
      _spinner = [[UIActivityIndicatorView alloc]
                      initWithActivityIndicatorStyle:
                            UIActivityIndicatorViewStyleWhite];
      _spinner.center = CGPointMake(
          self.messageLabel.center.x,
          self.messageLabel.center.y +
                      _spinner.bounds.size.height/2.0f + 15.0f);
      [_spinner startAnimating];
      [self.containerView addSubview:_spinner];
    }
  } else {
    [self.getButton setTitle:@"Get My Location"
                    forState:UIControlStateNormal];

    [_spinner removeFromSuperview];
    _spinner = nil;
  }
}
```

In addition to changing the button text to Stop, you now create a new `UIActivityIndicatorView` instance and store it in the `_spinner` instance variable. Then you do some calculations to position the spinner view below the message label at the top of the screen. The call to `addSubview` makes the spinner visible on the screen.

When it's time to revert the button to its old state, you first call `removeFromSuperview` to remove the activity indicator view from the screen and then set the `_spinner` variable to `nil` to make sure the object is properly deallocated.

And that's all you need to do.

➤ Run the app. There should now be a cool little animation while the app is busy talking to the GPS satellites.

Make some noise

Visual feedback is important but you can't expect users to keep their eyes glued on the screen all the time, especially if an operation might take a few seconds or more. Emitting an unobtrusive sound is a good way to alert the user that a task is complete. When your iPhone has sent an email, for example, you hear a soft "Whoosh" sound.

You're going to add a sound effect to the app too, which is to be played when the first reverse geocoding successfully completes. That seems like a reasonable moment to alert the user that GPS and address information has been captured.

There are many ways to play sound on iOS but you're going to use one of the simplest: system sounds. The System Sound API is intended for short beeps and other notification sounds, which is exactly the type of sound that you want to play here.

▸ Add an import for AudioToolbox, the framework for playing system sounds, to the top of **CurrentLocationViewController.m**:

```
#import <AudioToolbox/AudioServices.h>
```

▸ Add the `_soundID` instance variable:

```
SystemSoundID _soundID;
```

▸ Add the following methods to the bottom of the class:

```
#pragma mark - Sound Effect

- (void)loadSoundEffect
{
  NSString *path = [[NSBundle mainBundle]
                       pathForResource:@"Sound.caf" ofType:nil];

  NSURL *fileURL = [NSURL fileURLWithPath:path isDirectory:NO];
  if (fileURL == nil) {
    NSLog(@"NSURL is nil for path: %@", path);
    return;
  }

  OSStatus error = AudioServicesCreateSystemSoundID(
                         (__bridge CFURLRef)fileURL, &_soundID);

  if (error != kAudioServicesNoError) {
    NSLog(@"Error code %ld loading sound at path: %@",
                                                  error, path);
    return;
```

```
  }
}

- (void)unloadSoundEffect
{
  AudioServicesDisposeSystemSoundID(_soundID);
  _soundID = 0;
}

- (void)playSoundEffect
{
  AudioServicesPlaySystemSound(_soundID);
}
```

The `loadSoundEffect` method loads the file **Sound.caf** and puts it into a new System Sound object. The specifics don't really matter, but you end up with a reference to that object in the `_soundID` instance variable.

▸ Call `loadSoundEffect` in `viewDidLoad`:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  self.tabBarController.delegate = self;
  self.tabBarController.tabBar.translucent = NO;
  [self loadSoundEffect];
}
```

▸ In `didUpdateLocations`, in the geocoder's completion block, change the following code:

```
        _lastGeocodingError = error;
        if (error == nil && [placemarks count] > 0) {
          if (_placemark == nil) {
            NSLog(@"FIRST TIME!");
            [self playSoundEffect];
          }
          _placemark = [placemarks lastObject];
        } else {
          _placemark = nil;
        }
```

The added if-statement simply checks whether the `_placemark` variable is `nil`, in which case this is the first time you've reverse geocoded an address. It then plays a sound using the `playSoundEffect` method.

Before you can build and run the app, you first have to link the project with the AudioToolbox framework.

➤ Add **AudioToolbox.framework** to the **Linked Frameworks and Libraries** section.

Of course, you shouldn't forget to include the actual sound effect!

➤ Add the **Sound** folder from this tutorial's Resources to the project. Make sure **Copy items into destination group's folder (if needed)** is selected.

➤ Run the app and see if you can let it make some noise. The sound should only be played for the first address it finds, even if more precise locations keep coming in afterwards.

CAF audio files

The Sound folder contains a single file, **Sound.caf**. The **caf** extension stands for Core Audio Format, and it's the preferred file format for these kinds of short audio files on iOS. If you want to use your own sound file but it is in a different format than CAF and your audio software can't save CAF files, then you can use the `afconvert` utility to convert the audio file. You need to run it from the Terminal:

```
$ /usr/bin/afconvert -f caff -d LEI16 Sound.wav Sound.caf
```

This converts the Sound.wav file into Sound.caf. You don't need to do this for the audio file from this tutorial's Sound folder because that file is already in the correct format. But if you want to experiment with your own audio files, then knowing how to use `afconvert` might be useful. (By the way, iOS can play .wav files just fine, but .caf is more optimal.)4

Supporting 3.5-inch devices

So far you've been designing and testing the app for 4-inch screens. But users of older models (anything before the iPhone 5) have screens with fewer pixels vertically and – like it or not – it's necessary for your apps to support those smaller screens.

➤ Run the app on the **iPhone Retina (3.5-inch)** Simulator or a 3.5-inch device.

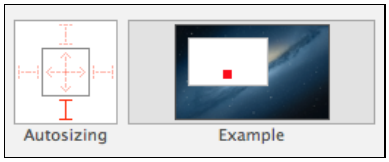
It's not too bad. Most of the screens resize without problems. There is only one real issue: the Get My Location button is no longer visible. Because this button was very near to the bottom of the 4-inch screen already, it simply drops off the screen of smaller devices. You will have to move it up a bit on those smaller screens.

Fortunately, this is really easy with **autosizing**. Auto Layout is all the rage today (you used it in the Bull's Eye chapter) but before Auto Layout was available, autosizing – also known as "springs & struts" – was the main tool for building

resizable user interface layouts. Each view has an autosizing setting that determines what happens to the size and position of that view when the size of its superview (i.e. the view that contains it) changes.

Because you disabled Auto Layout for this tutorial, you will use autosizing to keep the Get My Location button at a fixed distance from the bottom of the screen, no matter how large that screen is.

▸ Open the storyboard. Select the Get My Location button and go to the **Size inspector**. Change the autosizing options to the following:



The autosizing options connect the button to the bottom of its superview

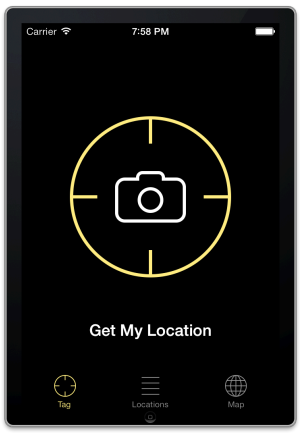
As you can see in the example animation on the right, the button (the red box) will now always be positioned relative to the bottom of its superview (the white box).

▸ Run the app. Whoops, the button now sits dangerously close to the logo. That wasn't the plan!

This happens because the tab bar is non-translucent on this screen and as a result the main view is resized during runtime. As I mentioned before, that makes the height of the main view 49 points smaller than what you see in Interface Builder. It also moves the Get My Location button too far up.

▸ In the storyboard, move the Get My Location button 49 points down, so that it sits at around Y = 500 and half overlaps the tab bar. It looks a little silly but it's necessary to position the button properly.

Now the app runs fine on 3.5-inch devices as well:



The app on the 3.5-inch Simulator

The icon and launch images

The Resources folder for this tutorial contains an **Icon** folder with the icons for this app and a **Launch Images** folder for the, you guessed it, launch images.

➤ Import the icon images into the asset catalog. Simply drag them from Finder into the **AppIcon** group:

The icons in the asset catalog

➤ No app is complete without a suitable Default.png image. Drag the files from the **Launch Images** folder into the asset catalog, under **LaunchImage**. Remember, the **-568h** file goes into the **R4** slot.

Done. That was easy. :-)

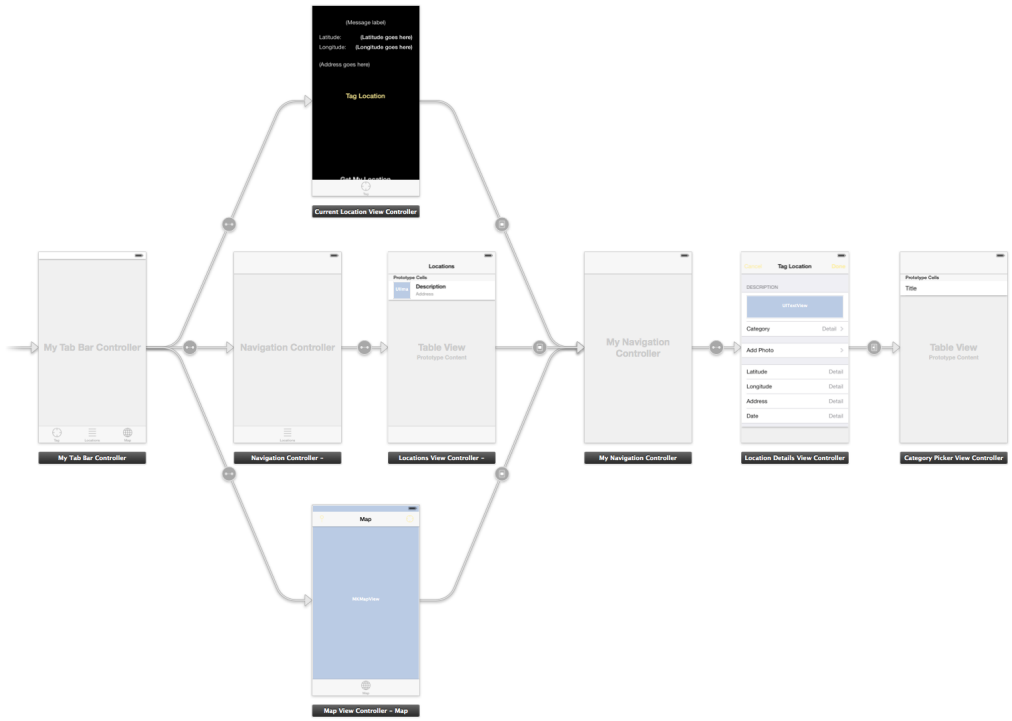
And with that, the MyLocations app is complete! Woohoo!

You can find the final project files for the app under **10 - Finished App** in the tutorial's Source Code folder.

The end

Congrats for making it this far! This has been another lengthy lesson with a lot of theory to boot. I hoped you learned a lot of useful stuff.

The final storyboard for the MyLocations app looks like this:



The final storyboard

In this lesson you took a more detailed look at Objective-C but there is still plenty to discover. To learn more about the Objective-C programming language, I recommend that you read the following books:

- **Programming in Objective-C 2.0 (2nd Edition)** by Stephen G. Kochan. A great introduction to both Objective-C and the underlying C language.
- **The C Programming Language** by Brian Kernighan and Dennis Ritchie. This book deals with the C language that Objective-C is based on and is considered essential reading by many. If someone tells you to go and read "K&R", then they mean this book.
- **Programming with Objective-C**. A very detailed document describing all the ins and outs of the language. You should probably read this a few months from now when you have more experience under your belt. It's a great reference. https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html

There are several good Core Data beginner books on the market, but if you want to get into the nitty gritty, then Apple's official **Core Data Programming Guide** is a must-read: https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html

Credits for this tutorial:

- Sound effect based on a flute sample by elmomo, downloaded from The Freesound Project (http://freesound.org)
- Image resizing category is based on code by Trevor Harmon (http://vocaro.com/trevor/blog/2009/10/12/resize-a-uiimage-the-right-way/)
- HudView code is based on MBProgressHud by Matej Bukovinski (https://github.com/matej/MBProgressHUD)