Up to date for iOS 10, Xcode 8 & Swift 3

iOS Apprentice

FIFTH EDITION

Tutorial 3: MyLocations

By Matthijs Hollemans

iOS Apprentice

Matthijs Hollemans

Notice of Rights

Notice of Liability

Trademarks

License

About the author

Matthijs Hollemans is a mystic who lives at the top of a mountain where he spends all of his days and nights coding up awesome apps. Actually he lives below sea level in the Netherlands and is pretty down-to-earth but he does spend too much time in Xcode. Check out his website at www.matthijshollemans.com.

About the cover

Striped dolphins live to about 55-60 years of age, can travel in pods numbering in the thousands and can dive to depths of 700 m to feed on fish, cephalopods and crustaceans. Baby dolphins don't sleep for a full a month after they're born. That puts two or three sleepless nights spent debugging code into perspective, doesn't it? :]

Table of Contents: Extended

3 Tutorial 3: MyLocations

By Matthijs Hollemans

The MyLocations app

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 Swift 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. Go somewhere

with your iPhone or iPod touch and press the Get My Location button to obtain GPS coordinates and the corresponding street address. 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!
4. Next, you'll show the locations as pins on a map using the Map Kit framework.
5. 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.
6. 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 Swift and object-oriented programming!

Swift review

In the past tutorials I've shown you a fair bit of the Swift 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, constants, and types

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

```
var count: Int
```

```
var shouldRemind: Bool
var text: String
var list: [ChecklistItem]
```

The **data type**, or just **type**, of a variable determines what kind of values it can contain. Some variables hold simple values such as `Int` and `Bool`, others hold more complex objects such as `String` and `Array`.

The basic types you've used so far are: `Int` for whole numbers, `Float` for numbers with decimals (also known as *floating-point* numbers), and `Bool` for boolean values (`true` and `false`).

There are a few other fundamental types as well:

- `Double`. Similar to a `Float` but with more precision. You will use `Doubles` later in this tutorial for storing latitude and longitude data.
- `Character`. Holds just a single character. A `String` is a collection of `Characters`.
- `UInt`. A variation on `Int` that you may encounter occasionally. The U stands for "unsigned", meaning the type can hold positive values only. It's called unsigned because it cannot have a negative sign (-) in front of the number. `UInt` can store numbers between 0 and 18 quintillion, but no negative numbers.
- `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`. 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 8 bytes for storage (a fact that you may immediately forget) and can fit positive and negative numbers up to about 19 digits. Those are big numbers!
- `CGFloat`. This isn't really a Swift type but a type defined by the iOS SDK. It's a decimal point number like `Float` and `Double`. For historical reasons, this is used throughout UIKit for floating-point values. (The "CG" prefix stands for the Core Graphics framework.)

Swift is very strict about types, more so than many other languages. If the type of a variable is `Int`, you cannot put a `Float` value into it. The other way around also won't work: an `Int` doesn't go into a `Float`.

Even though both types represent numbers of some sort, Swift won't automatically convert between different number types. You always need to convert the values explicitly.

For example:

```
var i = 10
var f: Float
f = i         // error
f = Float(i)  // OK
```

You don't always need to specify the type when you create a new variable. If you give the variable an initial value, Swift uses **type inference** to determine the type:

```
var i = 10              // Int
var d = 3.14            // Double
var b = true            // Bool
var s = "Hello, world"  // String
```

The integer value `10`, the floating-point value `3.14`, the boolean `true` and the string `"Hello, world"` are named **literal constants** or just **literals**.

Note that using the value 3.14 in the example above leads Swift to conclude that you want to use a `Double` here. If you intended to use a `Float` instead, you'd have to write:

```
var f: Float = 3.14
```

The "`: Float`" bit is called a **type annotation**. You use it to override the guess made by Swift's type inference mechanism, which may not always be what you intended.

Likewise, if you wanted the variable `i` to be a `Double` instead of an `Int`, you'd write:

```
var i: Double = 10
```

Or a little shorter, by giving the value 10 a decimal point:

```
var i = 10.0
```

These simple literals such as `10`, `3.14`, or `"Hello world"`, are useful only for creating variables of the basic types – `Int`, `Double`, `String`, and so on. To use more complex types, you'll need to **instantiate** an object first.

When you write the following,

```
var item: ChecklistItem
```

it only tells Swift you want to store a `ChecklistItem` object into the `item` variable, but it does not create that `ChecklistItem` object yet. For that you need to write:

```
item = ChecklistItem()
```

This first reserves memory to hold the object's data, followed by a call to `init()` to properly make the object ready for use. Reserving memory is also called **allocation**; filling up the object with its initial values is **initialization**.

The whole process is known as **instantiating** the object – you're making an object **instance**. The instance is the block of memory that holds the values of the object's variables (that's why they are called "instance variables", get it?).

Of course, you can combine the above into a single line:

```
var item = ChecklistItem()
```

Here you left out the ": ChecklistItem" type annotation because Swift is smart enough to realize that the type of `item` should be `ChecklistItem`.

However, you can't leave out the `()` parentheses – this is how Swift knows that you want to make a new `ChecklistItem` instance.

Some objects allow you to pass **parameters** to their `init` method. For example:

```
var item = ChecklistItem(text: "Charge my iPhone", checked: false)
```

This calls the corresponding `init(text, checked)` method to prepare the newly allocated `ChecklistItem` object for usage.

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" or properties) 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. Once the method ends, any local variables are destroyed.

```
class MyObject {
  var count = 0      // an instance variable

  func myMethod() {
    var temp: Int    // a local variable
    temp = count     // OK to use the instance variable here
  }

  // the local variable "temp" doesn't exist outside of the method
}
```

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:

```
class MyObject {
  var count = 7      // an instance variable

  func myMethod() {
    var count = 42   // local variable "shadows" instance variable
    print(count)     // prints 42
  }
}
```

Some developers place an underscore in front of their instance variable names to avoid this problem: `_count` instead of `count`. An alternative is to use the keyword `self` whenever you want to access an instance variable:

```
  func myMethod() {
    var count = 42
    print(self.count)   // prints 7
  }
```

Variables are not the only thing than can hold values. A variable is a container for a value that is allowed to *change* over the course of the app being run.

For example, in a note-taking app the user can change the text of the note, so you'd place that text into a `String` variable. Every time the user edits the text, the variable is updated.

Often you'll just want to store the result of a calculation or a method call into a temporary container, after which this value will never change. In that case, it is better to make this container a **constant** rather than a variable.

The following values never need to change once they've been set:

```
let pi = 3.141592
let difference = abs(targetValue - currentValue)
let message = "You scored \(points) points"
let image = UIImage(named: "SayCheese")
```

If a constant is local to a method, it's allowed to give the constant a new value the next time the method is called. The value from the previous method invocation has been destroyed when that method ended, and the next time the app enters that method you're creating a new constant with a new value (but with the same name). Of course, for the duration of that method call the constant's value must remain the same.

Tip: My suggestion is to use `let` for everything. That is the right solution 90% of the time. When you've got it wrong, the Swift compiler will warn that you're trying to change a constant. Only then you change it to a `var`. This ensures you're not making things variable that don't need to be.

When working with basic values such as integers and strings – so called **value types** – a constant created with `let` cannot be changed once it has been given a value:

```
let pi = 3.141592
pi = 3                // not allowed
```

However, with objects that are **reference types**, it is only the reference that is constant. The object itself can still be changed:

```
let item = ChecklistItem()
item.text = "Do the laundry"
item.checked = false
item.dueDate = yesterday
```

But this is not allowed:

```
let anotherItem = ChecklistItem()
item = anotherItem   // cannot change the reference
```

So how do you know what is a reference type and what is a value type?

Objects defined with `class` are reference types, while objects defined with `struct` or `enum` are value types. In practice this means most of the objects from the iOS SDK are reference types but things that are built into the Swift language, such as `Int`, `String`, and `Array`, are value types. (More about this important difference later.)

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 (`Array`) and dictionaries (`Dictionary`), 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.

```
// an array of ChecklistItem objects:
var items: Array<ChecklistItem>

// using shorthand notation:
var items: [ChecklistItem]

// making an instance of the array:
items = [ChecklistItem]()

// accessing an object from the array:
let item = items[3]
```

You can write an array as `Array<Type>` or `[Type]`. The first one is the official name, the second is "syntactic sugar" that is a bit easier to read. (Unlike other languages, you don't write `Type[]`. The type name goes inside the brackets.)

A **dictionary** stores key-value pairs. One object, usually a string, is the key that retrieves another object.

```
// a dictionary that stores (String, Int) pairs, for example a
// list of people's names and their ages:
var ages: Dictionary<String, Int>

// using shorthand notation:
var ages: [String: Int]

// making an instance of the dictionary:
ages = [String: Int]()

// accessing an object from the dictionary:
var age = dict["Jony Ive"]
```

The notation for retrieving an object from a dictionary looks very similar to reading from an array – both use the `[ ]` brackets. For indexing an array you always use a positive integer but for a dictionary you typically use a string.

There are other sorts of collections as well but array and dictionary are the most common ones.

`Array` and `Dictionary` are so-called **generics**, meaning that they are independent of the type of thing you want to store inside these collections.

You can have an `Array` of `Int` objects, but also an `Array` of `String` objects – or an `Array` of any kind of object, really (even an array of other arrays).

That's why before you can use `Array` you have to specify the type of object to store inside the array. In other words, you cannot write this:

```
var items: Array  // error: should be Array<TypeName>
var items: []     // error: should be [TypeName]
```

There should always be the name of a type inside the `[ ]` brackets or following the word `Array` in < > brackets. (If you're coming from Objective-C, be aware that the < > mean something completely different there.)

For `Dictionary`, you need to supply two type names: one for the type of the keys and one for the type of the values.

Swift requires that all variables and constants have a value. You can either specify a value when you declare the variable or constant, or by assigning a value inside an `init` method.

Sometimes it's useful to have a variable that can have no value, in which case you need to declare it as an **optional**:

```
var checklistToEdit: Checklist?
```

You cannot use this variable directly; you must always test whether it has a value or not. This is called **unwrapping** the optional:

```
if let checklist = checklistToEdit {
  // "checklist" now contains the real object
} else {
  // the optional was nil
}
```

The `age` variable from the dictionary example is actually an optional, because there is no guarantee that the dictionary contains the key "Jony Ive". Therefore, the type of `age` is `Int?` instead of just `Int`.

Before you can use a value from a dictionary, you need to unwrap it first using `if let`:

```
if let age = dict["Jony Ive"] {
  // use the value of age
}
```

If you are 100% sure that the dictionary contains the key you can also use **force**

unwrapping to read the corresponding value:

```
var age = dict["Jony Ive"]!
```

With the `!` you tell Swift, "This value will not be `nil`. I'll stake my reputation on it!" Of course, if you're wrong and the value *is* `nil`, the app will crash and your reputation is down the drain. Be careful with force unwrapping!

A slightly safer alternative to force unwrapping is **optional chaining**. For example, the following will crash the app if the `navigationController` property is `nil`:

```
navigationController!.delegate = self
```

But this won't:

```
navigationController?.delegate = self
```

Anything after the `?` will simply be ignored if `navigationController` does not have a value. It's equivalent to writing:

```
if navigationController != nil {
  navigationController!.delegate = self
}
```

It is also possible to declare an optional using an exclamation point instead of a question mark. This makes it an **implicitly unwrapped** optional:

```
var dataModel: DataModel!
```

Such a value is potentially unsafe because you can use it as a regular variable without having to unwrap it first. If this variable has the value `nil` when you don't expect it – and don't they always – your app will crash.

Optionals exist to guard against such crashes, and using `!` undermines the safety of using optionals.

However, sometimes using implicitly unwrapped optionals is more convenient than using pure optionals. Use them when you cannot give the variable an initial value at the time of declaration, nor in `init()`.

But once you've given the variable a value, you really ought not to make it `nil` again. If the value can become `nil` again, it's better to use a true optional with a question mark.

Methods and functions

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

When you call a method, the app jumps to that section of the code and executes all

the statements in the method one-by-one. As the end of the method is reached, the app jumps back to where it left off:

```
let result = performUselessCalculation(314)
print(result)

. . .

func performUselessCalculation(_ a: Int) -> Int {
  var b = Int(arc4random_uniform(100))
  var c = a / 2
  return (a + b) * c
}
```

Methods often return a value to the caller, usually the result of a computation or looking up something in a collection. The data type of the result value is written behind the `->` arrow. In the example above, it is `Int`. If there is no `->` arrow, the method does not return a value (also known as "`Void`").

Methods are **functions** that belong to an object, but there are also standalone functions such as `print()` and `arc4random_uniform()`.

Functions serve the same purpose as methods – they bundle functionality into small re-usable units – but live outside of any objects. Such functions are also called *free* functions or *global* functions.

These are examples of methods:

```
// Method that doesn't have any parameters, doesn't return a value.
override func viewDidLoad()

// Method that has one parameter, slider. It doesn't return a value.
// The keyword @IBAction means that this method can be connected
// to a control in Interface Builder.
@IBAction func sliderMoved(_ slider: UISlider)

// Method that doesn't have any parameters, returns an Int value.
func countUncheckedItems() -> Int

// Method with two parameters, cell and item, no return value.
// Note that the first parameter has an extra label, for, and the
// second parameter has an extra label, with.
func configureCheckmarkFor(for cell: UITableViewCell,
                           with item: ChecklistItem)

// Method with two parameters, tableView and section. Returns an Int.
// The _ means the first parameter does not have an external label.
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int

// Method with two parameters, tableView and indexPath.
// The question mark means it returns an optional IndexPath object
// (may also return nil).
override func tableView(_ tableView: UITableView,
                    willSelectRowAt indexPath: IndexPath) -> IndexPath?
```

To call a method on an object, you write `object.method(parameters)`. For example:

```
// Calling a method on the lists object:
lists.append(checklist)

// Calling a method with more than one parameter:
tableView.insertRows(at: indexPaths, with: .fade)
```

You can think of calling a method as *sending a message* from one object to another: "Hey `lists`, I'm sending you the `append` message for this `checklist` object."

The object whose method you're calling is known as the *receiver* of the message.

It is very common to call a method from the same object. Here, `loadChecklists()` calls the `sortChecklists()` method. Both are members of the `DataModel` object.

```
class DataModel {
  func loadChecklists() {
    . . .
    sortChecklists()  // this method also lives in DataModel
  }

  func sortChecklists() {
    . . .
  }
}
```

Sometimes this is written as:

```
  func loadChecklists() {
    . . .
    self.sortChecklists()
  }
```

The `self` keyword makes it clear that the `DataModel` object itself is the receiver of this message.

> **Note:** In these tutorials I leave out the `self` keyword for method calls, because it's not necessary to have it. Objective-C developers are very attached to `self`, so you'll probably see it used a lot in Swift too. It is a topic of heated debate in developer circles, but in the end the app doesn't really care whether you use `self` or not.

Inside a method you can also use `self` to get a reference to the object itself:

```
@IBAction func cancel() {
  delegate?.itemDetailViewControllerDidCancel(self)
}
```

Here `cancel()` sends a reference to the object (i.e. `self`) along to the delegate, so

the delegate knows who sent this `itemDetailViewControllerDidCancel()` message.

Also note the use of **optional chaining** here. The `delegate` property is an optional, so it can be `nil`. Using the question mark before the method call will ensure nothing bad happens if `delegate` is not set.

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. Consider `sumValuesFromArray()`, a method that has no parameters:

```
class MyObject {
  var numbers = [Int]()

  func sumValuesFromArray() -> Int {
    var total = 0
    for number in numbers {
      total += number
    }
    return total
  }
}
```

Here, `numbers` is an instance variable. The `sumValuesFromArray()` method is tied closely to that instance variable, and is useless without it.

Suppose you add a second array to the 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:

```
func sumValues(from array: [Int]) -> Int {
  var total = 0
  for number in array {
    total += number
  }
  return total
}
```

Now you can call this method with any `[Int]` or `Array<Int>` object as its parameter.

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

Often methods use two names for their parameters, the **external label** and the **internal label**. For example:

```
func downloadImage(for searchResult: SearchResult,
                   withTimeout timeout: TimeInterval,
                   andPlaceOn button: UIButton) {
  . . .
}
```

This method has three parameters: `searchResult`, `timeout`, and `button`. Those are the internal parameter names you'd use in the code inside the method.

The external labels become part of the method name. The full name for the method is `downloadImage(for, withTimeout, andPlaceOn)`. Method names in Swift are often quite long!

To call this method, you'd use the external labels:

```
downloadImage(for: result, withTimeout: 10, andPlaceOn: imageButton)
```

Sometimes you'll see a method whose first parameter does not have an external label, but has an _ underscore instead:

```
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int
```

This is often the case with delegate methods. It's a holdover from the Objective-C days, where the label for the first parameter was embedded in the first part of the method name. For example, in Objective-C the `downloadImage()` method would be named `downloadImageForSearchResult()`. These kinds of names should become less and less common in the near future.

Swift is pretty flexible with how it lets you name your methods, but it's smart to stick to the established conventions.

Inside a method you can do the following things:

- Create local variables and constants.
- Do basic arithmetic with mathematical operators such as `+`, `-`, `*`, `/`, and `%`.
- 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 = "\(count) Items"
}
```

The expression after `if` is called the **condition**. If a condition is true then the statements in the following `{ }` block are executed. The `else` section gets performed if none of the conditions are true.

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

`==` equal to

`!=` not equal

`>` greater than

`>=` greater than or equal

`<` less than

`<=` less than or equal

When you use the == operator, the contents of the objects are compared. This only returns true if `a` and `b` have the same value:

```
let a = "Hello, world"
let b = "Hello," + " world"
print(a == b)             // prints true
```

This is different from Objective-C, where == is only `true` if the two objects are the exact same instance in memory. However, in Swift == compares the values of the objects, not whether they actually occupy the same spot in memory. (If you need to do that use ===, the identity operator.)

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 `true` into `false`, and `false` into `true`. (Don't confuse this with the `!` that is used with optionals.)

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!

Swift has another very powerful construct in the language for making decisions, the `switch` statement:

```
switch condition {
  case value1:
    // statements

  case value2:
    // statements

  case value3:
    // statements

  default:
    // statements
}
```

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

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

In such a situation, the `switch` statement would be more convenient to use. Swift's

version of `switch` is more much powerful than the one in Objective-C. For example, you can match on ranges and other patterns:

```
switch difference {
  case 0:
    title = "Perfect!"
  case 1..<5:
    title = "You almost had it!"
  case 5..<10:
    title = "Pretty good!"
  default:
    title = "Not even close..."
}
```

The `..<` is the **half-open range** operator. It creates a range between the two numbers, but the top number is exclusive. So the half-open range `1..<5` is the same as the **closed range** `1...4`.

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:

```
func divide(_ a: Int, by b: Int) -> Int {
  if b == 0 {
    print("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:

```
func performDifficultCalculation(list: [Double]) {
  if list.count < 2 {
    print("Too few items in list")
    return
  }

  // perform the very difficult calculation here
}
```

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:

```
func performDifficultCalculation(list: [Double]) {
  if list.count < 2 {
    print("Too few items in list")
  } else {
    // perform the very difficult calculation here
  }
}
```

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:

```
func someMethod() {
  if condition1 {
    if condition2 {
      if condition3 {
        // 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:

```
func someMethod() {
  if !condition1 {
    // statements
    return
  }

  if !condition2 {
    // statements
    return
  }

  if !condition3 {
    // statements
    return
  }

  // statements
}
```

Both do exactly the same thing, but I find the second one much easier to understand. (Note that the conditions now use the `!` operator to invert their meaning.)

Swift even has a dedicated feature, `guard`, to help write this kind of code. It looks like this:

```
func someMethod() {
  guard condition1 else {
    // statements
    return
  }
```

```
  guard condition2 else {
    // statements
    return
  }
  . . .
```

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 in` statement for looping through an array:

```
for item in items {
  if !item.checked {
    count += 1
  }
}
```

Which can also be written as:

```
for item in items where !item.checked {
  count += 1
}
```

This performs the statements inside the `for in` block once for each object from the `items` array.

Note that the scope of the variable `item` 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.

Some languages, including Swift 2, have a for-statement that looks like this:

```
for var i = 0; i < 5; ++i {
  print(i)
}
```

When you run this code, it should print:

```
0
1
2
3
4
```

However, as of Swift 3.0 this kind of `for` loop is now removed from the language. Instead, you can loop over a range. This outputs the same as above:

```
for i in 0...4 {   // or 0..<5
  print(i)
}
```

By the way, you can also write this loop as:

```
for i in stride(from: 0, to: 5, by: 1) {
  print(i)
}
```

The `stride()` function creates a special object that represents the range 0 to 5 in increments of 1. If you wanted to show just the even numbers, you could change the `by` parameter to 2. You can even use `stride()` to count backwards if you give `by` a negative number.

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

```
repeat {
  // 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 loop that counts the `ChecklistItems` as follows using a `while` statement:

```
var count = 0
var i = 0
while i < items.count {
  let item = items[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.

Still, using a `while` is slightly more cumbersome than "`for item in items`", which is why you'll see `for in` used most of the time.

There really is no significant difference between using a `for`, `while`, or `repeat while` loop, except that one may be easier to read than the others, depending on what you're trying to do.

Note: `items.count` and `count` in this example are two different things with the same name. The first `count` is a property on the `items` array that returns the number of elements in that array; the second `count` is a local variable that contains the number of unchecked to-do items counted so far.

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:

```
var found = false
for item in array {
  if item == searchText {
    found = true
    break
  }
}
```

This example loops through the array until it finds an `item` that is equal to the value of `searchText` (presumably both are strings). Then it sets the variable `found` to `true` 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. You use `continue` to say, "I'm done with the current item, let's look at the next one."

Loops can often be replaced by *functional programming* constructs such as `map`, `filter`, or `reduce`. These are functions that operate on a collection, perform some code for each element, and return a new collection with the results.

For example, using `filter` on an array will only keep items if they satisfy a certain condition. To get a list of all the unchecked `ChecklistItem` objects, you'd write:

```
var uncheckedItems = items.filter { item in !item.checked }
```

That's a lot simpler than writing a loop. Functional programming is an advanced topic so we won't spend too much time on it here.

Objects

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

The data is made up of the object's instance variables and constants. We often refer to these as the object's **properties**. The functionality is provided by the object's methods.

In your Swift programs you will use existing objects, such as `String`, `Array`, `Date`, `UITableView`, and you'll also make your own.

To define a new object, you need a **MyObject.swift** file that contains a `class` section:

```
class MyObject {
  var text: String
  var count = 0
  let maximum = 100

  init() {
    text = "Hello world"
  }

  func doSomething() {
    // statements
  }
}
```

Inside the brackets for the class, you add properties (the instance variables and constants) and methods.

There are two types of properties:

- **stored properties** are the usual instance variables and constants
- **computed properties** don't store a value but perform logic when you read or write their values

This is an example of a computed property:

```
var indexOfSelectedChecklist: Int {
  get {
    return UserDefaults.standard().integerForKey("ChecklistIndex")
  }
  set {
    UserDefaults.standard().set(newValue, forKey: "ChecklistIndex")
  }
}
```

The `indexOfSelectedChecklist` property does not store a value like a normal variable would. Instead, every time someone uses this property it performs the code from the `get` or `set` block. The alternative would be to write separate `setIndexOfSelectedChecklist()` and `getIndexOfSelectedChecklist()` methods, but that doesn't read as nicely.

The keyword `@IBOutlet` means that a property can refer to a user interface element in Interface Builder, such as a label or button. Such properties are usually declared `weak` and optional. Similarly, the keyword `@IBAction` is used for methods that will be performed when the user interacts with the app.

There are three kinds of methods:

- instance methods
- class methods

- init methods

You know that a method is a function that belongs to an object. To call such a method you first need to have an instance of the object:

```
let myInstance = MyObject()   // create the object instance
. . .
myInstance.doSomething()      // call the method
```

You can also make **class methods**, which can be used without an instance. In fact, they are often used as "factory" methods, to create new instances:

```
class MyObject {
  . . .

  class func makeObject(text: String) -> MyObject {
    let m = MyObject()
    m.text = text
    return m
  }
}

let myInstance = MyObject.makeObject(text: "Hello world")
```

Init methods, or **initializers**, are used during the creation of new object instances. Instead of the above factory method you might as well use a custom `init` method:

```
class MyObject {
  . . .

  init(text: String) {
    self.text = text
  }
}

let myInstance = MyObject(text: "Hello world")
```

The main purpose of an `init` method is to fill in the object's instance variables. Any instance variables or constants that do not have a value yet must be given one in the `init` method.

Swift does not allow variables or constants to have no value (except for optionals), and `init` is your last chance to make this happen.

Objects can have more than one `init` method; which one you use depends on the circumstances.

A `UITableViewController`, for example, can be initialized either with `init?(coder)` when automatically loaded from a storyboard, with `init(nibName, bundle)` when manually loaded from a nib file, or with `init(style)` when constructed without a storyboard or nib. Sometimes you use one, sometimes the other.

You can also provide a `deinit` method that gets called when the object is no longer

in use, just before it gets destroyed.

By the way, `class` isn't the only way to define an object in Swift. It also supports other types of objects such as `structs` and `enums`. You'll learn more about these later, so I won't give away the whole plot here (no spoilers!).

Protocols

Besides objects you can also define **protocols**. A protocol is simply a list of method names (and possibly properties as well):

```
protocol MyProtocol {
  func someMethod(value: Int)
  func anotherMethod() -> String
}
```

A protocol is like a job ad. It lists all the things that a candidate for a certain position in your company should be able to do.

But the ad itself doesn't do the job – it's just words printed in the careers section of the newspaper – so you need to hire an actual employee who can get the job done. That would be an object.

Objects need to indicate that they conform to a protocol:

```
class MyObject: MyProtocol {
  . . .
}
```

This object now has to provide an implementation for the methods listed in the protocol. (If not, it's fired!)

From then on you can refer to this object as a `MyObject` (because that is its class name) but also as a `MyProtocol` object:

```
var m1: MyObject = MyObject()
var m2: MyProtocol = MyObject()
```

To any part of the code using the `m2` variable, it doesn't matter that the object is really a `MyObject` under the hood. The type of `m2` is `MyProtocol`, not `MyObject`.

All your code sees is that `m2` is *some* object conforming to `MyProtocol`, but it's not important what sort of object that is.

In other words, you don't really care that your employee may also have another job on the side, as long as it doesn't interfere with the duties you've hired him for.

Protocols are often used to define **delegates** but they come in handy for other uses as well, as you'll find out later on.

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

Getting GPS coordinates

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

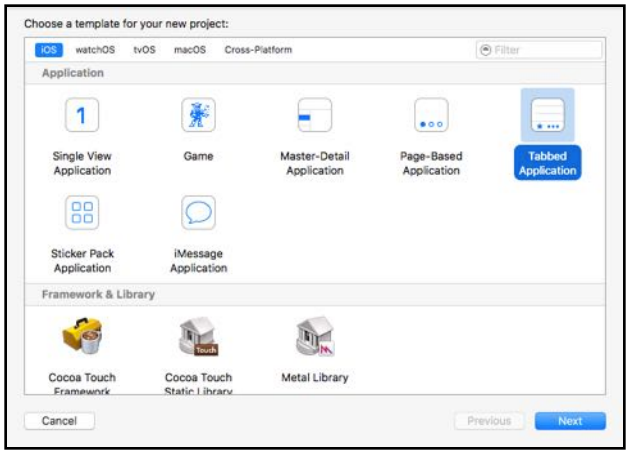
When you're done with this section, the app will look like this:

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 you 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
- Organization Identifier: Your own identifier in reverse domain notation
- Language: **Swift**
- Devices: **iPhone**
- Include Unit Tests and Include UI Tests: unchecked

➤ Save the project.

If you run the app, it looks like this:

The app from the Tabbed Application template

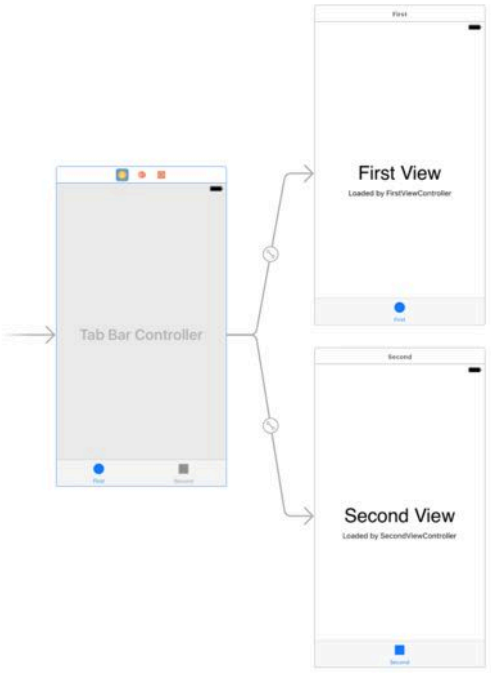
The app has a tab bar along the bottom with two tabs: First and Second.

Even though it doesn't do much yet, the app already employs three view controllers:

1. its *root controller* is the `UITabBarController` that contains the tab bar and performs the switching between the different screens;
2. the view controller for the First tab;
3. the view controller for the Second tab.

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!

As before, you'll be editing the storyboard using the iPhone SE dimensions, and then later you'll make the app work on the other iPhone models as well.

➤ In the **View as:** pane at the bottom, choose **iPhone SE**.

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.

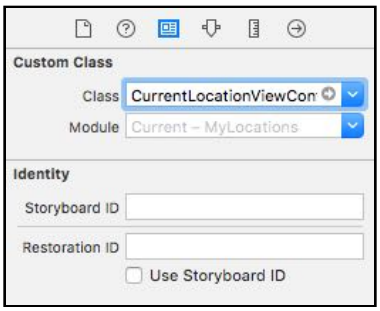
Let's give `FirstViewController` a better name.

➤ In the project navigator, click **FirstViewController.swift** twice (but not too quickly) and rename it to **CurrentLocationViewController.swift**.

➤ Open **CurrentLocationViewController.swift** and change the `class` line to:

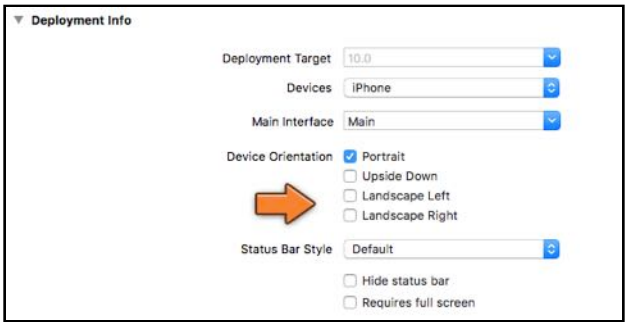
```
class CurrentLocationViewController: UIViewController {
```

➤ Switch to **Main.storyboard** and select the view controller connected to the first tab. In the **Identity inspector**, change the **Class** field from FirstViewController to **CurrentLocationViewController**:



Changing the class in Interface Builder

➤ 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.



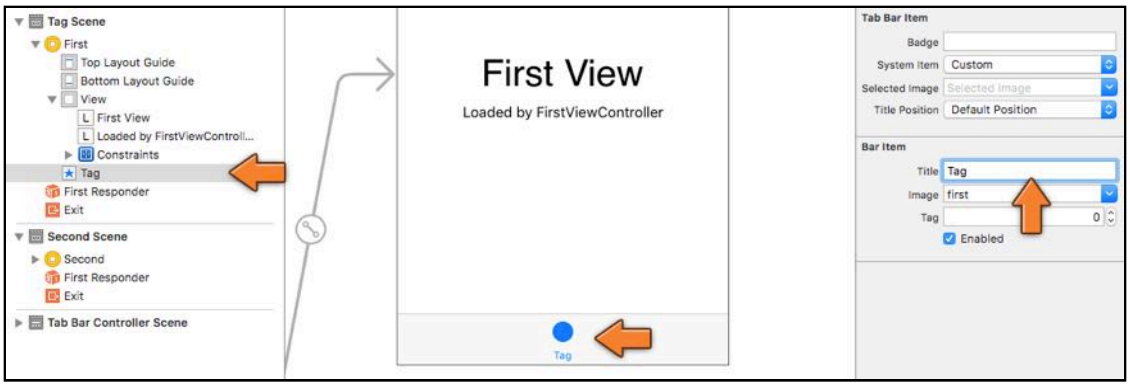
The app only works in portrait

➤ 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 in the Checklists app, 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 **Tab Bar Item** object from the "First Scene" (this is the Current Location View Controller) and go to the **Attributes inspector**. Change the **Title** to **Tag**.



Changing the title of the Tab Bar Item

Later on you'll also set an image on the Tab Bar Item; it currently uses the default image of a circle from the template.

You will now design the screen for this first tab. It gets two buttons and a few labels that show 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 the class inside **CurrentLocationViewController.swift**:

```
@IBOutlet weak var messageLabel: UILabel!
@IBOutlet weak var latitudeLabel: UILabel!
@IBOutlet weak var longitudeLabel: UILabel!
@IBOutlet weak var addressLabel: UILabel!
@IBOutlet weak var tagButton: UIButton!
@IBOutlet weak var getButton: UIButton!

@IBAction func getLocation() {
  // do nothing yet
}
```

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.

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

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!

> **Note:** Because you're initially designing for the iPhone SE screen size, it's best to use the iPhone SE Simulator to run the app. The app won't look good yet on the larger iPhone 6s, 7, and Plus simulators. You'll fix this later in the tutorial.

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. For devices with a compass it can also give the heading (you won't be using that in this tutorial). Core Location can also provide 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 just ask it for the current coordinates and see what happens.

➤ At the very top of **CurrentLocationViewController.swift**, add an import statement:

```
import CoreLocation
```

That is all you have to do to add the Core Location framework to your project.

Core Location, as so many other parts of the iOS SDK, works with a delegate, so you should make the view controller conform to the `CLLocationManagerDelegate` protocol.

➤ Add `CLLocationManagerDelegate` to the view controller's `class` line:

```
class CurrentLocationViewController: UIViewController,
                                     CLLocationManagerDelegate {
```

(This goes all on a single line.)

➤ Also add a new property:

```
let locationManager = CLLocationManager()
```

The `CLLocationManager` is the object that will give you the GPS coordinates. You're putting the reference to this object in a constant (using `let`), not a variable (`var`). Once you have created the location manager object, the value of `locationManager` will never have to change.

The new `CLLocationManager` object doesn't give out GPS coordinates right away. To begin receiving coordinates, you have to call its `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 only interested in receiving something from Core Location, just so you know that it works.

➤ Change the `getLocation()` action method to the following:

```
@IBAction func getLocation() {
  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 to receive locations with an accuracy of up to ten meters. Then you start the location manager. From that moment on the `CLLocationManager` object will send location updates to its delegate, i.e. the view controller.

➤ Speaking of the delegate, add the following code too:

```
// MARK: - CLLocationManagerDelegate
```

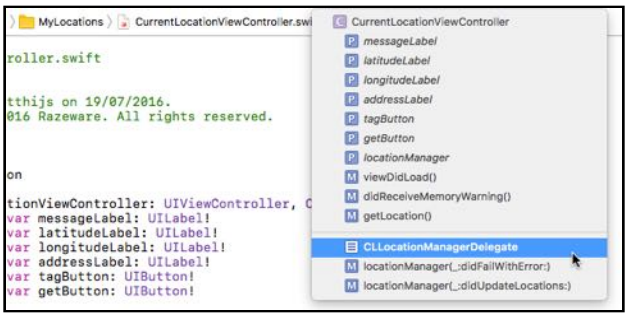
```
func locationManager(_ manager: CLLocationManager,
                     didFailWithError error: Error) {
  print("didFailWithError \(error)")
}

func locationManager(_ manager: CLLocationManager,
                     didUpdateLocations locations: [CLLocation]) {
  let newLocation = locations.last!
  print("didUpdateLocations \(newLocation)")
}
```

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

★ ★ ★

// MARK:

In the code above there is a comment line that starts with `// MARK:`. Such a comment gives a hint to Xcode that you have organized your source file into neat sections. You can see this from the Jump Bar:



The dash – makes Xcode draw a handy divider line. You can also use `// TODO:` and `// FIXME:` and Xcode will put those into the Jump Bar popup too.

★ ★ ★

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

Whoops, nothing seems to happen. That's because you need to ask for permission to use the user's location first.

➤ Add the following lines to the top of `getLocation()`:

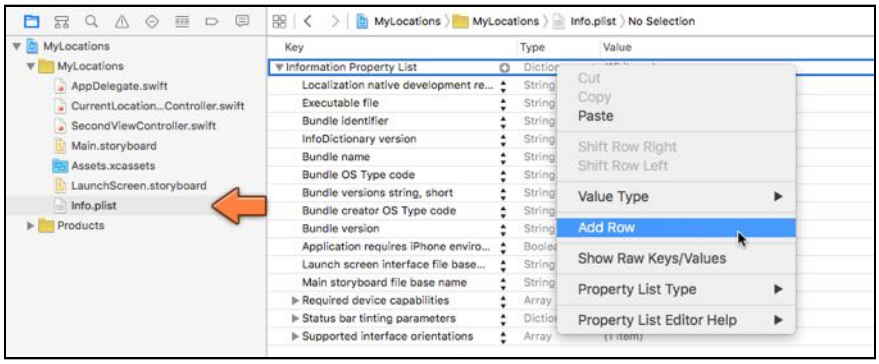
```
let authStatus = CLLocationManager.authorizationStatus()

if authStatus == .notDetermined {
  locationManager.requestWhenInUseAuthorization()
  return
}
```

This checks the current authorization status. If it is `.notDetermined`, meaning that this app has not asked for permission yet, then the app will request "When In Use" authorization. That allows the app to get location updates while it is open and the user is interacting with it.

There is also "Always" authorization, which permits the app to check the user's location even when it is not active. That's useful for a navigation app, for example. For most apps, including MyLocations, when-in-use is what you want to ask for.

Just adding these lines of code is not enough. You also have to add a special key to the app's Info.plist.

➤ Open **Info.plist** from the project navigator. Right-click somewhere inside Info.plist and choose **Add Row** from the menu.



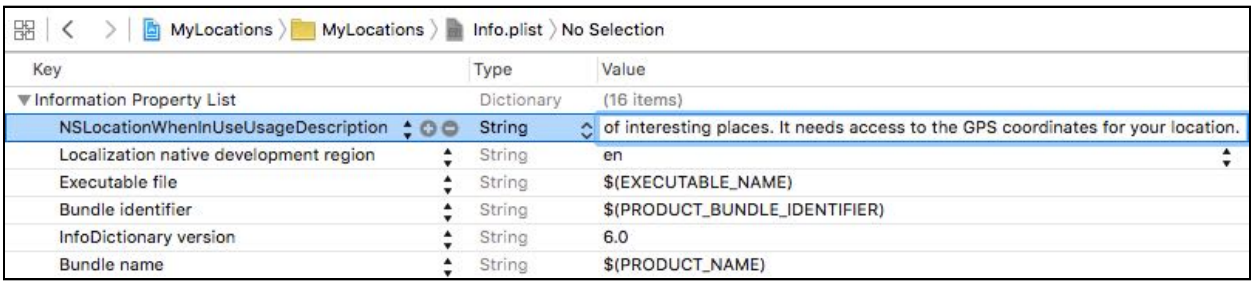
Adding a new row to Info.plist

➤ For the key, type **NSLocationWhenInUseUsageDescription** (or choose **Privacy - Location When In Use Usage Description** from the list).

➤ Type the following text in the Value column:

This app lets you keep track of interesting places. It needs access to the GPS coordinates for your location.

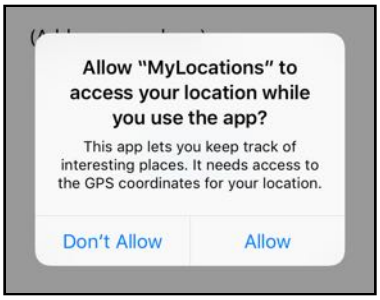
This description tells the user what the app wants to use the location data for.



Adding the new item to Info.plist

➤ Run the app again and press the Get My Location button.

Core Location will pop up the following alert, asking the user for permission:



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. Now press Get My Location again.

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

```
didFailWithError Error Domain=kCLErrorDomain Code=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 `Error` 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!).

This `Error` object 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 `CLError.denied`, which means the user did not allow the app to obtain location information.

Note: The `k` prefix is often used by the iOS frameworks to signify that a name represents a constant value (I guess whoever came up with this prefix thought

it was spelled "konstant"). This is an old convention and you won't see it used much in new frameworks or in Swift code, but it still pops up here and there.

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

When you press the Get My Location button, the app does not ask for permission anymore but immediately gives you the same error message.

Let's make this a bit more user-friendly, because a normal user would never see that `print()`.

➤ Add the following method to **CurrentLocationViewController.swift**:

```
func showLocationServicesDeniedAlert() {
  let alert = UIAlertController(title: "Location Services Disabled",
    message:
        "Please enable location services for this app in Settings.",
    preferredStyle: .alert)

  let okAction = UIAlertAction(title: "OK", style: .default,
                               handler: nil)
  alert.addAction(okAction)

  present(alert, animated: true, completion: nil)
}
```

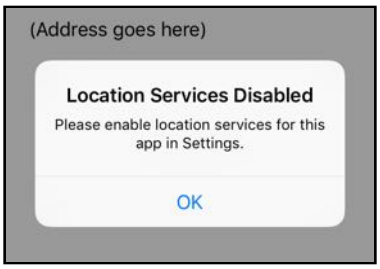
This pops up an alert with a helpful hint. This app is pretty useless without access to the user's location, so it should encourage the user to enable location services. It's not necessarily the user of the app who has denied access to the location data; a systems administrator or parent may have restricted location access.

➤ To show this alert, add the following lines to `getLocation()`, just before you set the `locationManager`'s delegate:

```
if authStatus == .denied || authStatus == .restricted {
  showLocationServicesDeniedAlert()
  return
}
```

This shows the alert if the authorization status is denied or restricted. Notice the use of `||` here, the "logical or" operator. `showLocationServicesDeniedAlert()` will be called if either of those two conditions is true.

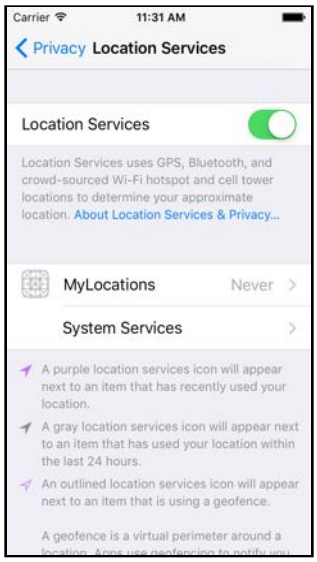
➤ Try it out. Run the app and tap Get My Location. You should now get the Location Services Disabled alert:



The alert that pops up when location services are not available

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

➤ Open the **Settings** app in the Simulator and go to **Privacy → Location Services**.



Location Services in the Settings app

➤ Click **MyLocations** and then **While Using the App** to enable location services again. Switch back to the app (or run it again from within Xcode) and press the Get My Location button.

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

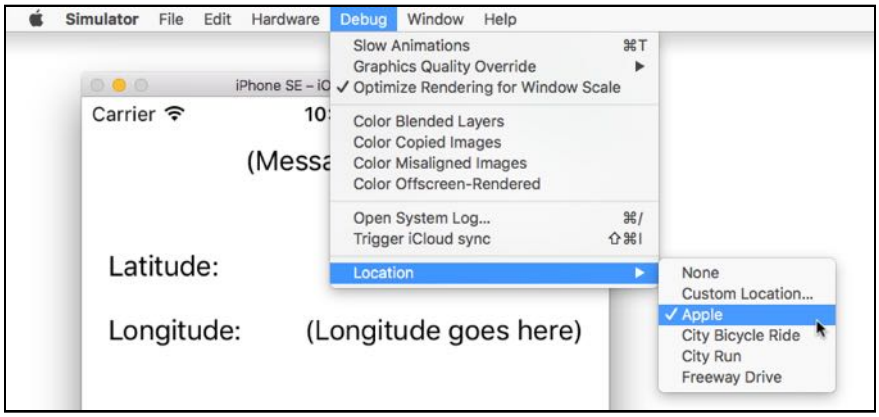
```
didFailWithError Error Domain=kCLErrorDomain Code=0
```

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:

```
didUpdateLocations <+37.33259552,-122.03031802> +/- 500.00m (speed -1.00
mps / course -1.00) @ 7/19/16 4:03:52 PM Central European Summer Time

didUpdateLocations <+37.33241023,-122.03051088> +/- 65.00m (speed -1.00
mps / course -1.00) @ 7/19/16 4:03:54 PM Central European Summer Time

didUpdateLocations <+37.33233141,-122.03121860> +/- 50.00m (speed -1.00
mps / course -1.00) @ 7/19/16 4:04:01 PM Central European Summer Time

didUpdateLocations <+37.33233141,-122.03121860> +/- 30.00m (speed 0.00
mps / course -1.00) @ 7/19/16 4:04:03 PM Central European Summer Time

didUpdateLocations <+37.33233141,-122.03121860> +/- 10.00m (speed 0.00
mps / course -1.00) @ 7/19/16 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 find 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.
- The very best results come 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. After you start an operation 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 can 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 an operation that is synchronous, the app won't continue until that operation is done. In effect, it freezes up.

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, switching to another tab during that time would have no effect. The user expects to always be able to change 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 will kill your app without mercy – so don't do that!

The take-away is this: Any operation that takes long enough to be noticeable by the user should be done asynchronously, in the background.

Putting the coordinates on the screen

The `locationManager(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 to **CurrentLocationViewController.swift**:

```
var location: CLLocation?
```

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

This needs to be an optional, because it is possible to *not* have a location, for example when you're stranded out in the Sahara desert somewhere and there is not a cell tower or GPS satellite in sight (it happens).

But even when everything works as it should, the value of `location` will still be `nil` until Core Location reports back with a valid `CLLocation` object, which as you've seen may take a few seconds. So an optional it is.

➤ Change the `locationManager(didUpdateLocations)` method to:

```
func locationManager(_ manager: CLLocationManager,
                     didUpdateLocations locations: [CLLocation]) {
  let newLocation = locations.last!
  print("didUpdateLocations \(newLocation)")

  location = newLocation
  updateLabels()
}
```

You store the `CLLocation` object that you get from the location manager into the instance variable and call a new `updateLabels()` method.

Keep the `print()` in there because it's handy for debugging.

➤ Add the `updateLabels()` method below:

```
func updateLabels() {
  if let location = location {
    latitudeLabel.text =
                  String(format: "%.8f", location.coordinate.latitude)
    longitudeLabel.text =
                  String(format: "%.8f", location.coordinate.longitude)
    tagButton.isHidden = false
    messageLabel.text = ""

  } else {
    latitudeLabel.text = ""
    longitudeLabel.text = ""
    addressLabel.text = ""
    tagButton.isHidden = true
    messageLabel.text = "Tap 'Get My Location' to Start"
  }
}
```

Because the `location` instance variable is an optional, you use the `if let` syntax to unwrap it.

Note that it's OK for the unwrapped variable to have the same name as the optional – here they are both called `location`. Inside the if-statement, `location` now refers to an actual `CLLocation` object that can never be `nil`.

If there is a valid location object, you convert the latitude and longitude, which are

values with type `Double`, into strings and put them into the labels.

You've seen *string interpolation* before to put values into strings, so why doesn't this code simply do the following?

```
latitudeLabel.text = "\(location.coordinate.latitude)"
```

That would certainly work, but it doesn't give you any control over how the latitude value appears. For this app, you want both latitude and longitude to be shown with 8 digits behind the decimal point.

For that sort of control, you need to use a so-called *format string*.

Like string interpolation, a format string uses placeholders that will be replaced by the actual value during runtime. These placeholders, or *format specifiers*, can be quite intricate.

To create the text for the latitude label you do this:

```
String(format: "%.8f", location.coordinate.latitude)
```

This creates a new `String` object using the format string `"%.8f"`, and the value to replace in that string, `location.coordinate.latitude`.

Placeholders always start with a `%` percent sign. Examples of common placeholders are: `%d` for integer values, `%f` for decimals, and `%@` for arbitrary objects.

Format strings are very common in Objective-C code but less so in Swift because string interpolation is much simpler, but also less powerful.

The `%.8f` format specifier does the same thing as `%f`: it takes a decimal number and puts it in the string. The `.8` 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

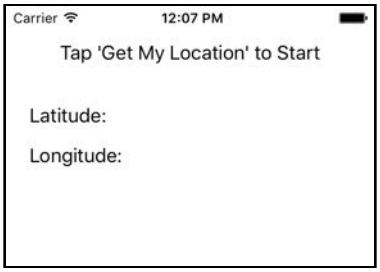
When the app starts up it has no location object (`location` is still `nil`) and therefore ought to show the "Tap 'Get My Location' to Start" message at the top as a hint to

the user. But it doesn't do that yet – the app doesn't call `updateLabels()` until it receives the first coordinates.

➤ To fix this, also call `updateLabels()` from `viewDidLoad()`:

```
override func viewDidLoad() {
  super.viewDidLoad()
  updateLabels()
}
```

➤ Run the app. Initially, the screen should now say, Tap 'Get My Location' to Start, and the latitude and longitude labels are empty.



What the app looks like when you start it

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 for this tutorial. 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 had 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.

➤ Add these two instance variables to **CurrentLocationViewController.swift**:

```
var updatingLocation = false
```

```
var lastLocationError: Error?
```

➤ Change the `locationManager(didFailWithError)` delegate method to the following:

```
func locationManager(_ manager: CLLocationManager,
                     didFailWithError error: Error) {
  print("didFailWithError \(error)")

  if (error as NSError).code == CLError.locationUnknown.rawValue {
    return
  }

  lastLocationError = error

  stopLocationManager()
  updateLabels()
}
```

The location manager may report errors for a variety of scenarios. You can look at the `code` property of the `Error` object to find out what type of error you're dealing with. (You do need to cast to `NSError` first. This is the subclass of `Error` that actually contains the `code` property.)

Some of the possible Core Location errors:

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

There are more (having to do with the compass and geocoding), but you get the point. Lots of reasons for things to go wrong!

Note: These error codes are defined in the `CLError` enumeration. Recall that an enumeration, or `enum`, is a list of values and names for these values.

The error codes used by Core Location have simple integer values. Rather than using the values 0, 1, 2 and so on in your program, Core Location has given them symbolic names using the `CLError` enum. That makes these codes easier to understand and you're less likely to pick the wrong one.

To convert these names back to an integer value you ask for the `rawValue`.

In your updated `locationManager(didFailWithError)`, you do:

```
if (error as NSError).code == CLError.locationUnknown.rawValue {
  return
}
```

The `CLError.locationUnknown` 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, you store the error object into a new instance variable, `lastLocationError`:

```
  lastLocationError = error
```

That way you can look up later what kind of error you were dealing with. This comes in useful in `updateLabels()`. You'll be extending that method shortly to show the error to the user because you don't want to leave them in the dark about such things.

Exercise. Can you explain why `lastLocationError` is an optional? ■

Answer: When there is no error, `lastLocationError` will not have a value. In other words it can be `nil`, and variables that can be `nil` must be optionals in Swift.

Finally, `locationManager(didFailWithError)` does:

```
  stopLocationManager()
  updateLabels()
```

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

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:

```
  func stopLocationManager() {
    if updatingLocation {
      locationManager.stopUpdatingLocation()
      locationManager.delegate = nil
      updatingLocation = false
    }
  }
```

There's an if-statement in here that checks whether the boolean instance variable `updatingLocation` is `true` or `false`. If it is `false`, 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.

➤ Put some extra code in `updateLabels()` to show the error message:

```
func updateLabels() {
  if let location = location {
    . . .

  } else {
    latitudeLabel.text = ""
    longitudeLabel.text = ""
    addressLabel.text = ""
    tagButton.isHidden = true

    // The new code starts here:
    let statusMessage: String
    if let error = lastLocationError as? NSError {
      if error.domain == kCLErrorDomain &&
         error.code == CLError.denied.rawValue {
        statusMessage = "Location Services Disabled"
      } else {
        statusMessage = "Error Getting Location"
      }
    } else if !CLLocationManager.locationServicesEnabled() {
      statusMessage = "Location Services Disabled"
    } else if updatingLocation {
      statusMessage = "Searching..."
    } else {
      statusMessage = "Tap 'Get My Location' to Start"
    }

    messageLabel.text = statusMessage
  }
}
```

The new 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 `CLError.denied` (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 it might still be impossible to get location coordinates if the user disabled Location Services completely on her 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. No one likes waiting, so it's nice to let the user know that the app is actively looking up her location. That is what you're using the `updatingLocation` boolean for.

> **Note:** 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…

➤ Also add the `startLocationManager()` method. I suggest you put it right above `stopLocationManager()`, to keep related functionality together:

```
func startLocationManager() {
  if CLLocationManager.locationServicesEnabled() {
    locationManager.delegate = self
    locationManager.desiredAccuracy =
                                kCLLocationAccuracyNearestTenMeters
    locationManager.startUpdatingLocation()
    updatingLocation = true
  }
}
```

Starting the location manager used to happen in the `getLocation()` action method. 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 `true`.

➤ Change `getLocation()` to:

```
@IBAction func getLocation() {
  let authStatus = CLLocationManager.authorizationStatus()

  if authStatus == .notDetermined {
    . . .
  }

  if authStatus == .denied || authStatus == .restricted {
    . . .
  }
```

```
    startLocationManager()
    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.

➤ At the bottom of `locationManager(didUpdateLocations)`, add the following line just before calling `updateLabels()`:

```
  lastLocationError = nil
```

This clears out the old error state. After receiving a valid coordinate, any previous error you may have encountered is no longer applicable.

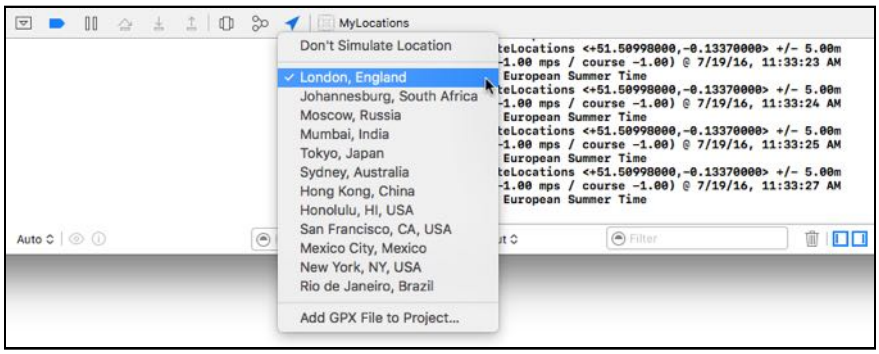
➤ Run the app. While the app is waiting for incoming coordinates, the label at the top should say "Searching..." until it finds a valid coordinate or encounters a fatal error.

The app is waiting to receive GPS coordinates

Play around with the Simulator's location settings for a while and see what happens when you choose different locations.

Note that changing the Simulator's location to None isn't an error anymore. This still returns the `.locationUnknown` 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 on the move, in which case their GPS coordinates *do* change.

However, you're not building a navigation app so for the purposes of this tutorial 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 `locationManager(didUpdateLocations)` to the following:

```
func locationManager(_ manager: CLLocationManager,
                     didUpdateLocations locations: [CLLocation]) {
  let newLocation = locations.last!
  print("didUpdateLocations \(newLocation)")

  // 1
  if newLocation.timestamp.timeIntervalSinceNow < -5 {
    return
  }
```

```
    // 2
    if newLocation.horizontalAccuracy < 0 {
      return
    }

    // 3
    if location == nil ||
       location!.horizontalAccuracy > newLocation.horizontalAccuracy {

      // 4
      lastLocationError = nil
      location = newLocation
      updateLabels()

      // 5
      if newLocation.horizontalAccuracy <=
                                    locationManager.desiredAccuracy {
        print("*** We're done!")
        stopLocationManager()
      }
    }
  }
```

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

1. If the time at which the given 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 in the last few seconds (obviously this does not take into consideration people with jet packs).

   You'll simply ignore these cached locations if they are too old.

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

3. 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 so you cannot assume that the next reading truly is always more accurate.

   Note that a larger accuracy value 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 optional instance variable that stores the `CLLocation` object that you obtained in a previous call to

"`didUpdateLocations`". If `location` is `nil` then this is the very first location update you're receiving and in that case you should also continue.

So if this is the very first location reading (`location` is `nil`) or the new location is more accurate than the previous reading, you continue to step 4. Otherwise you ignore this location update.

4. You've seen this before. It clears out any previous error if there was one and stores the new `CLLocation` object into the `location` variable.
5. If the new location's accuracy is equal to or better than the desired accuracy, you can call it a day and 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.

Short circuiting

Because `location` is an optional object you cannot access its properties directly – you first need to unwrap it. You could do that with `if let`, but if you're sure that the optional is not `nil` you can also force *unwrap* it with `!`.

That's what you are doing in this line:

```
    if location == nil ||
       location!.horizontalAccuracy > newLocation.horizontalAccuracy {
```

You wrote `location!.horizontalAccuracy` with an exclamation point instead of just `location.horizontalAccuracy`.

But what if `location == nil`, won't the force unwrapping fail then? Not in this case, because it is never performed.

The `||` operator (logical or) tests whether either of the two conditions is true. If the first one is true (`location` is `nil`), it will ignore the second condition. That's called *short circuiting*. There is no need for the app to check the second condition if the first one is already true.

So the app will only look at `location!.horizontalAccuracy` when `location` is guaranteed to be non-`nil`. Blows your mind, eh?

➤ 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 (but don't press Get My Location again). 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.

> **Note:** It's possible the above steps won't work for you. If the screen does not say "Searching…" but shows an old set of coordinates instead, then the Simulator is holding on to old location data. This seems to happen when you pick a location from within Xcode (using the arrow in the debug area) instead of the Simulator's Debug menu.
>
> The quickest way to fix this is to quit the Simulator and run the app again (which launches a new Simulator). If you can't get it to work, no worries, it's not that important. Just be aware that the Simulator can be finicky sometimes.

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 saving this location to their library right away, but at this point the location may not be accurate enough yet. 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 switch it 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 to **CurrentLocationViewController.swift**:

```
func configureGetButton() {
  if updatingLocation {
    getButton.setTitle("Stop", for: .normal)
  } else {
    getButton.setTitle("Get My Location", for: .normal)
  }
}
```

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 `configureGetButton()` from the following places:

```
override func viewDidLoad() {
```

```
    super.viewDidLoad()
    updateLabels()
    configureGetButton()
  }
```

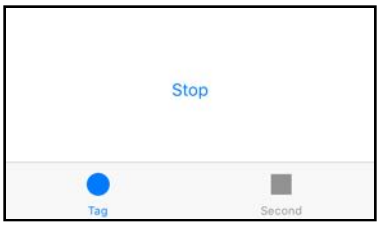
```
  @IBAction func getLocation() {
    . . .
    startLocationManager()
    updateLabels()
    configureGetButton()
  }
```

```
  func locationManager(_ manager: CLLocationManager,
                       didFailWithError error: Error) {
    . . .
    updateLabels()
    configureGetButton()
  }
```

```
  func locationManager(_ manager: CLLocationManager,
                       didUpdateLocations locations: [CLLocation]) {
      . . .
      if newLocation.horizontalAccuracy <=
                                    locationManager.desiredAccuracy {
        print("*** We're done!")
        stopLocationManager()
        configureGetButton()
      }
    }
  }
```

Anywhere you did `updateLabels()`, you're now also calling `configureGetButton()`. In `locationManager(didUpdateLocations)` you only call it when you're done.

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



The stop button

When a button says "Stop" you naturally 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 at all. 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 any taps on the button call this method.

➤ In `getLocation()`, replace the call to `startLocationManager()` with the following:

```
if updatingLocation {
  stopLocationManager()
} else {
  location = nil
  lastLocationError = nil
  startLocationManager()
}
```

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.

> **Note:** If the Stop button doesn't appear long enough for you to click it, set the location back to None first, tap Get My Location a few times, and then select the Apple location again.

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. (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.)

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 their 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. After all, it makes no sense to

reverse geocode the same set of coordinates over and over.

Reverse geocoding needs an active 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 properties to **CurrentLocationViewController.swift**:

```
let geocoder = CLGeocoder()
var placemark: CLPlacemark?
var performingReverseGeocoding = false
var lastGeocodingError: Error?
```

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.

The `placemark` variable needs to be an optional because it will have no value when there is no location yet, or when the location doesn't correspond to a street address (I don't think it will respond with "Sahara desert, Africa" but to be honest, I haven't been there to try).

You set `performingReverseGeocoding` to `true` when a geocoding operation is taking place, and `lastGeocodingError` will contain an `Error` object if something went wrong (or `nil` if there is no error).

➤ You'll put the geocoder to work in `locationManager(didUpdateLocations)`:

```
func locationManager(_ manager: CLLocationManager,
                     didUpdateLocations locations: [CLLocation]) {
  . . .
  if location == nil ||
     location!.horizontalAccuracy > newLocation.horizontalAccuracy {
    . . .

    if newLocation.horizontalAccuracy <=
                                  locationManager.desiredAccuracy {
      print("*** We're done!")
      stopLocationManager()
      configureGetButton()
    }

    // The new code begins here:
    if !performingReverseGeocoding {
      print("*** Going to geocode")

      performingReverseGeocoding = true

      geocoder.reverseGeocodeLocation(newLocation, completionHandler: {
        placemarks, error in

        print("*** Found placemarks: \(placemarks), error: \(error)")
      })
    }
    // End of the new code
  }
```

```
  }
```

The app should only perform a single reverse geocoding request at a time, so first you check whether it is not busy yet by looking at the `performingReverseGeocoding` variable. Then you start the geocoder.

The code looks straightforward enough, except… what is that `completionHandler` thing!?

Unlike the location manager, `CLGeocoder` does not use a delegate to tell you about the result, but something called a **closure**.

A closure is a block of code, much like a method or a function, that is written inline instead of in a separate method. The code inside a closure is usually not executed immediately but it is stored somewhere and performed at some later point.

Closures are an important Swift feature and you can expect to see them all over the place. (For Objective-C programmers, a closure is similar to a "block".)

Currently the closure contains just a `print()` – so you can see what's going on – but soon you'll add more code to it.

Unlike the rest of the code in the `locationManager(didUpdateLocations)` method, the code in this closure is not performed right away. After all, you can only print the geocoding results once the geocoding completes (which may be several seconds later).

The closure is kept for later by the `CLGeocoder` object and is only performed after `CLGeocoder` finds an address or encounters an error.

So why does `CLGeocoder` use a closure instead of a delegate?

The problem with using a delegate to provide feedback is that you need to write one or more separate methods. For example, for `CLLocationManager` those are the `locationManager(didUpdateLocations)` and `locationManager(didFailWithError)` methods.

By making different methods you move the code that deals with the response away from the code that makes the request. With closures, on the other hand, you can put that handling code in the same place. That makes the code more compact and easier to read. (Some APIs do both and you have a choice between using a closure or becoming a delegate.)

So when you write,

```
geocoder.reverseGeocodeLocation(newLocation, completionHandler: {
  placemarks, error in
    // put your statements here
}
```

you're telling 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 closure itself is:

```
{ placemarks, error in
    // put your statements here
}
```

The things before the `in` keyword – `placemarks` and `error` – are the parameters for this closure 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, it invokes the closure and executes the statements within. The `placemarks` parameter 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.

Just to rub it in: the statements in the closure are *not* executed right away when `locationManager(didUpdateLocations)` is called. Instead, the closure 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 closure.

It's the exact same principle as using delegate methods, except you're not putting the code into a separate method but in a closure.

It's OK if closures have got you scratching your head right now. You'll see them used many more times in this tutorial and the next.

➤ 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 (give it a second or two):

```
didUpdateLocations <+37.33240754,-122.03047460> +/- 65.00m (speed -1.00
mps / course -1.00) @ 7/19/16, 1:43:25 PM Central European Summer Time

didUpdateLocations <+37.33233141,-122.03121860> +/- 50.00m (speed -1.00
mps / course -1.00) @ 7/19/16, 1:43:30 PM Central European Summer Time

*** Going to geocode
*** Found placemarks: Optional([Apple Inc., Apple Inc., Cupertino, CA
95014, United States @ <+37.33202890,-122.02956600> +/- 100.00m, region
CLCircularRegion (identifier:'<+37.33214710,-122.03128175> radius
368.39', center:<+37.33214710,-122.03128175>, radius:368.39m)]), error:
nil
```

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. Nice!

Note: Several readers that reported that if you are in China and are trying to reverse geocode an address that is outside of China, you may get an error and `placemarks` will be `nil`. Try a location inside China instead.

➤ Add the following code to the closure, directly below the `print()` statement:

```
self.lastGeocodingError = error
if error == nil, let p = placemarks, !p.isEmpty {
  self.placemark = p.last!
} else {
  self.placemark = nil
}

self.performingReverseGeocoding = false
self.updateLabels()
```

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 this time, `lastGeocodingError`.

The next line does something you haven't seen before:

```
if error == nil, let p = placemarks, !p.isEmpty {
```

You know that `if let` is used to unwrap optionals. Here, `placemarks` is an optional so it needs be unwrapped before you can use it or you risk crashing the app when `placemarks` is `nil`. The unwrapped `placemarks` array gets the temporary name `p`.

The `!p.isEmpty` bit says that we should only enter this if-statement if the array of placemark objects is not empty.

You should read this line as:

```
if there's no error and the unwrapped placemarks array is not empty {
```

Of course, Swift doesn't speak English so you have to express this in terms that Swift understands.

You could also have written this as three different, nested if-statements:

```
if error == nil {
  if let p = placemarks {
    if !p.isEmpty {
```

But it's just as easy to combine all of these conditions into a single `if`.

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 have at least one object, but you're not going to trust that it always will. Good developers are suspicious!

If all three conditions are met – there is no error, the `placemarks` array is not `nil`, and there is at least one `CLPlacemark` inside this array – then you take the last of those `CLPlacemark` objects:

```
    self.placemark = p.last!
```

The `last` property refers to the last item from an array. It's an optional because there is no such item if the array is empty. As alternative you can also write `placemarks[placemarks.count - 1]` but that's not as tidy.

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 at a time, so you'll just pick the last one (which usually is the only one).

If there was an error during geocoding, you set `self.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 remember, not all GPS coordinates correspond to actual street addresses (there is no corner of 52nd and Broadway in the Sahara desert).

> **Note:** Did you notice that inside the `completionHandler` closure you used `self` to refer to the view controller's instance variables and methods? This is a Swift requirement.
>
> Closures are said to *capture* all the variables they use and `self` is one of them. You may immediately forget about that; just know that Swift requires that all captured variables are explicitly mentioned.
>
> As you've seen, outside a closure you can use `self` to refer to instance variables and methods, but it's not a requirement. However, it is a compiler error to leave out `self` inside a closure, so you don't have much choice there.

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

➤ Change `updateLabels()` to:

```
func updateLabels() {
  if let location = location {
    . . .
    if let placemark = placemark {
      addressLabel.text = string(from: placemark)
    } else if performingReverseGeocoding {
      addressLabel.text = "Searching for Address..."
    } else if lastGeocodingError != nil {
      addressLabel.text = "Error Finding Address"
    } else {
```

```
        addressLabel.text = "No Address Found"
      }
    } else {
      . . .
    }
  }
```

Because you only do the address lookup once the app has a valid 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 `string(from)` method:

```
  func string(from placemark: CLPlacemark) -> String {
    // 1
    var line1 = ""

    // 2
    if let s = placemark.subThoroughfare {
      line1 += s + " "
    }

    // 3
    if let s = placemark.thoroughfare {
      line1 += s
    }

    // 4
    var line2 = ""

    if let s = placemark.locality {
      line2 += s + " "
    }
    if let s = placemark.administrativeArea {
      line2 += s + " "
    }
    if let s = placemark.postalCode {
      line2 += s
    }

    // 5
    return line1 + "\n" + line2
  }
```

Let's look at this in detail:

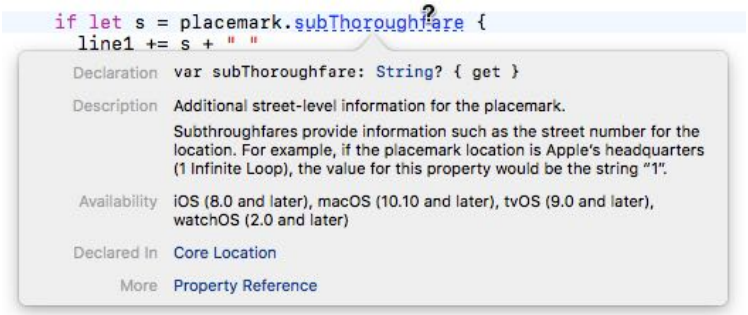
1. Create new string variable for the first line of text.
2. If the placemark has a `subThoroughfare`, add it to the string. This is an optional property, so you unwrap it with `if let` first. Just so you know, `subThoroughfare` is a fancy name for house number.

3. Adding the `thoroughfare` (or street name) is done similarly. Note that you put a space between it and `subThoroughfare` so they don't get glued together.

4. The same logic goes for the second line of text. This adds the locality (the city), administrative area (the state or province), and postal code (or zip code), with spaces between them where appropriate.

5. Finally, the two lines are concatenated (added together) with a newline character in between. The `\n` adds the line break into the string.

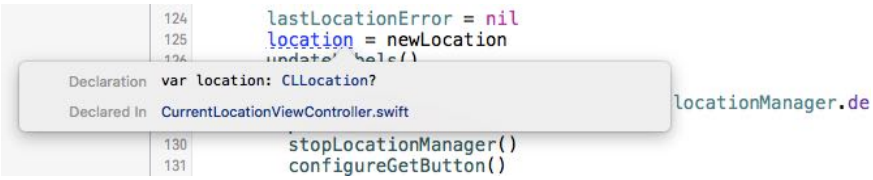
Tip: If you're ever wondering what some property or method name means, hold down the Option key and click the name to get a pop-up with a short description:



Viewing documentation by Option-clicking

(If you get an empty pop-up, then open the Xcode Preferences window and under Components, Documentation first install the iOS 10 documentation.)

This also works for your own variables. Swift's type inference is great but it also makes it harder to see the data types of your variables. Option-click keeps you from having to dig through the code to find that out:



Option-click your own variables to see their type

➤ In `getLocation()`, clear out the `placemark` and `lastGeocodingError` variables to start with a clean slate. Put this just above the call to `startLocationManager()`:

```
placemark = nil
lastGeocodingError = nil
```

➤ 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

It's fairly common that street numbers or other details are missing from the address. The `CLPlacemark` object may contain incomplete information, which is why its properties are all optionals. Geocoding is not an exact science!

> **Note:** For some reason, UK postal codes are missing the last few characters. The above address should really be London England W1J 9HP. This appears to be a bug in Core Location.

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.

> **Note:** If you're playing with different locations in the Simulator or from the Xcode debugger menu and you get stuck, then the quickest way to get unstuck is to reset the Simulator. Sometimes it just doesn't want to move to a new location even if you tell it to, and then you have to show it who's boss!

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 actually happen on my 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`" delegate method some more.

➤ Change `locationManager(didUpdateLocations)` to:

```
func locationManager(_ manager: CLLocationManager,
                     didUpdateLocations locations: [CLLocation]) {
  . . .

  if newLocation.horizontalAccuracy < 0 {
    return
  }

  // This section is new!
  var distance = CLLocationDistance(DBL_MAX)
  if let location = location {
    distance = newLocation.distance(from: location)
  }

  if location == nil ||
     location!.horizontalAccuracy > newLocation.horizontalAccuracy {

    . . .

    if newLocation.horizontalAccuracy <=
                                    locationManager.desiredAccuracy {
      print("*** We're done!")
      stopLocationManager()
      configureGetButton()

      // This statement is new!
      if distance > 0 {
        performingReverseGeocoding = false
      }
    }

    if !performingReverseGeocoding {
      . . .
    }

  // This section is new!
  } else if distance < 1 {
```

```
      let timeInterval = newLocation.timestamp.timeIntervalSince(
                                                        location!.timestamp)
      if timeInterval > 10 {
        print("*** Force done!")
        stopLocationManager()
        updateLabels()
        configureGetButton()
      }
    }
  }
```

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

```
  var distance = CLLocationDistance(DBL_MAX)
  if let location = location {
    distance = newLocation.distance(from: location)
  }
```

This calculates the distance between the new reading and the previous reading, if there was one. We can use this `distance` to measure if our location updates are still improving.

If there was no previous reading, then the distance is `DBL_MAX`. 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-statement after where you stop the location manager:

```
  if distance > 0 {
    performingReverseGeocoding = false
  }
```

This forces a reverse geocoding for the final location, even if the app is already currently performing another geocoding request.

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, this step would normally be skipped.

Simply by setting `performingReverseGeocoding` to `false`, you always force the geocoding to be done for this final coordinate.

(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.)

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

```
  } else if distance < 1 {
    let timeInterval = newLocation.timestamp.timeIntervalSince(
```

```
                                                    location!.timestamp)
    if timeInterval > 10 {
      print("*** Force done!")
      stopLocationManager()
      updateLabels()
      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 can stop fetching the location.

This is the improvement that was necessary to make my iPod touch stop scanning eventually. 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.

(By the way, did you notice you did `location!` to unwrap it before accessing the timestamp property? When the app gets inside this else-if, the value of `location` is guaranteed to be non-`nil`, so its safe to force unwrap the optional using the exclamation point.)

➤ 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 `print()`'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.

➤ First add a new instance variable:

```
  var timer: Timer?
```

➤ Then change `startLocationManager()` to:

```
  func startLocationManager() {
```

```
    if CLLocationManager.locationServicesEnabled() {
      . . .

      timer = Timer.scheduledTimer(timeInterval: 60, target: self,
          selector: #selector(didTimeOut), userInfo: nil, repeats: false)
    }
  }
```

The new lines set up a timer object that sends the "`didTimeOut`" message to `self` after 60 seconds; `didTimeOut` is the name of a method that you have to provide.

A *selector* is the term that Objective-C uses to describe the name of a method, and the `#selector()` syntax is how you make those selectors from Swift.

➤ Change `stopLocationManager()` to:

```
  func stopLocationManager() {
    if updatingLocation {
      . . .

      if let timer = timer {
        timer.invalidate()
      }
    }
  }
```

You have to cancel the timer in case the location manager is stopped before the time-out fires. This happens when an accurate enough location is found within one minute after starting, or when the user tapped the Stop button.

➤ Finally, add the `didTimeOut()` method:

```
  func didTimeOut() {
    print("*** Time out")

    if location == nil {
      stopLocationManager()

      lastLocationError = NSError(domain: "MyLocationsErrorDomain",
                                  code: 1, userInfo: nil)

      updateLabels()
      configureGetButton()
    }
  }
```

`didTimeOut()` 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.

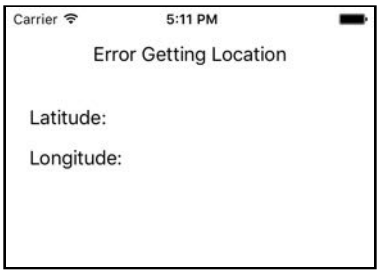
An error domain is simply a string, so `"MyLocationsErrorDomain"` will do. For the code I picked 1. The value of code doesn't really matter at this point because you only have one custom error, but you can imagine that when an app becomes bigger you'd have the need for multiple error codes.

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. In this case `updateLabels()` was already using an `NSError` anyway, so having your own error object just makes 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. (iOS development sometimes requires nerves of steel!)

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, or
- 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!

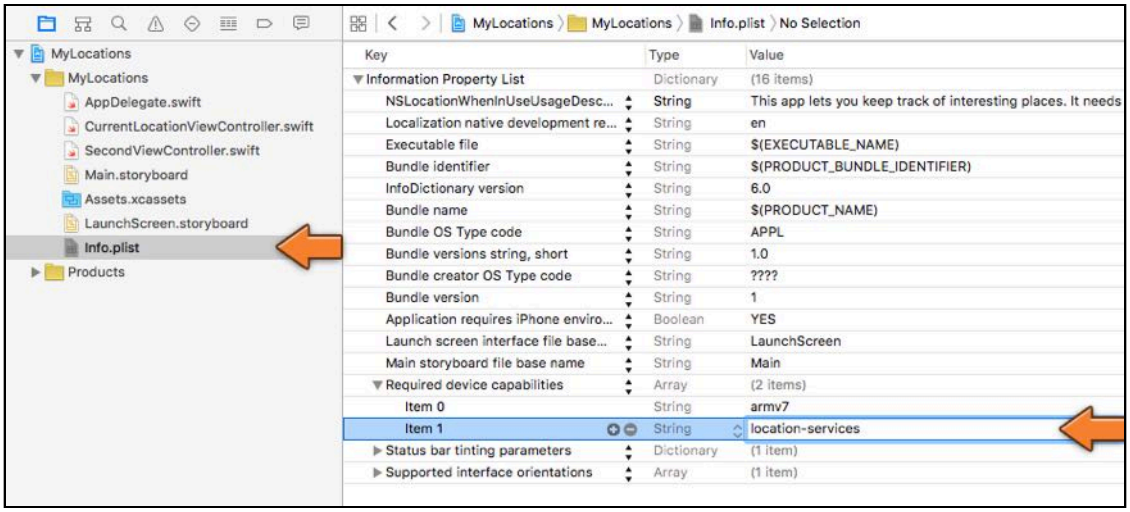
★ ★ ★

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 **Info.plist**:



Adding location-services to Info.plist

You could also add the item **gps**, in which case the app requires a GPS receiver. But 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 *App Programming Guide for iOS* on the Apple Developer website.

★ ★ ★

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

Supporting other iPhone models

So far you've been designing and testing the app for the iPhone SE's 4-inch screen.

As discussed previously, older iPhone models with smaller 3.5-inch screens are not supported by iOS 10, but your users may be running the app on the iPad and then it forces the app to use the 3.5-inch dimensions anyway.

Also, if you find work as a professional iOS developer you may need to support iOS 9 or even iOS 8, and that means it's still necessary for your apps to work on those smaller screens.

➤ To see what the app looks like on the iPhone 4s, go to **Main.storyboard** and use the **View as** panel at the bottom to switch to the smallest iPhone model.

The 'Get My Location' button is missing on the 3.5-inch screen

It's a good thing you tried this because the Get My Location button is no longer visible. This button was very close to the bottom of the 4-inch screen already, and it simply drops off the screen of smaller devices. You will have to move the button up a bit or end up with 1-star reviews in the App Store.

In the previous tutorials you've used Auto Layout to make the app's user interface resizable. You used the Pin and Align menus to create constraints that hold your views in place. This works well enough, but ask any iOS developer and they'll agree that it can be a bit of a hassle to manage all those constraints, especially as your UI designs grow more complex.

Fortunately, there's a handy shortcut that doesn't require you to make any constraints at all: **autoresizing**.

Before Auto Layout was available, autoresizing – also known as "springs & struts" – was the main tool for building resizable user interface layouts. It is simple to use but has its limitations. However, in many cases autoresizing is more than adequate.

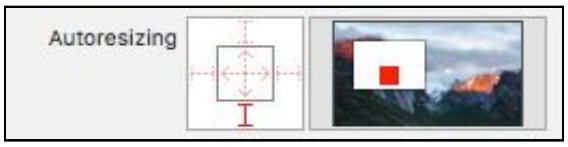
Here's how it works: each view has an autoresizing setting that determines what happens to the size and position of that view when the size of its superview – i.e. the view that contains it – changes. You can make the view stick to any of the four sides of its superview, and resize the view horizontally or vertically to fill up the space in the superview.

As of iOS 10 you can easily combine autoresizing with Auto Layout. Instead of making your own constraints, you simply set the autoresizing options for your views and UIKit will automatically make constraints for them.

Let's see how this works in practice. You will use autoresizing to keep the Get My Location button at a fixed distance from the bottom of the screen, no matter how large or small that screen is.

➤ First use the **View as** panel to switch back to the **iPhone SE**, so you can see the Get My Location button again.

➤ Select the Get My Location button and go to the **Size inspector**. Change the autoresizing options to the following:



The autoresizing 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).

➤ Use **View as** to switch to the **iPhone 4s** (or open the Preview pane in the Assistant editor). Now the button is visible on 3.5-inch devices as well.

If you find your Tag Location button is now too close to Get My Location, then move the Tag Location button up a bit in the storyboard. Around Y = 250 should be fine.

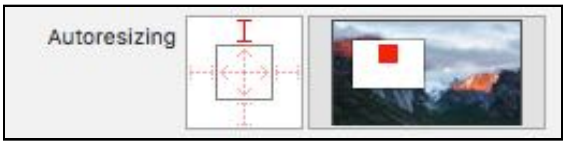
➤ Now use **View as** to view the app on the iPhone 6s Plus (or run the app in the iPhone 6s Plus or 7 Plus Simulator):

The app on the iPhone 6s Plus or 7 Plus Simulator

The labels are no longer aligned with the right edge of the screen and the Tag Location button isn't centered. That looks quite messy. Autoresizing to the rescue!

➤ Before you make the following changes, switch back to the **iPhone SE** in the storyboard.

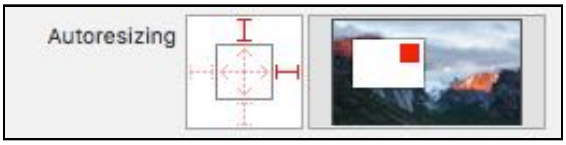
➤ For the **(Message label)** and the **Tag Location** button, change the autoresizing settings to:



The autoresizing settings for the message label and the button

Now the label and the button will always be centered horizontally in the main view.

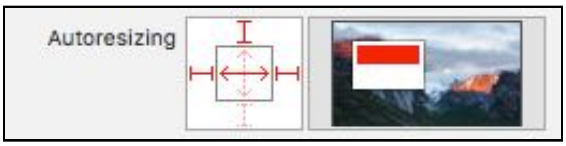
➤ For the **(Latitude goes here)** and **(Longitude goes here)** labels, change the autoresizing settings to:



The autoresizing settings for the coordinate labels

This keeps these two labels aligned with the right edge of the screen.

➤ Finally, for the **(Address goes here)** label, change the autoresizing settings to:



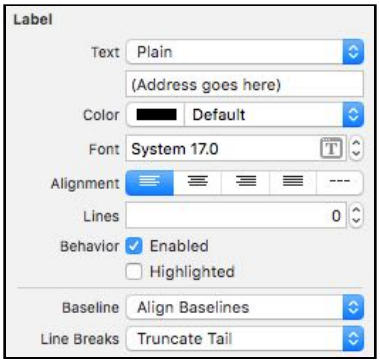
The autoresizing settings for the address label

This stretches the address label to be as wide as the screen allows. Now the app should look decent on the iPhone 6, 7, and Plus too. Try it out!

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

Attributes and properties

Most of the attributes in Interface Builder's inspectors correspond directly to properties on the selected object. For example, a `UILabel` has the following attributes:



These are directly related to the following properties:

```
Text          label.text
Color         label.textColor
Font          label.font
Alignment     label.textAlignment
Lines         label.numberOfLines
Enabled       label.isEnabled
Baseline      label.baselineAdjustment
Line Breaks   label.lineBreakMode
```

And so on… As you can see, the names may not always be exactly the same ("Lines" and `numberOfLines`) but you can easily figure out which property goes with which attribute.

You can find the properties in the documentation for `UILabel`. From the Xcode **Help** menu, select **Documentation and API Reference**. Type "uilabel" into the search field to bring up the class reference for `UILabel`:

The documentation for `UILabel` does not list properties for all of the attributes from the inspectors. For example, in the Attributes inspector there is a section named "View". The attributes in this section come from `UIView`, which is the base class of `UILabel`. So if you can't find a property in the `UILabel` class, you may need to check the documentation under "Inherits From".

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,

```
class ChecklistItem: NSObject {
  . . .
}
```

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

```
let item = ChecklistItem()
```

The `item` variable now contains an object of the class `ChecklistItem`. 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, "instance of class `ChecklistItem`" is the **type** of this `item` variable.

The Swift language and the iOS frameworks already come with a lot of types built-in but you can also add types of your own by making new 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:

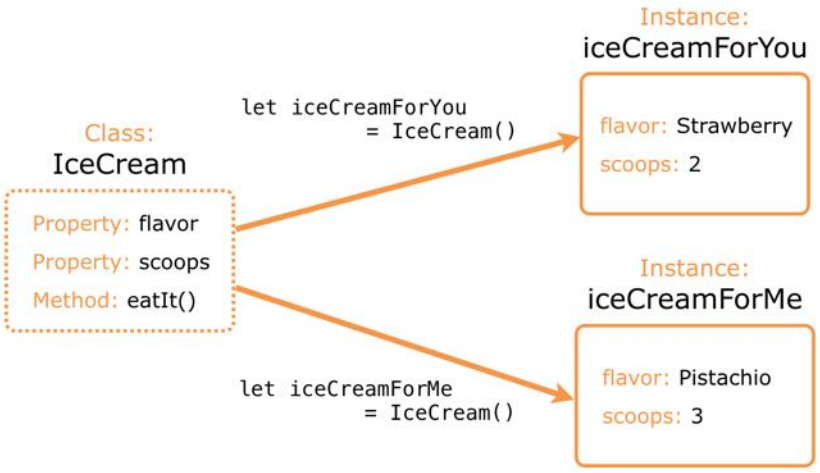
```
class IceCream: NSObject {
  var flavor: String
  var scoops: Int

  func eatIt() {
    // code goes in here
  }
}
```

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

```
// one for you
let iceCreamForYou = IceCream()
iceCreamForYou.flavor = "Strawberry"
iceCreamForYou.scoops = 2

// and one for me
let iceCreamForMe = IceCream()
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 declares: objects of this type have two properties, `flavor` and `scoops`, and a method named `eatIt()`.

Any new instance that is made from this template will have those instance variables and methods, but it lives in its own section of computer memory and therefore has its own values.

If you're more into architecture than food, 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

Sorry, 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 the iOS frameworks. You can see that in the `class` line that defines `IceCream`:

```
  class 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 the iOS frameworks. Most

objects that you'll encounter are made from a class that either directly inherits from `NSObject` or from another class that is ultimately based on `NSObject`. You can't escape it!

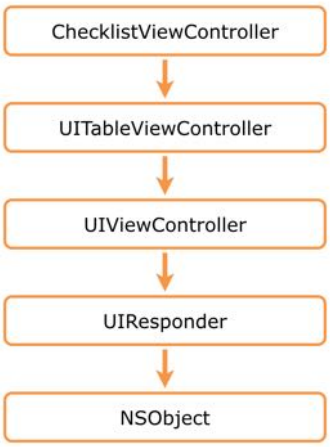
You've also seen class declarations that look like this:

```
class 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, it knows how to deal with prototype cells and static cells, and it 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.

`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 framework 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 an `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** or **to subclass**. These phrases all mean the same thing.

You don't want to write all your own screen and view handling code. If you'd have to program each screen totally from scratch, you'd still be working on lesson 1!

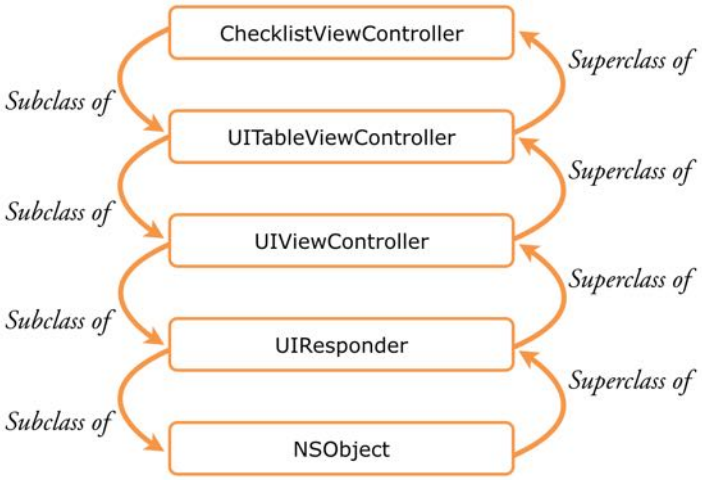
Thank goodness that stuff has been taken care of by very smart people working at Apple and they've bundled it into `UIViewController`. You simply make a class that inherits from `UIViewController` and you get all that functionality for free. You just add your own data and logic to that class and off 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 lets you re-use existing code with minimal effort. It can save you a lot of time!

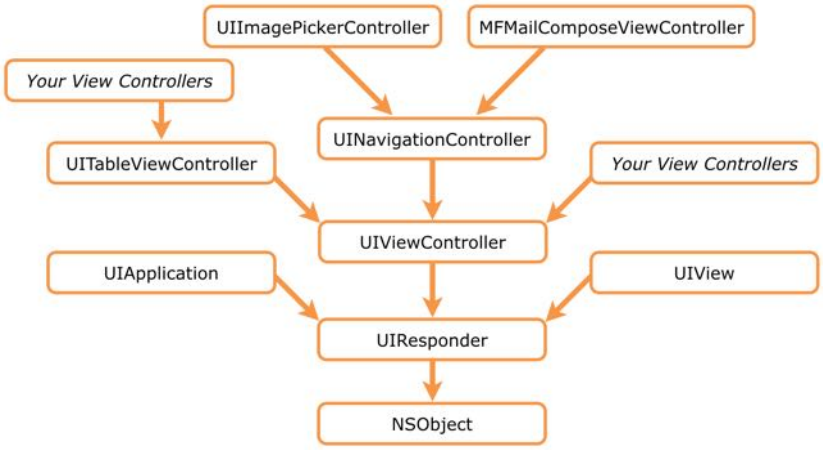
When programmers talk about inheritance, they'll often throw around the terms **superclass** and **subclass**.

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 Swift 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 you can 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.

Note: In Objective-C code, all your classes must at least inherit from the `NSObject` class. This is not the case with Swift. You could also have written the `IceCream` class as follows:

```
class IceCream {
  . . .
}
```

Now IceCream does not have a base class at all. This is fine in pure Swift code, but it won't always work if you try to use `IceCream` instances in combination with the iOS frameworks (which are written in Objective-C).

For example, you cannot use `IceCream` with `NSCoder` and `NSCoding` unless you also make it an `NSObject` first. So sometimes you'll have to use the `NSObject` base class, even if you're writing the app in Swift only.

Overriding methods

Inheriting from a class means your new class gets to use the properties and methods from its superclass.

If you create a new base class `Snack`,

```
class Snack {
  var flavor: String
  func eatIt() {
```

```
    // code goes in here
  }
}
```

and make `IceCream` inherit from that class,

```
class IceCream: Snack {
  var scoops: Int
}
```

then elsewhere in your code you can do:

```
let iceCreamForMe = IceCream()
iceCreamForMe.flavor = "Chocolate"
iceCreamForMe.scoops = 1
iceCreamForMe.eatIt()
```

This works even though `IceCream` did not explicitly declare an `eatIt()` method or `flavor` instance variable. But `Snack` does! Because `IceCream` inherits from `Snack`, it automatically gets this method and instance variable for free.

`IceCream` can also provide its own `eatIt()` method if it's important for your app that eating ice cream is different from eating any other kind of snack (you may want to eat it faster, before it melts):

```
class IceCream: Snack {
  var scoops: Int

  override func eatIt() {
    // code goes in here
  }
}
```

Now when someone calls `iceCreamForMe.eatIt()`, this new version of the method is invoked. Note that Swift requires you to write the `override` keyword in front of any methods that you provide that already exist in the superclass.

A possible implementation of this overridden version of `eatIt()` could look like this:

```
class IceCream: Snack {
  var scoops: Int
  var isMelted: Bool

  override func eatIt() {
    if isMelted {
      throwAway()
    } else {
      super.eatIt()
    }
  }
}
```

If the ice cream has melted, you want to throw it in the trashcan. But if it's still

edible, you'll call `Snack`'s version of `eatIt()` using `super`.

Just like `self` refers to the current object, the `super` keyword refers to the object's superclass. That is the reason you've been calling `super` in various places in your code, to let any superclasses do their thing.

Something that often happens in the iOS frameworks is that methods are 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 `viewWillAppear(true)`. 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:

```
class MyViewController: UIViewController {
  override func viewWillAppear(_ animated: Bool) {
    // do your own stuff before super

    // don't forget to call super!
    super.viewWillAppear(animated)

    // do your own stuff after super
  }
}
```

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.

You've also seen `override` already in the table view data source methods:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  . . .
}
```

`UITableViewController`, the superclass, already implements these methods, so if you want to provide your own implementation you need to override the existing ones.

Note: Inside those table view delegate and data source methods it's usually

not necessary to call `super`. The iOS API documentation can tell you whether you need to call `super` or not.

When making a subclass, the `init` methods require special care.

If you don't want to change any of the `init` methods from your superclass or add any new `init` methods, then it's easy: you don't have to do anything. The subclass will automatically take over the `init` methods from the superclass.

Most of the time, however, you will want to override an `init` method or add your own, for example to put values into the subclass's new instance variables. In that case, you may have to override not just that one `init` method but all of them.

In the next tutorial you'll create a class `GradientView` that extends `UIView`. The app from that tutorial uses `init(frame)` to create and initialize the `GradientView` object. `GradientView` overrides this method to set the background color:

```
class GradientView: UIView {
  override init(frame: CGRect) {
    super.init(frame: frame)
    backgroundColor = UIColor.black
  }
  required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
  }
  . . .
}
```

But because `UIView` also has another init method, `init?(coder)`, `GradientView` needs to implement that method too even if it doesn't do anything but call `super`.

Also note that `init(frame)` is marked as `override`, but `init?(coder)` is `required`. The `required` keyword is used to enforce that every subclass always implements this particular `init` method.

Swift wants to make sure that subclasses don't forget to add their own stuff to such required `init` methods, even if the app doesn't actually use that `init` method, as in the case of `GradientView` – it can be a bit of an over-concerned parent, that Swift.

The rules for inheritance of `init` methods are somewhat complicated – the official Swift Programming Guide devotes many pages to it – but at least if you make a mistake, Xcode will tell you what's wrong and what you should do to fix it.

Private parts

So… does a subclass get to use all the methods from its superclass? Not quite.

`UIViewController` and other UIKit classes have a lot more methods hidden away than you have access to. Often these secret methods do cool things, so it is tempting to use them. But they are not part of the official API, making them off-limits for mere mortals such as you and I.

If you ever hear other developers speak of "private APIs" in hushed tones and down dark alleys, then this is what they are talking about.

It is in theory possible to call such hidden methods if you know their names but this is not recommended. It may even get your app rejected from the App Store, as Apple is known to scan apps for usage of these private APIs.

You're not supposed to use private APIs for two reasons: 1) these APIs may have unexpected side effects and not be as robust as their publicly available relatives; 2) there is no guarantee these methods will exist from one version of iOS to the next. Using them is very risky, as your apps may suddenly stop working.

Sometimes, however, using a private API is the only way to access certain functionality on the device. If so, you're out of luck. Fortunately, for most apps the official public APIs are more than enough and you won't need to resort to the private stuff.

Casts

Often your code will refer to an instance not by its own class but by one of its superclasses. That probably sounds very weird, so let's look at an example.

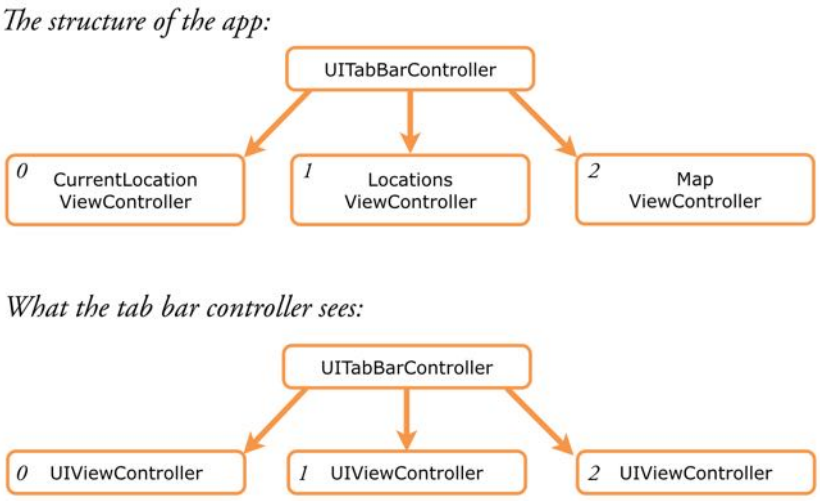
The app you're writing in this tutorial has a `UITabBarController` with three tabs, each of which is represented by a view controller. The view controller for the first tab is `CurrentLocationViewController`. Later in this tutorial you'll add two others, `LocationsViewController` for the second tab, and `MapViewController` for the third.

The designers of iOS obviously didn't know anything about those three particular 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 UITabBarController does not see your subclasses

The same thing goes for `UINavigationController`. To the navigation controller, any new view controllers that get pushed 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 reference to a "`UIViewController`" object, even though that is not the full type 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.

In the previous tutorial you did the following in `prepare(for, sender)`:

```
let navigationController = segue.destination as! UINavigationController

let controller = navigationController.topViewController
                                              as! ItemDetailViewController

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`. The value it returns is of the plain `UIViewController` type, which naturally doesn't have your `delegate` property.

If you were to do it without the "`as! ItemDetailViewController`" bit, like so,

```
let controller = navigationController.topViewController
```

then Xcode gives an error on the line below it. Swift now infers that the type of `controller` should also be `UIViewController`, but `UIViewController` does not have the `delegate` property. That property is something you only added to the subclass, `ItemDetailViewController`.

You know that `topViewController` refers to an `ItemDetailViewController`, but Swift doesn't. Even though all `ItemDetailViewControllers` are `UIViewControllers`, not all `UIViewControllers` are `ItemDetailViewControllers`!

Just because your friend Chuck has no hair, that doesn't mean all bald guys are named Chuck.

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

With the cast, the code looks like this:

```
let controller = navigationController.topViewController
                                              as! ItemDetailViewController
```

(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 `controller` as an `ItemDetailViewController` object. The compiler can'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.

Casts can fail for other reasons. For example, the value that you're trying to cast may actually be `nil`. If that's a possibility, it's a good idea to use the `as?` operator to make it an optional cast. You must also store the result of the cast into an optional value or use `if let` to safely unwrap it.

Note that a cast doesn't magically convert one type into another. You can't cast an `Int` into a `String`, 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.

Casting is very common in Swift programs because of the Objective-C heritage of the iOS frameworks. You'll be doing a lot of it!

To summarize, there are three kinds of casts you can perform:

as? for casts that are allowed to fail. This would happen if the object is `nil` or doesn't have a type that is compatible with the one you're trying to cast to. It will try to cast to the new type and if it fails, then no biggie. This cast returns an optional that you can unwrap with `if let`.

as! for casts between a class and one of its subclasses. This is also known as a

downcast. As with implicitly unwrapped optionals this cast is potentially unsafe and you should only use `as!` when you are certain it cannot possibly go wrong. You often need to use this cast when dealing with objects coming from UIKit and the other iOS frameworks. Better get used to all those exclamation marks!

as for casts that can never possibly fail. Swift can sometimes guarantee that a type cast will always work, for example between `NSString` and `String`. In that case you can leave off the `?` or the `!` and just write `as`.

It can sometimes be confusing to decide which of these three cast operators you need. If so, just type "as" and Xcode will suggest the correct variant. But nine times out of ten it will be `as!`.

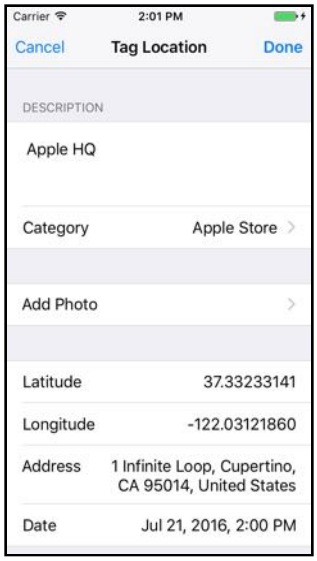
The Tag Location screen

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.

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 using the **Swift File** template. Name the file **LocationDetailsViewController**.

You know what's next: create outlets and connect them to the controls on the storyboard. 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.swift** with the following:

```
import UIKit

class LocationDetailsViewController: UITableViewController {
  @IBOutlet weak var descriptionTextView: UITextView!
  @IBOutlet weak var categoryLabel: UILabel!
  @IBOutlet weak var latitudeLabel: UILabel!
  @IBOutlet weak var longitudeLabel: UILabel!
  @IBOutlet weak var addressLabel: UILabel!
  @IBOutlet weak var dateLabel: UILabel!

  @IBAction func done() {
    dismiss(animated: true, completion: nil)
  }

  @IBAction func cancel() {
    dismiss(animated: true, completion: nil)
  }
}
```

Nothing special here, just a bunch of outlet properties and two action methods that

both dismiss the screen.

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

➤ In the **Identity inspector**, change the **Class** attribute of the table view controller to **LocationDetailsViewController** to link it with the source code file you just created.

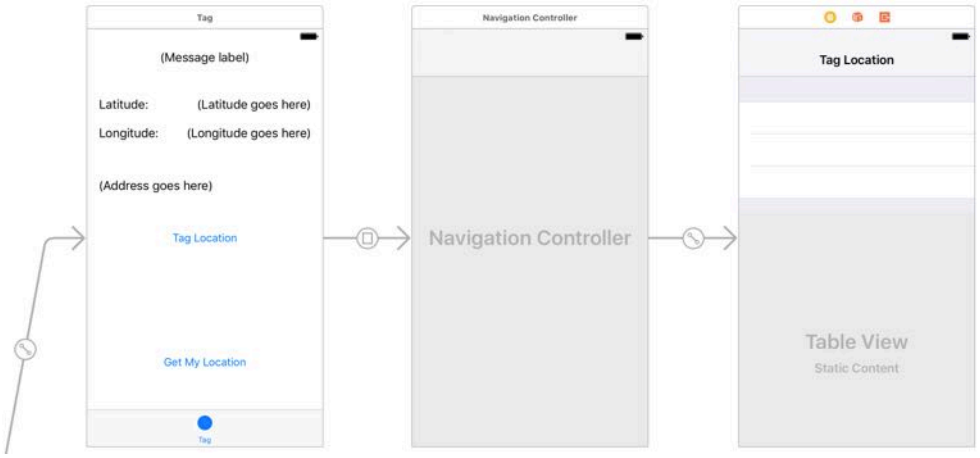
➤ With the new Table View Controller selected, choose **Editor → Embed In → Navigation Controller** from Xcode's menu bar to put it inside a new navigation controller.

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

➤ Double-click the Location Details View Controller's navigation bar roughly in the center, 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. (If you're using the Connections inspector, 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 **System Item** attributes to **Done**, and connect it to the **done** action.

➤ 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.

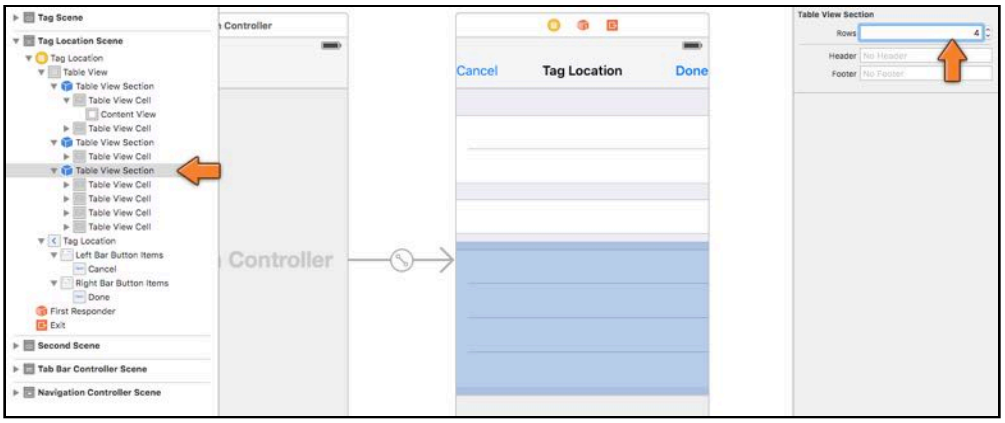
➤ Open the storyboard. Select the table view and go to the **Attributes** inspector. Change the **Sections** field from 1 to 3.

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. (If it won't delete, make sure you selected the whole Table View Cell and not its Content View.)

➤ Delete two cells from the middle section.

➤ Select the last Table View Section object (that is easiest in the outline pane on the left of the storyboard editor) and in the **Attributes inspector** set its **Rows** to 4.



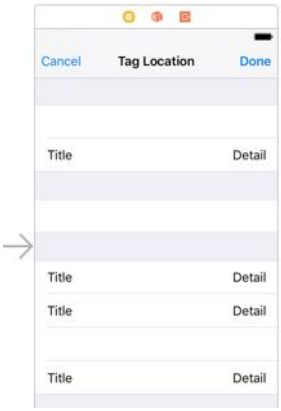
Adding a row to a table view section

(Alternatively you can drag a new Table View Cell from the Object Library into the

table.)

The second row from the first section, and the first, second 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 regular `UILabels`, so you can select them and change their properties.

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

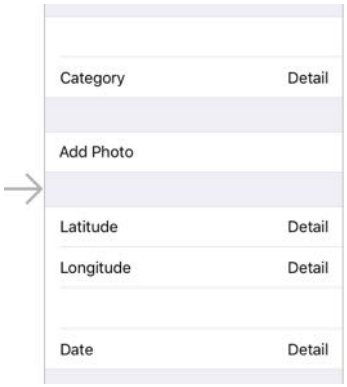
(If Xcode moves the label when you type into it or cuts off the text, then change the cell style to Left Detail and back again to Right Detail. That seems to fix it.)

➤ 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 cell so you'll design it yourself. Name this label **Add Photo**. (Later on you'll also add an image view to this cell.)

➤ Make sure the font of the label is **System**, size **17**, so it's the same size as the labels from the Right Detail cell style. If necessary, so use **Editor → Size to Fit Content** to resize the label to its optimal size.

➤ Put the Add Photo label at X: 15 (in the **Size inspector**) and vertically centered in its cell. You can use the **Editor → Align → Vertically in Container** menu option for this. (If this menu option is grayed out, deselect the label and select it again.)

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.
>
> 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 outline pane sidebar 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 cells.

➤ Select all the cells except Category and Add Photo. 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. Set the X position to 15, Y position to 11.

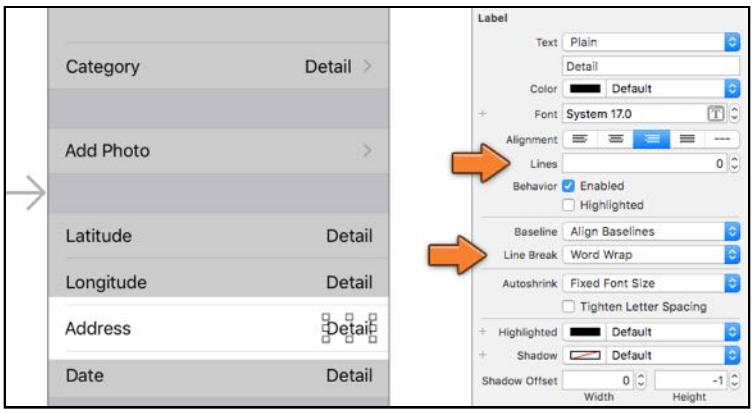
➤ Drag another **Label** into the same cell and name it **Detail**. Put it on the right, X position 261, Y position 11.

➤ Make sure the font of both labels is **System**, size **17**.

➤ Change the **Alignment** of the address detail label to right-aligned.

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.

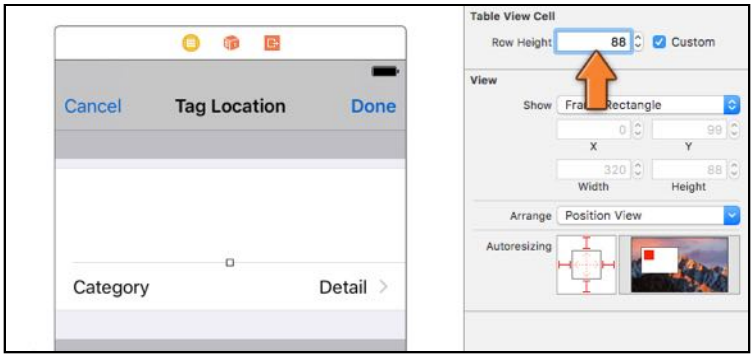
➤ In the **Attributes inspector** for the address detail label, set **Lines** to **0** and **Line Break** to **Word Wrap**. When the number of lines is 0, the label will resize vertically to fit all the text that you put into it, which is exactly what you need.



The address detail 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

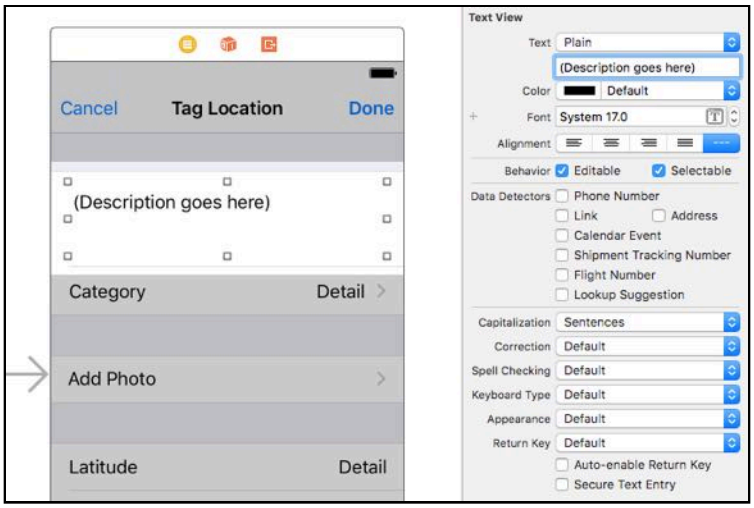
You can also drag the cell to this new height by the handle at its bottom, but I prefer to simply type in the new value.

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.

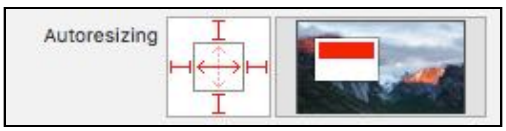
➤ By default, Interface Builder puts a whole bunch of fake Latin text (Lorem ipsum dolor, etc) into the text view. Replace that text with "(Description goes here)". The user will never see that text, but it's handy to remind yourself what this view is for.

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



The attributes for the text view

➤ With the text view selected, go to the **Size inspector**. Change the **Autoresizing** settings to the following:



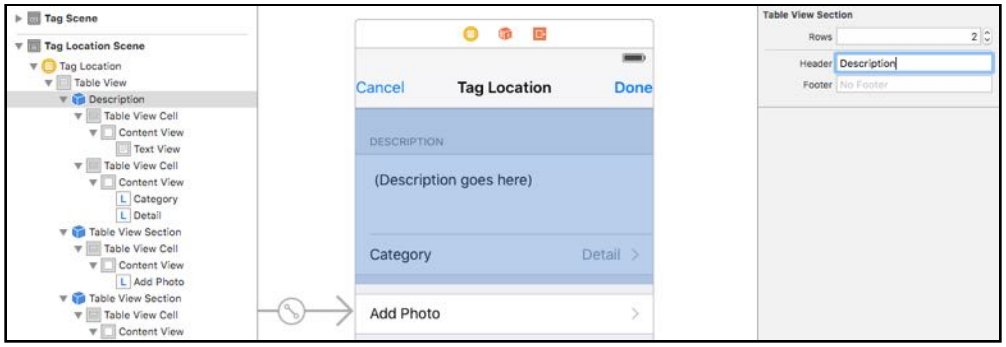
The autoresizing settings for the text view

With the "springs" enabled, the text view will automatically grow larger to fill up the extra space on the iPhone 6s, 7, and Plus devices.

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 the text view 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

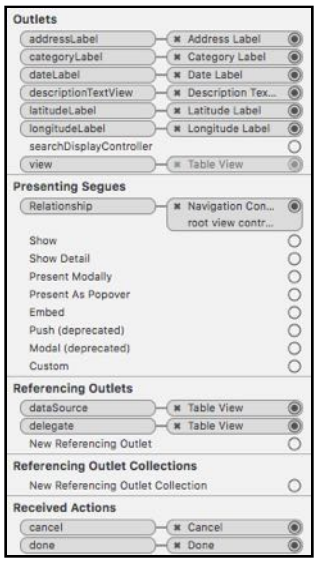
That's the layout done. The Tag Location screen should look like this in the storyboard:

The finished design of the Tag Location screen

Now you can actually make the screen do stuff.

➤ Connect the Detail labels and the text view to their respective outlets. It should be obvious which one goes where. (Tip: Ctrl-drag from the round yellow icon that represents the view controller to each of the labels. That's the quickest way.)

If you look at the **Connections inspector** for this view controller, 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 actual coordinates and address because you haven't given it any data yet.

Putting the location info into the screen

➤ Add two new properties to **LocationDetailsViewController.swift**:

```
var coordinate = CLLocationCoordinate2D(latitude: 0, longitude: 0)
var placemark: CLPlacemark?
```

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. This is an optional because there is no guarantee that the geocoder finds an address.

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. The `coordinate` is not an optional, so you must give it some initial value.

Exercise. Why is coordinate not an optional? ■

Answer: You cannot tap the Tag Location button unless GPS coordinates have been found, so you'll never open the `LocationDetailsViewController` without a valid set of coordinates.

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.

Xcode isn't happy with the two lines you just added. It complains about "Use of unresolved identifier CLLocationCoordinate2D" and "CLPlacemark". That means Xcode does not know anything about these types yet.

That's because they are part of the Core Location framework – and before you can use anything from a framework you first need to import it.

➤ Add to the top of the file, below the import for UIKit:

```
import CoreLocation
```

Now Xcode's error messages should disappear after a second or two. If they don't, type ⌘**+B** to build the app again.

Structs

I said `CLLocationCoordinate2D` was an object, but unlike the objects you've seen before, it is not described by a class. `CLLocationCoordinate2D` is a so-called **struct**.

Structs are like classes but a little less powerful. They can have properties and methods, but unlike classes they cannot inherit from one another.

The definition for `CLLocationCoordinate2D` is as follows:

```
struct CLLocationCoordinate2D {
  var latitude: CLLocationDegrees
  var longitude: CLLocationDegrees
}
```

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

```
typealias CLLocationDegrees = Double
```

The `Double` type 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:

```
struct CLLocationCoordinate2D {
  var latitude: Double
  var longitude: Double
}
```

The reason the designers of Core Location used `CLLocationDegrees` instead of `Double` is that "CL Location Degrees" tells you what this type is intended for: it stores the degrees of a location from the Core Location framework.

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.

UIKit and the other iOS frameworks also use structs regularly. Common examples are `CGPoint` and `CGRect`. In fact, `Array` and `Dictionary` are also structs.

Structs are more lightweight than classes. If you just need to pass around a set of values 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.

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

➤ Switch to **CurrentLocationViewController.swift** and add the *prepare-for-segue* method:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "TagLocation" {
    let navigationController = segue.destination
                                          as! UINavigationController
    let controller = navigationController.topViewController
                                       as! LocationDetailsViewController

    controller.coordinate = location!.coordinate
    controller.placemark = placemark
  }
}
```

You've seen how this works before. You use some casting magic to 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.

Because `location` is an optional you need to unwrap it before you can access its `coordinate` property. It's perfectly safe to force unwrap at this point because the Tag Location button that triggers the segue won't be visible unless a location is found. At this point, `location` will never be `nil`.

The `placemark` variable is also an optional, but so is the `placemark` property on `LocationDetailsViewController`, so you don't need to do anything special here. You can always assign the value of one optional to another optional without problems.

`viewDidLoad()` is a good place to display these things on the screen.

➤ Add the `viewDidLoad()` method to **LocationDetailsViewController.swift**:

```
override func viewDidLoad() {
  super.viewDidLoad()

  descriptionTextView.text = ""
  categoryLabel.text = ""

  latitudeLabel.text = String(format: "%.8f", coordinate.latitude)
  longitudeLabel.text = String(format: "%.8f", coordinate.longitude)

  if let placemark = placemark {
    addressLabel.text = string(from: placemark)
  } else {
    addressLabel.text = "No Address Found"
  }

  dateLabel.text = format(date: Date())
}
```

This simply puts something in every label. It uses two helper methods that you haven't defined yet: `string(from)` to format the `CLPlacemark` object into a string, and `format(date)` to do the same for a `Date` object.

➤ Add the `string(from)` method below `viewDidLoad()`:

```
func string(from placemark: CLPlacemark) -> String {
  var text = ""

  if let s = placemark.subThoroughfare {
    text += s + " "
  }
  if let s = placemark.thoroughfare {
    text += s + ", "
  }
  if let s = placemark.locality {
    text += s + ", "
  }
  if let s = placemark.administrativeArea {
    text += s + " "
  }
  if let s = placemark.postalCode {
    text += s + ", "
  }
  if let s = placemark.country {
    text += s
  }
  return text
}
```

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

To format the date you'll use a `DateFormatter` object. You've seen this class in the previous tutorial. It converts the date and time that are encapsulated by the `Date` object into a human-readable string, taking into account the user's language and locale settings.

In the previous tutorial you created a new instance of `DateFormatter` every time you wanted to convert a `Date` to a string. Unfortunately, `DateFormatter` 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 `DateFormatter` just once and then re-use that same object over and over. The trick is that you won't create the `DateFormatter` object until the app actually needs it. This principle is called **lazy loading** and it's a very important pattern for iOS apps. The work that you don't do won't cost any battery power.

In addition, you'll only ever create one instance of `DateFormatter`. The next time you need to use `DateFormatter` you won't make a new instance but re-use the existing one.

To pull this off you'll use a *private global* constant. That's a constant that lives outside of the `LocationDetailsViewController` class (global) but it is only visible inside the **LocationDetailsViewController.swift** file (private).

➤ Add the following to the top of **LocationDetailsViewController.swift**, in between the `import` and `class` lines:

```
private let dateFormatter: DateFormatter = {
  let formatter = DateFormatter()
  formatter.dateStyle = .medium
  formatter.timeStyle = .short
  return formatter
}()
```

What is going on here? You're creating a new constant named `dateFormatter` of type `DateFormatter`, that much should be obvious. This constant is `private` so it cannot be used outside of this Swift file.

You're also giving `dateFormatter` an initial value, but what follows the `=` is not an ordinary value – it looks like a bunch of source code in between `{ }` brackets. That's because it is a closure.

Normally you'd create a new object like this:

```
private let dateFormatter = DateFormatter()
```

But to initialize the date formatter it's not enough to just make an instance of `DateFormatter`, you also want to set the `dateStyle` and `timeStyle` properties of this instance.

To create the object and set its properties in one go, you can use a closure:

```
private let dateFormatter: DateFormatter = {
  // the code that sets up the DateFormatter object
  return formatter
}()
```

Inside the closure is the code that creates and initializes the new `DateFormatter` object, and then returns it. This returned value is what gets put into `dateFormatter`.

The knack to making this work is the `()` at the end. Closures are like functions, and to perform the code inside the closure you call it just like you'd call a function.

> **Note:** If you leave out the `()`, Swift thinks you're assigning the closure itself to `dateFormatter` – in other words, `dateFormatter` will contain a block of code, not an actual `DateFormatter` object. That's not what you want.
>
> Instead you want to assign the *result* of that closure to `dateFormatter`. To make that happen, you use the `()` to perform or **evaluate** the closure – this runs the code inside the closure and returns the `DateFormatter` object.

Using a closure to create and configure an object all at once is a nifty trick; you can expect to see this often in Swift programs.

In Swift, globals are always created in a lazy fashion, which means the code that creates and sets up this `DateFormatter` object isn't performed until the very first time the `dateFormatter` global is used in the app.

That happens inside the new `format(date)` method.

➤ Add this `format(date)` method. It goes inside the `class` again:

```
func format(date: Date) -> String {
  return dateFormatter.string(from: date)
}
```

How simple is that? It just asks the `DateFormatter` to turn the `Date` into a `String` and returns that.

Exercise. How can you verify that the date formatter is really only created once? ■

Answer: Add a `print()` just before the `return formatter` line in the closure. This `print()` text should appear only once in the Xcode debug pane.

➤ 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:

The Address label is too small to fit the entire address

The address seems to be missing something… only the first part of the address is visible (just the subthoroughfare or street number).

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 **LocationDetailsViewController.swift**:

```
// MARK: - UITableViewDelegate

override func tableView(_ tableView: UITableView,
                        heightForRowAt indexPath: IndexPath) -> CGFloat {
  if indexPath.section == 0 && indexPath.row == 0 {
    return 88

  } else if indexPath.section == 2 && indexPath.row == 2 {
    addressLabel.frame.size = CGSize(
                                  width: view.bounds.size.width - 115,
                                  height: 10000)
    addressLabel.sizeToFit()
    addressLabel.frame.origin.x = view.bounds.size.width -
                                      addressLabel.frame.size.width - 15
    return 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 if you wanted to change the height of all the cells at once (using the Row Height attribute in the storyboard or the `tableView.rowHeight` property).

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 `tableView(heightForRowAt)` correspond to those three situations. Let's take a look at the code for sizing the Address label:

```
// 1
addressLabel.frame.size = CGSize(width: view.bounds.size.width - 115,
                                 height: 10000)
// 2
addressLabel.sizeToFit()
```

```
  // 3
  addressLabel.frame.origin.x = view.bounds.size.width -
                                        addressLabel.frame.size.width - 15
  // 4
  return addressLabel.frame.size.height + 20
```

This uses a bit of trickery to resize the `UILabel` to make all its text fit 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.

The `frame` property is a `CGRect` that describes the position and size of a view.

`CGRect` is a struct that describes a rectangle. This rectangle has an origin made up of a `CGPoint` value with an (X, Y) coordinate, and a `CGSize` value for the width and height.

All `UIView` objects – and that includes subclasses such as `UILabel` – have a frame rectangle. Changing the `frame` property is how views are positioned on the screen.

Step-by-step this is what the code does:

1. Change the width of the label to be 115 points less than the width of the screen, which makes it about 200 points wide on the iPhone SE.

   Those 115 points that get subtracted account for the "Address" label on the left, the margins at the edges of the cell (15 points each), and some extra space between the two labels.

   This code also makes the frame a whopping 10,000 points high. That is done to make the rectangle tall enough to fit a lot of text.

   Because you're changing the `frame` property, the multi-line `UILabel` will now word-wrap the text to fit the requested width. This works because you already set the text on the label in `viewDidLoad()`.

2. 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()`.

3. The call to `sizeToFit()` removed any spare space to the right and bottom of the label. It may also have changed the width so that the text fits inside the label as snugly as possible, and because of that the X-position of the label may no longer be correct.

   A "detail" label like this should be placed against the right edge of the screen with a 15-point margin between them. That's done by changing the `frame`'s `origin.x` position.

4. 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.

> **Note:** 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. But it works, and that's the important thing!
>
> You may be wondering if this isn't one of those situations that Auto Layout can help with, and you'd be right. You can indeed use Auto Layout to automatically calculate the height of the address cell using so-called *self-sizing* table view cells.
>
> However, using multiline labels in Auto Layout is always a bit finicky. I find it easier to perform the calculations by hand. Besides, doing a little math never hurt anyone… ;-)

➤ Run the app. Now the reverse geocoded address should completely fit in the Address cell (even on models with larger screens such as the iPhone 6s and 7). Try it out with a few different locations.

The label resizes to fit the address

Frame vs. bounds

In the code above, you do the following:

```
addressLabel.frame.size = CGSize(
                            width: view.bounds.size.width - 115,
                            height: 10000)
```

You use the view's `bounds` to calculate the address label's `frame`. Both frame and bounds are of type `CGRect`, which describes a rectangle. So what is the difference between the bounds and the frame?

The frame describes the position and size of a view in its parent view. If you want to put a 150×50 label at position X: 100, Y: 30, then its frame is (100, 30, 150, 50). To move a view from one position to another, you change its `frame` property (or its `center` property, which in turn will modify the frame).

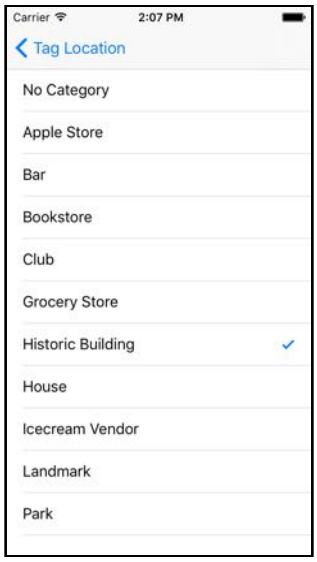
Where the frame describes the outside of the view, the bounds describe the inside. The X and Y coordinates of the bounds are (0, 0) and the width and height will be the same as the frame. So for the example, the bounds are (0, 0, 150, 50). It's a matter of perspective.

Sometimes it makes sense to use the bounds; sometimes you need to use the frame. The frame is actually derived from a combination of properties: the `center` position of the view, the `bounds`, and any `transform` that is set on the view. (Transforms are used for rotating or scaling the view.)

When you set Auto Layout constraints on a view, those constraints are used to calculate the view's frame. If a view has constraints, you shouldn't change the `frame` or `bounds` properties yourself, or it will conflict with Auto Layout and the results may be unpredictable.

The category picker

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



The category picker

This is a new screen, so you also need a new view controller. The way this 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.

➤ Add a new file to the project named **CategoryPickerViewController.swift**.

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

```
import UIKit

class CategoryPickerViewController: UITableViewController {
  var selectedCategoryName = ""

  let categories = [
    "No Category",
    "Apple Store",
    "Bar",
    "Bookstore",
    "Club",
    "Grocery Store",
    "Historic Building",
    "House",
    "Icecream Vendor",
    "Landmark",
    "Park"]

  var selectedIndexPath = IndexPath()

  override func viewDidLoad() {
    super.viewDidLoad()

    for i in 0..<categories.count {
      if categories[i] == selectedCategoryName {
        selectedIndexPath = IndexPath(row: i, section: 0)
        break
      }
    }
  }

  // MARK: - UITableViewDataSource

  override func tableView(_ tableView: UITableView,
                          numberOfRowsInSection section: Int) -> Int {
    return categories.count
  }

  override func tableView(_ tableView: UITableView,
                 cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
                                withIdentifier: "Cell", for: indexPath)

    let categoryName = categories[indexPath.row]
    cell.textLabel!.text = categoryName

    if categoryName == selectedCategoryName {
      cell.accessoryType = .checkmark
    } else {
      cell.accessoryType = .none
    }
    return cell
  }

  // MARK: - UITableViewDelegate
```

```
    override func tableView(_ tableView: UITableView,
                            didSelectRowAt indexPath: IndexPath) {
      if indexPath.row != selectedIndexPath.row {
        if let newCell = tableView.cellForRow(at: indexPath) {
          newCell.accessoryType = .checkmark
        }
        if let oldCell = tableView.cellForRow(at: selectedIndexPath) {
          oldCell.accessoryType = .none
        }
        selectedIndexPath = indexPath
      }
    }
  }
```

There's nothing special going on here. This is a table view controller that shows a list of category names. It has table view data source and delegate methods. The data source gets its rows from the `categories` array.

The only thing worth noting is the `selectedIndexPath` instance variable. When the screen opens it shows a checkmark next to the currently selected category. This comes from the `selectedCategoryName` property, which is filled in when you segue to this screen.

When the user taps a 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. You can't use `selectedCategoryName` for this because that is a string, not a row number. Therefore, you first need to find the row number – or index-path – for the selected category name.

That happens in `viewDidLoad()`. You loop through the array of categories and compare the name of each category to `selectedCategoryName`. If they match, you create an index-path object and store it in the `selectedIndexPath` variable. Once a match is found, you can `break` out of the loop because there's no point in looking at any of the other categories.

Now that you know the row number, you can remove the checkmark for this row in `tableView(didSelectRowAt)` when another row gets tapped.

It's a bit of work for such a small feature, but in a good app it's the details that matter.

There are several different ways of looping through the contents of an array.

You've already seen `for in`, which is used as follows:

```
for category in categories { . . .
```

This puts the name of each category into the temporary constant named `category`.

However, in order to make the index-path object you don't want the name of the category but the index of that category in the array. So you'll have to loop in a slightly different fashion:

```
for i in 0..<categories.count {
  let category = categories[i]
  . . .
}
```

Thanks to the half-open range operator `..<`, `i` is a number that increments from 0 to `categories.count - 1`. This is a very common pattern for looping through an array if you want to have the index as well.

Another way to do this is to use the `enumerated()` method, of which you'll see an example in the next tutorial. As a quick preview, this is how you'd use it:

```
for (i, category) in categories.enumerated() {
  . . .
}
```

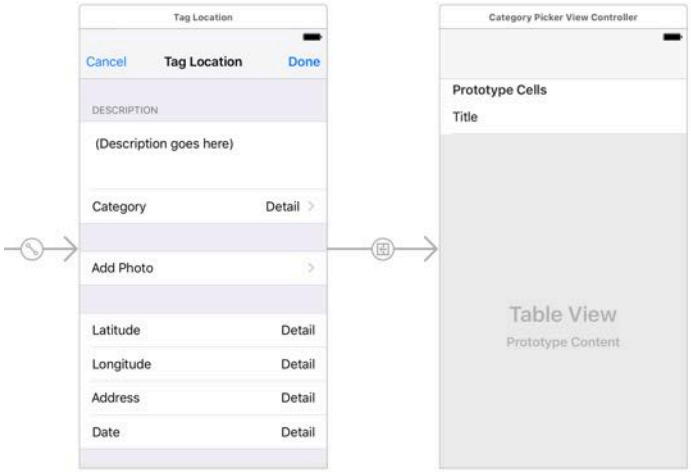
➤ Open the storyboard and drag a new **Table View Controller** into the canvas. Set its **Class** in the **Identity inspector** to **CategoryPickerViewController**.

➤ 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 choose **Selection Segue - Show**.

➤ 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 Apple recommends that you do not 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.

➤ Switch back to **LocationDetailsViewController.swift** and add a new instance variable `categoryName`. You'll use this to temporarily store the chosen category.

```
var categoryName = "No Category"
```

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:

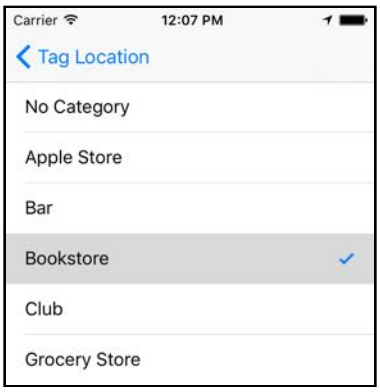
```
override func viewDidLoad() {
  super.viewDidLoad()

  descriptionTextView.text = ""
  categoryLabel.text = categoryName      // change this
  . . .
```

➤ Finally, add the `prepare(for:sender:)` method:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "PickCategory" {
    let controller = segue.destination as! CategoryPickerViewController
    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.



Selecting a new category

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 handy storyboarding feature that can accomplish the same thing with less work: **unwind segues**.

In case you were wondering what the orange "Exit" icons in the storyboard are for, you now have your answer: unwind segues.

The Exit icon

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 orange 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.swift**, add the following method:

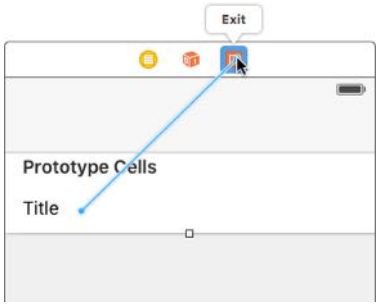
```
@IBAction func categoryPickerDidPickCategory(_ segue: UIStoryboardSegue)
{
  let controller = segue.source as! CategoryPickerViewController
  categoryName = controller.selectedCategoryName
  categoryLabel.text = categoryName
}
```

This is an action method because it has the `@IBAction` annotation. What's different from a regular action method is the parameter, a `UIStoryboardSegue` object.

Normally if an action method has a parameter, it points to the control that triggered the action, such as a button or slider. But in order to make an unwind segue you need to define an action method that takes a `UIStoryboardSegue` parameter.

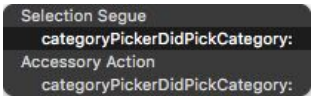
What happens inside the method is pretty straightforward. You look at the view controller that sent the segue (the `source`), which of course is the `CategoryPickerViewController`, and then read the value of its `selectedCategoryName property`. That property contains the category that the user picked.

➤ Open the storyboard. **Ctrl-drag** from the prototype cell to the Exit button. This time it allows you to make a connection:



Ctrl-dragging to the Exit icon to make an unwind segue

From the popup choose **Selection Segue - categoryPickerDidPickCategory:**, the name of the unwind action method you just added.



The popup lists the unwind action methods

(If Interface Builder doesn't let you make a connection, then make sure you're really Ctrl-dragging from the Cell, not from its Content View or the label.)

Now when you tap a cell in the category picker, the screen will close and this new method is called.

➤ Run the app to try it out. That was easy! Well, not quite. 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? `prepare(for:sender:)`, of course! This works for segues in both directions.

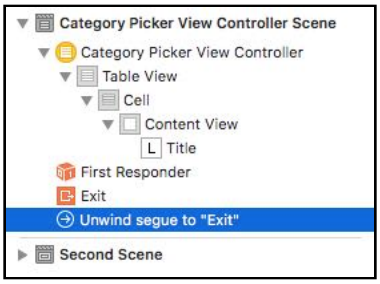
➤ Add the following method to **CategoryPickerViewController.swift**:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "PickedCategory" {
    let cell = sender as! UITableViewCell
    if let indexPath = tableView.indexPath(for: cell) {
      selectedCategoryName = categories[indexPath.row]
    }
  }
}
```

This looks at the selected index-path and puts the corresponding category name into the `selectedCategoryName` property.

This logic assumes the unwind segue is named "PickedCategory", so you still have to set an identifier on the unwind segue.

Unfortunately, there is no visual representation of that 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**.

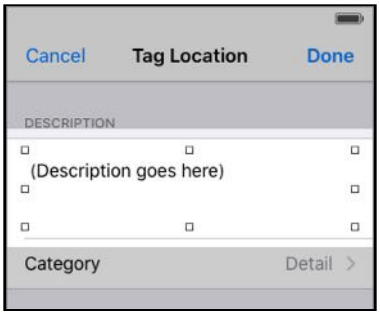
➤ Run the app. Now the category picker should work properly. As soon as you tap the name of a category, the screen closes and the new category name is displayed.

Unwind segues are pretty cool and are often easier than using a delegate protocol, especially for simple picker screens such as this one.

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 she's actually tapping outside the text view, and she 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 text view should activate, even if the tap wasn't on the text view itself.

➤ Add the following two methods in the `// MARK: - UITableViewDelegate` section in **LocationDetailsViewController.swift**:

```
override func tableView(_ tableView: UITableView,
                        willSelectRowAt indexPath: IndexPath) -> IndexPath? {
  if indexPath.section == 0 || indexPath.section == 1 {
    return indexPath
  } else {
    return nil
  }
}

override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  if indexPath.section == 0 && indexPath.row == 0 {
    descriptionTextView.becomeFirstResponder()
  }
```

```
  }
```

The `tableView(willSelectRowAt)` method limits taps to just the cells from the first two sections. Recall that `||` means "or", so if the section number equals 0 *or* when it equals 1, you accept the tap on the cell. The third section only has read-only labels anyway, so it doesn't need to allow taps.

The `tableView(didSelectRowAt)` method 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 you will give the input focus to that text view. Here you use `&&`, meaning "and", to make sure the tap was in the first section *and* also on the first row of that section.

➤ Try it out. Run the app and click or tap somewhere along the edges of the first cell. Anywhere you tap inside that first cell should now make the text view active and bring up the keyboard (on the Simulator you may need to press **⌘+K** to make the keyboard visible).

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. And 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()`:

```
  let gestureRecognizer = UITapGestureRecognizer(target: self,
                                                 action: #selector(hideKeyboard))
  gestureRecognizer.cancelsTouchesInView = false
  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:

```
  . . . target: self, action: #selector(hideKeyboard)) . .
```

You use this syntax to tell the `UITapGestureRecognizer` that it should call the

method named by the `#selector()` whenever the gesture happens.

This pattern is known as **target-action** and you've already 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 a method to respond to that message.

➤ Add the `hideKeyboard()` method to **LocationDetailsViewController.swift**. It doesn't really matter where you put it, but below `viewDidLoad()` is a good place:

```
func hideKeyboard(_ gestureRecognizer: UIGestureRecognizer) {
  let point = gestureRecognizer.location(in: tableView)
  let indexPath = tableView.indexPathForRow(at: point)

  if indexPath != nil && indexPath!.section == 0
                      && indexPath!.row == 0 {
    return
  }

  descriptionTextView.resignFirstResponder()
}
```

Whenever the user taps somewhere in the table view, the gesture recognizer calls this method. Conveniently, it also passes a reference to itself as the parameter, which lets you ask `gestureRecognizer` where the tap happened.

The `gestureRecognizer.location(in: tableView)` method returns a `CGPoint` value. `CGPoint` is a 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 text view! If the user tapped anywhere else, you do hide the keyboard.

Exercise. Does the logic in the if-statement make sense to you? Explain how this works. ■

Answer: It is possible that the user taps inside the table view but not on a cell, for example somewhere in between two sections or on the section header. In that case `indexPath` will be `nil`, making this an optional (of type `IndexPath?`). And to use an optional you need to unwrap it somehow, either with `if let` or with `!`.

You only want to hide the keyboard if the index-path for the tap is not section 0, row 0, which is the cell with the text view. If the user did tap that particular cell, you bail out of `hideKeyboard()` with the `return` statement before the code reaches

the call to `resignFirstResponder()`.

> **Note:** You don't want to force unwrap an optional if there's a chance it might be `nil` or you risk crashing the app. Force unwrapping `indexPath!.section` and `indexPath!.row` may look dangerous here, but it is guaranteed to work thanks to the **short-circuiting** behavior of the `&&` operator.
>
> If `indexPath` equals `nil`, then everything behind the first `&&` is simply ignored. The condition can never become true anymore if one of the terms is false. So when the app gets to look at `indexPath!.section`, you know that the value of `indexPath` will never be `nil` at that point.

An alternative way to write this logic is:

```
if indexPath == nil ||
          !(indexPath!.section == 0 && indexPath!.row == 0) {
  descriptionTextView.resignFirstResponder()
}
```

Can you wrap your head around that? Here the if-statement checks for the exact opposite. The `&&` and `||` operators are each other's opposite in Boolean logic and you can often flip the meaning of a condition around by turning `&&` into `||` and introducing the `!` not operator.

You don't need to worry about this so early on in your programming career, but at some point you'll have to learn these rules of Boolean logic. They can be mind-benders!

Of course, you can also use `if let` to safely unwrap `indexPath`. So a third way to write this if-statement is as follows:

```
if let indexPath = indexPath, indexPath.section != 0 &&
                              indexPath.row != 0 {
  return
}
descriptionTextView.resignFirstResponder()
```

I just wanted to give you a brief glimpse of the various ways you can write the conditions in if-statements. There's often more than one way to do something in Swift, so choose whatever approach you find easiest to understand.

➤ Run the app. Tap in the text view to bring up the keyboard. (If the keyboard doesn't come up, press **⌘+K**.) Tap anywhere else in the table view to hide the keyboard again.

The table view can also automatically dismiss the keyboard when the user starts scrolling. You can enable this in the storyboard.

➤ Open the storyboard and select the table view inside the Tag Location screen. In the **Attributes inspector** change the **Keyboard** option to **Dismiss on drag**. Now

scrolling should also hide the keyboard.

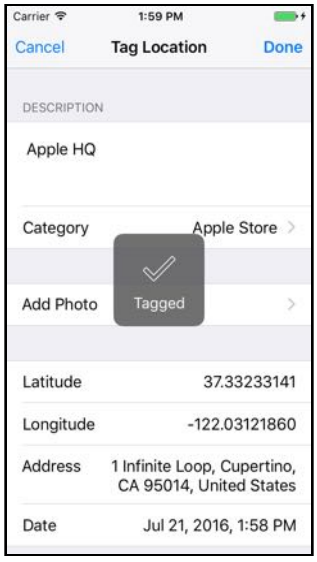
The "Dismiss on drag" option for the keyboard

(If this doesn't work for you, try it on a real device. The keyboard in the Simulator can be a bit wonky.)

➤ Also try the **Dismiss interactively** option. Which one do you like best?

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 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, in fact.

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.

➤ Add a new file to the project using the **Swift File** template. Name it **HudView**.

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.swift** with the following:

```
import UIKit

class HudView: UIView {
  var text = ""

  class func hud(inView view: UIView, animated: Bool) -> HudView {
    let hudView = HudView(frame: view.bounds)
    hudView.isOpaque = false

    view.addSubview(hudView)
    view.isUserInteractionEnabled = false

    hudView.backgroundColor = UIColor(red: 1, green: 0, blue: 0,
                                      alpha: 0.5)
    return hudView
  }
}
```

The `hud(inView, animated)` method is known as a **convenience constructor**. It creates and returns a new `HudView` instance.

Normally you would create a new `HudView` object by writing:

```
let hudView = HudView()
```

But using the convenience constructor you'd write:

```
let hudView = HudView.hud(inView: parentView, animated: true)
```

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

declaration begins with `class func` instead of just `func`.

When you call `HudView.hud(inView: parentView, animated: true)` you don't have an instance of `HudView` yet. 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:

```
class func hud(inView view: UIView, animated: Bool) -> HudView {
  let hudView = HudView(frame: view.bounds)

  . . .

  return hudView
}
```

It calls `HudView()`, or actually `HudView(frame)` which is an `init` method inherited from `UIView`. 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 "parent" view object. This is the navigation controller's view so the HUD will cover the entire screen.

It also sets view's `isUserInteractionEnabled` property to `false`. 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 `isUserInteractionEnabled` to `false` 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.

Let's add code to call this funky new HUD, so that is out of the way.

➤ Change the `done()` method from **LocationDetailsViewController.swift** to:

```
@IBAction func done() {
  let hudView = HudView.hud(inView: navigationController!.view,
                                    animated: true)
  hudView.text = "Tagged"
}
```

This creates a `HudView` object and adds it to the navigation controller's view with an animation. You also set the `text` property on the new object.

Previously, `done()`dismissed the view controller. For testing purposes you're not going to do that anymore. You want to have enough time to see what the `HudView` looks like as you're building it up step-by-step; if you immediately close the screen after showing the HUD, it will be hard to see what's going on (unless you have the ability to slow down time somehow). You'll put the code that closes the screen back later.

➤ Run the app. When you press the Done button, the screen will look like this:

The HUD view covers the whole screen

The app is now totally unresponsive because user interaction is disabled.

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 `hud(inView: animated)` method.

➤ Add the following method to **HudView.swift**:

```
override func draw(_ rect: CGRect) {
  let boxWidth: CGFloat = 96
  let boxHeight: CGFloat = 96

  let boxRect = CGRect(
    x: round((bounds.size.width - boxWidth) / 2),
    y: round((bounds.size.height - boxHeight) / 2),
    width: boxWidth,
    height: boxHeight)

  let roundedRect = UIBezierPath(roundedRect: boxRect, cornerRadius: 10)
  UIColor(white: 0.3, alpha: 0.8).setFill()
  roundedRect.fill()
}
```

The `draw()` method is invoked whenever UIKit wants your view to redraw itself.

Recall that everything in iOS is event-driven. The view doesn't draw anything on the screen unless UIKit sends it the `draw()` event. That means you should never call

`draw()` yourself.

Instead, if you want view to redraw, you should send it the `setNeedsDisplay()` message. UIKit will then trigger a `draw()` 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 redrawing the screen whenever you feel like it, but on 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):

```
let boxWidth: CGFloat = 96
let boxHeight: CGFloat = 96
```

This declares two constants you'll be using in the calculations that follow. You're using constants because it's clearer to refer to the symbolic name `boxWidth` than the number 96. That number doesn't mean much by itself, but "box width" is a pretty clear description of its purpose.

Note that you force the type of these constants to be `CGFloat`, which is the type used by UIKit to represent decimal numbers. When working with UIKit or Core Graphics (CG, get it?) you use `CGFloat` instead of the regular `Float` or `Double`.

```
let boxRect = CGRect(
  x: round((bounds.size.width - boxWidth) / 2),
  y: round((bounds.size.height - boxHeight) / 2),
  width: boxWidth,
  height: boxHeight)
```

There is `CGRect` again, the struct that represents a rectangle. You use it to calculate the position for the HUD. The HUD rectangle should be centered horizontally and vertically on the screen. The size of the screen is given by `bounds.size` (this really is the size of `HudView` itself, which spans the entire screen).

The above calculation uses the `round()` function to make sure the rectangle doesn't end up on fractional pixel boundaries because that makes the image look fuzzy.

```
let roundedRect = UIBezierPath(roundedRect: boxRect, cornerRadius: 10)
UIColor(white: 0.3, alpha: 0.8).setFill()
roundedRect.fill()
```

`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.

➤ Run the app. 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@2x.png** and **Checkmark@3x.png**. Add these files to the asset catalog, **Assets.xcassets**.

You can do this with the + button or simply drag them from Finder into the Xcode window with the asset catalog open.

➤ Add the following code to the bottom of `draw()`:

```
if let image = UIImage(named: "Checkmark") {
  let imagePoint = CGPoint(
    x: center.x - round(image.size.width / 2),
    y: center.y - round(image.size.height / 2) - boxHeight / 8)

  image.draw(at: 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 (`center`) and the dimensions of the image (`image.size`).

Finally, it draws the image at that position.

➤ Run the app to see the HUD view with the image:

The HUD view with the checkmark image

Failable initializers

To create the `UIImage` you used `if let` to unwrap the resulting object. That's because `UIImage(named)` is a so-called *failable* initializer.

It is possible that loading the image fails, because there is no image with the specified name or the file doesn't really contain a valid image. You can't fool `UIImage` into loading something that isn't an image!

That's why `UIImage`'s `init(named)` method is really defined as `init?(named)`. The question mark indicates that this method returns an optional. If there was a problem loading the image, it returns `nil` instead of a brand spanking new `UIImage` object.

You'll see these failable initializers through the iOS frameworks. One that you have encountered before is `init?(coder)`. Whenever it is possible that creating a new object will fail, the responsible `init` method will return an optional that you need to unwrap before you can use it.

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 `draw()` by adding the following code:

```
let attribs = [ NSFontAttributeName: UIFont.systemFont(ofSize: 16),
                NSForegroundColorAttributeName: UIColor.white ]

let textSize = text.size(attributes: attribs)

let textPoint = CGPoint(
  x: center.x - round(textSize.width / 2),
  y: center.y - round(textSize.height / 2) + boxHeight / 4)

text.draw(at: textPoint, withAttributes: attribs)
```

When drawing text you first need to know how big the text is, so you can figure out where to position it. `String` 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. As of iOS 9, the system font is San Francisco (on iOS 8 and before it was Helvetica Neue, which looks very similar).

You also choose a color for the text, plain white. The font and foreground color are placed into a dictionary named `attribs` (short for "attributes").

Recall that a dictionary stores key-value pairs. To make a dictionary you write:

```
let myDictionary = [ key1: value1,
                     key2: value2,
                     key3: value3 ]
```

So in the dictionary from `draw()`, the `NSFontAttributeName` key is associated with the `UIFont` object, and the `NSForegroundColorAttributeName` key is associated with the `UIColor` object. In other words, the `attribs` dictionary describes what the text will look like.

You use these attributes and the string from the `text` property to calculate how wide and tall the text will be. The result ends up in the `textSize` constant, which is of type `CGSize`. (As you can tell, `CGPoint`, `CGSize`, and `CGRect` are types you use a lot when making your own views.)

Finally, you calculate where to draw the text (`textPoint`), and then draw it. Quite simple, really.

➤ Run the app to try it out. Lookin' good!

The HUD view with the checkmark and the text

➤ Make sure to test the HUD on the different Simulators. No matter the device dimensions, the HUD should always appear centered in the screen.

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 `show(animated)` method to **HudView.swift**:

```
func show(animated: Bool) {
  if animated {
    // 1
```

```
      alpha = 0
      transform = CGAffineTransform(scaleX: 1.3, y: 1.3)
      // 2
      UIView.animate(withDuration: 0.3, animations: {
        // 3
        self.alpha = 1
        self.transform = CGAffineTransform.identity
      })
    }
  }
```

In the Bull's Eye tutorial 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, making the view 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.animate(withDuration: . . .)` to set up an animation. You give this a closure that describes the animation. Recall that a closure is a piece of inline code that is not executed right away. UIKit will animate the properties that you change inside the closure from their initial state to the final state.

3. Inside the closure, set up the new state of the view that it should have after the animation completes. You set `alpha` to 1, which means the `HudView` is now fully opaque. You also set the `transform` to the "identity" transform, restoring the scale back to normal. Because this code is part of a closure, you need to use `self` to refer to the `HudView` instance and its properties. That's the rule for closures.

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 `hud(inView, animated)` method to call `show(animated)` just before it returns:

```
  class func hud(inView view: UIView, animated: Bool) -> HudView {
    . . .
    hudView.show(animated: animated)
    return hudView
  }
```

➤ Run the app and marvel at the magic of `UIView` animation.

You can actually do one better. iOS has something called "spring" animations, which

bounce up and down and are much more visually interesting than the plain old animations. Using them is very simple.

➤ Replace the `UIView.animate(withDuration: . . .)` code with the following:

```
UIView.animate(withDuration: 0.3, delay: 0, usingSpringWithDamping: 0.7,
  initialSpringVelocity: 0.5, options: [], animations: {
    self.alpha = 1
    self.transform = CGAffineTransform.identity
  },
  completion: nil)
```

The code in the closure is still the same – it sets alpha to 1 and restores the identity transform – but this new animation method has a lot more options. Feel free to play with these options to see what they do.

➤ Run the app and watch it bounce. Actually, the effect is very subtle, but subtle is good when it comes to user interfaces. You don't want your users to get seasick from using the app!

Back to **LocationDetailsViewController**… You still need to close the screen when the user taps Done.

There's a challenge here: you don't want to dismiss the screen right away. It won't look very good if the screen already closes before the HUD is finished animating. You didn't spend all that time writing `HudView` for nothing – you want to give your users a chance to see it.

➤ Add a new import at the top of **LocationDetailsViewController.swift**:

```
import Dispatch
```

This imports the Grand Central Dispatch framework, or GCD for short. GCD is a very handy but somewhat low-level library for handling asynchronous tasks. Telling the app to wait for a few seconds is a perfect example of such an "async" task.

➤ Add these lines to the bottom of the `done()` action method:

```
let delayInSeconds = 0.6
DispatchQueue.main.asyncAfter(deadline: .now() + delayInSeconds, execute:
{
  self.dismiss(animated: true, completion: nil)
})
```

Believe it or not, these incantations tells the app to close the Tag Location screen after 0.6 seconds.

The magic happens in `DispatchQueue.main.asyncAfter()`. This function takes a closure as its final parameter. Inside that closure you tell the view controller to dismiss itself. This doesn't happen right away, though. That's the exciting thing about closures: even though this code sits side-by-side with all the other code in the method, everything that's inside the closure is ignored for now and kept for a

later time.

`DispatchQueue.main.asyncAfter()` uses the time given by `.now() + delayInSeconds` to schedule the closure for some point in the future. Until then, the app just sits there twiddling its thumbs. (By the way, `.now()` is a shortcut for `DispatchTime.now()`. Swift's type inference already knows that the type of the `when:` parameter is always a `DispatchTime` object, so you don't have to mention it explicitly.)

After 0.6 seconds, the code from the closure finally runs and the screen closes.

> **Note:** 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 close the screen too quickly or the effect from showing the HUD is lost, but it shouldn't take too long either or it will annoy the user. Animations are cool but they 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.

Now, I don't know about you but I find this Grand Central Dispatch stuff to be a bit messy. So let's clean up the code and make it easier to understand.

➤ Add a new file to the project using the **Swift File** template. Name the file **Functions.swift**.

➤ Replace the contents of the new file with:

```
import Foundation
import Dispatch

func afterDelay(_ seconds: Double, closure: @escaping () -> ()) {
  DispatchQueue.main.asyncAfter(deadline: .now() + seconds,
                                execute: closure)
}
```

That looks very much like the code you just added to `done()`, except it now lives in its own function, `afterDelay()`. This is a **free function**, not a method inside an object, and as a result it can be used from anywhere in your code.

Take a good look at `afterDelay()`'s second parameter, the one named `closure`. Its type is `() -> ()`. That's not some weird emoticon; it is Swift notation for a parameter that takes a closure with no arguments and no return value.

The `->` symbol means that the type represents a closure. The type for a closure generally looks like this:

```
(parameter list) -> return type
```

In this case, both the parameter list and the return value are empty, `()`. This can also be written as `Void -> Void` but I like the empty parentheses better.

So whenever you see a –> in the type annotation for a parameter, you know that this function takes a closure as one of its parameters.

`afterDelay()` simply passes this closure along to `DispatchQueue.main.asyncAfter()`.

The annotation `@escaping` is necessary for closures that are not performed immediately, so that Swift knows that it should hold on to this closure for a while.

You may be wondering why you're going through all this trouble. No fear! The reason why will become apparent after you've made the following change…

➤ Go back to **LocationDetailsViewController.swift** and change `done()` to:

```
@IBAction func done() {
  let hudView = HudView.hud(inView: navigationController!.view,
                            animated: true)
  hudView.text = "Tagged"

  afterDelay(0.6, closure: {
    self.dismiss(animated: true, completion: nil)
  })
}
```

Now that's the power of Swift! It only takes one look at this code to immediately understand what it does. After a delay, something happens.

By moving the nasty GCD stuff into a new function, `afterDelay()`, you have added a new **abstraction** into your code that makes it much easier to follow. Writing good programs is all about finding the right abstractions.

Note: Because the code to dismiss the view controller sits in a closure it needs to use `self` to call the method. Inside closures you always need to use `self` explicitly.

You can make this read even more naturally. Change the code to:

```
afterDelay(0.6) {
  self.dismiss(animated: true, completion: nil)
}
```

Now the closure sits *outside* of the call to `afterDelay()`.

Swift has a handy rule that says you can put a closure behind a function call if it's the last parameter. This is known as **trailing closure syntax**. You will usually see closures being used in this manner because it reads the nicest.

By the way, you can remove the `import Dispatch` line again from **LocationDetailsViewController.swift**. This file no longer uses GCD directly and therefore doesn't need to import the framework.

➤ Run the app again to make sure the timing still works. Boo-yah!

You can find the project files for this section under **02 – Tagging Locations** in the tutorial's Source Code folder.

Value types and reference types

A type walks into a bar. "Do you have any variables?" he asks.

"Well, I don't know if it will be of much value," says the bartender, "but I can give you a reference!"

Anyway… Swift makes a distinction between **value types** and **reference types** and it uses different rules for each.

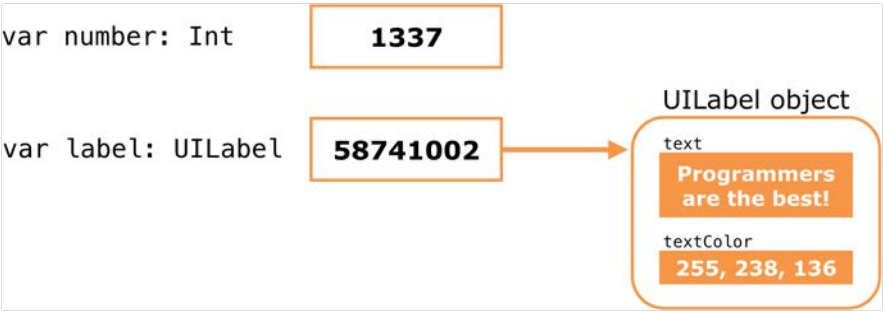
Objects that are made from classes are reference types. Anything else – all the basic types such as `Int`, `Bool`, `Float` and `Double`, `String`, `Array` and `Dictionary`, as well as `enum`s – are all value types.

So what exactly is the difference between a value type and a reference type?

When you declare a variable as an `Int` or a `CLLocationCoordinate2D` – both value types – the compiler reserves a small portion of memory to hold the value that you wish to store. For an `Int` that is 8 bytes, for a `CLLocationCoordinate2D` that is 16 bytes (on a 64-bit CPU such as in the iPhone 5s and later models).

Variables for reference types are no different. When you create a `UILabel` outlet, for example, the compiler also reserves 8 bytes of memory, exactly the same as for an `Int`. But a label can hold several lines of text – surely that won't all fit into those measly 8 bytes?

When you put some values into these variables, what happens is this:



The label variable contains a reference to a UILabel object

The value 1337 is put directly into `number`'s memory but the value of `label` is some weird number. It sure doesn't look like a text label. That number is not the actual `UILabel` object but the memory address of where that `UILabel` object lives. It is a *reference* to the label object.

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 `UILabel` object.

Obviously the 8 bytes that are reserved for the label variable are not enough to hold a `UILabel` object with the text `"Programmers are the best!"` and all the other label properties such as the text color.

So when the `UILabel` object is allocated, the computer puts it somewhere else in memory and returns the address of that memory location. That address – PO Box 58741002, Mike's iPhone, USA – is what gets put into the variable.

It's quite useful to think of a reference as a postal address. Your house is the actual object, but not every one of your acquaintances has a copy of your house. That would be silly. Instead, these people have a reference to your house: your address. To access your house (for example, to come over and eat ice cream), they look up the address so they can find where you live.

Don't worry if the distinction between value type and reference type doesn't quite make sense to you yet. It's a very powerful idea in computing and powerful ideas can take a while to sink in.

What you do need to know about the difference between value types and references types is this:

Rule 1. When you make a constant with `let`, you cannot change a value type once you have given it a value:

```
let coord1 = CLLocationCoordinate2D(latitude: 0, longitude: 0)
coord1.latitude = 37.33233141  // error

var coord2 = CLLocationCoordinate2D(latitude: 0, longitude: 0)
coord2.latitude = 37.33233141  // OK because you used “var”
```

But the object from a reference type can always be changed, even if you used `let`. For example:

```
let label = UILabel()
label.text = "Hello, world!"  // OK
label.text = "I like change"  // OK
```

What is constant is the reference itself. In other words you cannot put a new object into it:

```
let newLabel = UILabel()
label = newLabel              // error
```

Rule 2. When you put an object with a value type into a new variable or constant, it is copied. `Arrays` are value types in Swift. That means when you put an array into a new variable it will actually make a copy:

```
var a = [1, 2, 3]
```

```
let b = a
a.append(4)
print(a)    // prints [1, 2, 3, 4]
print(b)    // prints [1, 2, 3]
```

This doesn't happen with a reference type. The object itself isn't copied, only the reference:

```
var firstLabel = UILabel()
firstLabel.text = "Programmers are the best!"

var secondLabel = firstLabel
secondLabel.text = "I like ice cream"

print(firstLabel.text)   // prints "I like ice cream"
```

What happens is that `secondLabel` points to the same address as `firstLabel`, and therefore putting a new string in `secondLabel`'s text property changes it for both.

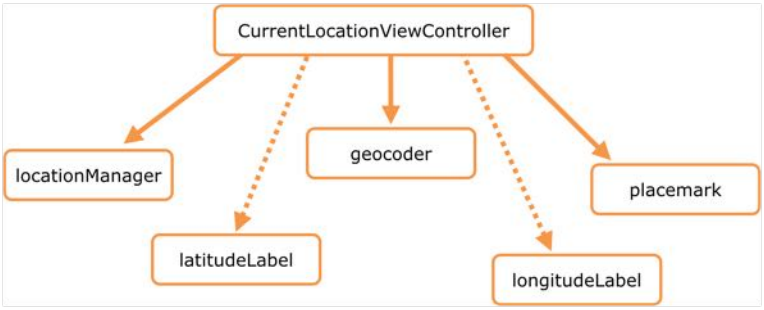
Both label variables refer to the same object

Remember, only objects made from classes are reference types. Everything else is a value type.

Object ownership

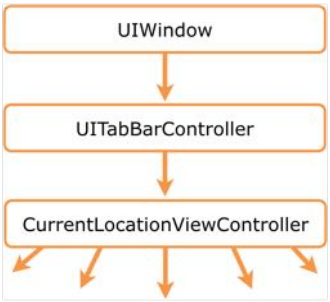
Objects are rarely hermits, off by themselves on a mountain somewhere. Your apps will have many objects that all need to work together. They are a very cooperative bunch!

The relationships between the objects in your app 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 constants, 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 *types* 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 also.

So what does it mean to "own" an object? This is where the concept of **strong** and **weak** references comes in.

Most of the time when you declare a variable that holds a reference object, you are

creating a strong relationship. This variable assumes ownership of the object, a responsibility that it shares with any other variables referencing the same object.

```
var image: UIImage            // a strong variable
```

In a weak relationship, there is no such ownership. A variable with a weak relationship to an object is not helping to keep this object alive. You need to explicitly declare this with the `weak` keyword (and make it an optional):

```
weak var weakImage: UIImage?  // a weak variable
```

The lifetime of an object is determined by how many owners it has at any given point. As soon as there are no more strong references to an object, no matter how many weak references there still are, the object is ruthlessly disposed of.

That's also when the object's `deinit` method is called, giving the object one last chance to get its affairs in order before it meets its maker.

So what's the point of `weak`?

It primarily exists to avoid the problem of **ownership cycles** where two objects end up owning each other. Because these objects keep each other alive, the app will never reclaim their memory after they've served their purpose. This slowly eats up the available memory for the app – the dreaded memory leak! – until there is no free memory left and the app crashes.

In the previous tutorial you learned to make delegates weak to avoid this kind of situation. Later on you'll see another example of an ownership cycle and how to fix it (hint: you'll use a weak reference).

Another common use of `weak` is with `@IBOutlet` properties. The reason outlets are weak is that the view controller isn't really their owner; instead, the outlets belong to the view controller's top-level view.

Nothing untoward will happen if you have strong outlets, but by declaring them `weak` the view controller says, "I'm only renting these outlets, I don't need to own them."

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, 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 would break loose if you tried to use it. (Using such "zombie" objects is a common cause of crashes in languages that don't have weak references.)

This is also why weak variables must always be optionals (either `?` or `!`) because they may become `nil` at some point.

> **Note:** Swift has a third type of relationship: `unowned`. Like `weak` this is used to break cycles. The difference is that `unowned` variables shouldn't be `nil` and therefore don't have to be optionals. It's less common than `weak`, but you'll probably run into `unowned` variables here or there. Just think of them as `weak` – it's what I do.

To recap: if you want to keep an object around for a while, you should store it with a strong relationship. But if you don't need to own the object and you don't mind it going away at some point, use a `weak` or `unowned` reference.

Let's look at a situation where there is more than one owner of an object.

For example, when the user taps the Tag Location button, the Current Location view controller passes a `CLPlacemark` object to the Location Details view controller. Both references are strong so from that moment on, the Location Details view controller assumes shared ownership of it. Now the `CLPlacemark` has two owners.

It's important to realize that the concept of shared ownership and object relationships only applies to reference types. Value types are always copied from one place to the other, so they never have more than one owner.

`LocationDetailsViewController` also has a `coordinate` property that is filled in upon the segue from the Tap Location button. This is a `CLLocationCoordinate2D`, a struct and therefore a value type.

In `prepare(for:sender:)` the following happens:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  . . .
  controller.coordinate = location!.coordinate
  controller.placemark = placemark
}
```

The line that puts `location!.coordinate` into `controller.coordinate` makes a copy of that `CLLocationCoordinate2D` struct. Each view controller now has its own unique copy of those GPS coordinates (it's a value type!).

For `placemark` the only thing that is copied is the reference. Both view controllers point to the same `CLPlacemark` object (or `nil` if `placemark` has no value).

The coordinate is copied but the placemark is a reference

Make sense? Great! If not, read it again. ;-)

Seriously though, if you understand these concepts you're well on your way to programming greatness. And if not, don't give up. These can be tough ideas to grok. Just keep going until it does make sense (and I promise it will!).

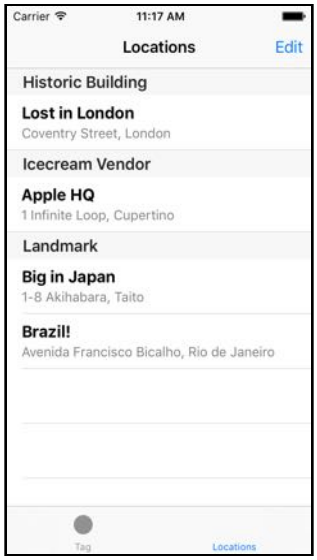
> **Note:** If you want an object with a reference type to be copied when it is assigned to another variable, you can declare it as `@NSCopying`. This first creates a duplicate of the object and puts that in the new variable. Now the two variables each refer to their own object. `@NSCopying` is more of an Objective-C thing, but now you know what it's for should you come across it.

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 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 framework that can take a lot of work out of your hands: Core Data.

Core Data is an object persistence framework for iOS apps. If you've looked at Core Data before, you may have found the official documentation 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.

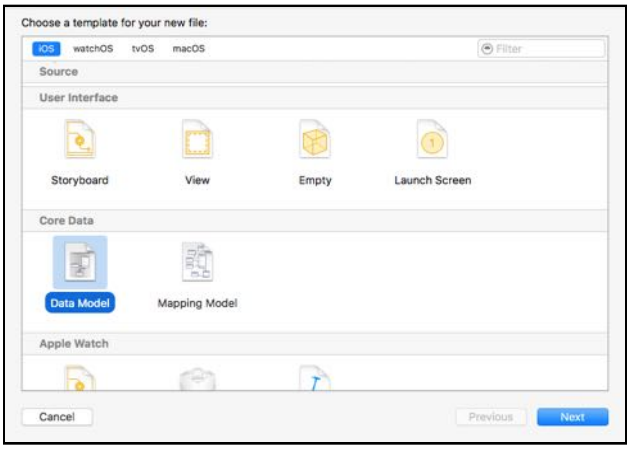
With Core Data, you can designate some objects as being persistent so they will always be saved to a **data store**. Even when all references to such a **managed object** are gone and the instance gets destroyed, its data is still safely stored in Core Data and you can get it back at any time.

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

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 (scroll down in the template chooser):

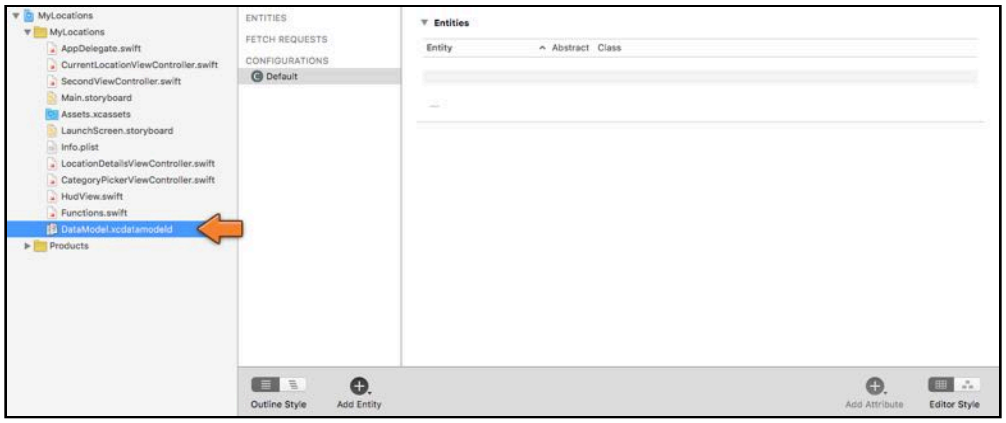


Adding a Data Model file to the project

➤ Save it as **DataModel**.

This will add a new file to the project, DataModel.xcdatamodeld.

➤ Click **DataModel.xcdatamodeld** in the Project navigator 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**.

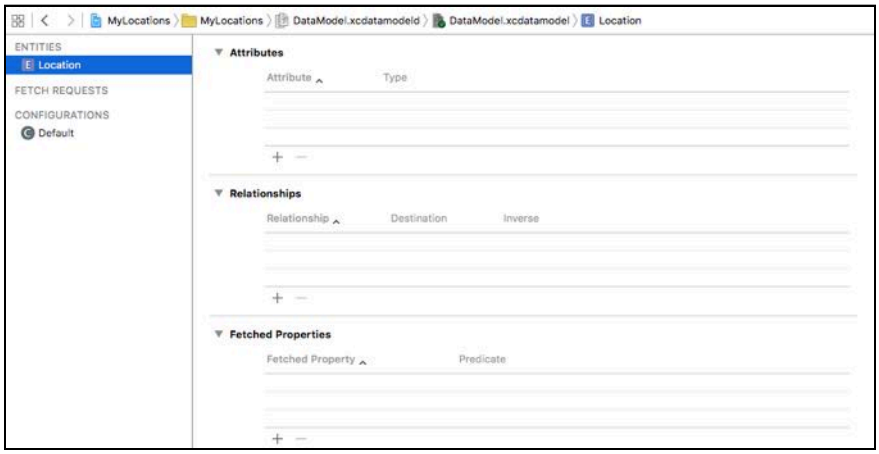
An entity describes which data fields your objects will have. In a sense it serves the same purpose as a class but specifically for Core Data's data store. (If you've worked with SQL databases before, you can think of an entity as a table.)

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
- the user's 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 pane on the right.)



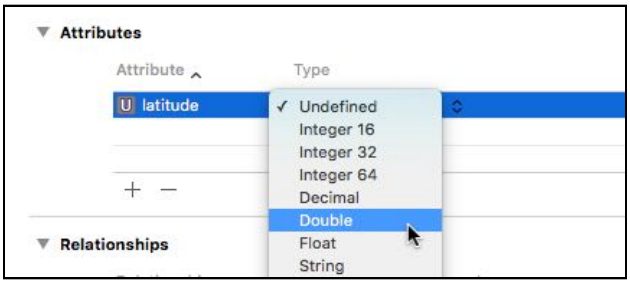
The new Location entity

Inside the entity 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 type. You've seen earlier that the latitude and longitude coordinates really have the data type `Double`, so that's what you're choosing for the attribute as well.

> **Note:** 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

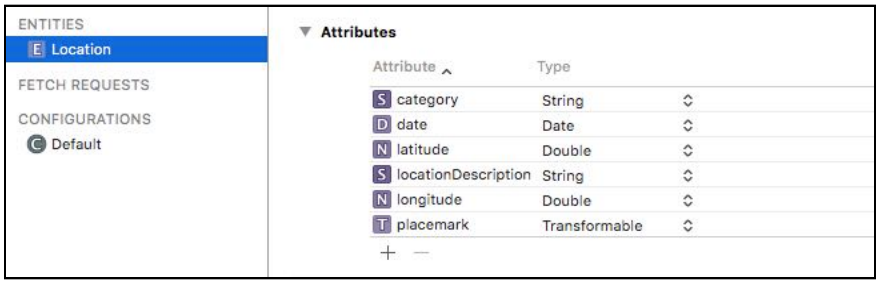
what an entity describes: the data of an object, and optionally how that object relates to other objects if you use Relationships and Fetched Properties.

In a short while you are going to define your own `Location` class by creating a Swift 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

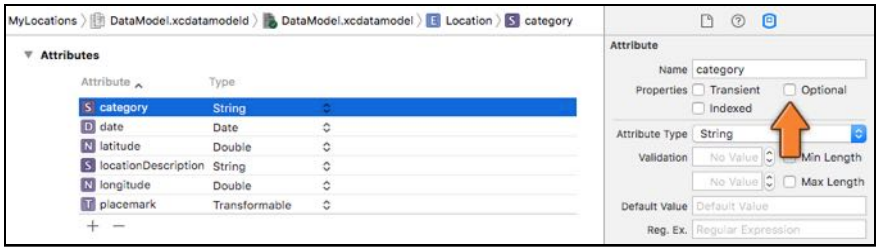
Why didn't you just call it "description" instead of "locationDescription"? As it turns out, `description` is the name of a method from `NSObject`. If you try to name an attribute "description", then it will cause a naming conflict with that method. Xcode will give you an error message if you try to do this.

The type of the placemark attribute is Transformable. Core Data only supports a limited number of data types right out the box, such as `String`, `Double`, and `Date`. The placemark is a `CLPlacemark` object and is not in the list of supported data types.

Fortunately, Core Data has a provision for handling arbitrary data types. 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.

By default, entity attributes are optional, meaning they can be `nil`. In our app, the only thing that can be `nil` is the placemark, in case reverse geocoding failed. It's a good idea to embed this constraint in the data model.

➤ Select the category attribute. In the inspectors panel, switch to the Data Model inspector (third tab). Uncheck the Optional setting:



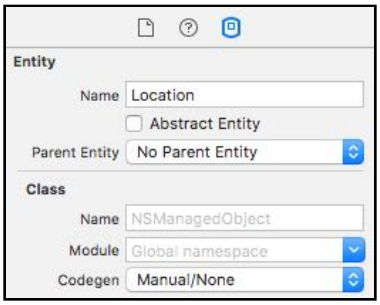
Making the category attribute non-optional

➤ Repeat this for the other attributes, except for placemark. (Tip: you can select multiple attributes at the same time.)

➤ Press ⌘**+S** to save your changes. Xcode is supposed to do this automatically, but I've found the data model editor to be a little unreliable at times. Better safe than sorry!

You're done with the data model, but there's one more thing I'd like to point out.

➤ Click on the Location entity to select it and go to the Data Model inspector.



The Data Model inspector

The Class field currently says "NSManagedObject". 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. You're not required to make your own classes for your entities, but it does make Core Data easier to use.

Now when you retrieve a Location entity from the data store, Core Data doesn't give you an `NSManagedObject` but an instance of your own `Location` class.

➤ In the inspector, change **Codegen** to **Manual/None**.

Note: As of version 8, Xcode can generate the source code for the entity's class automatically from the data model. The Codegen setting determines how it does this. For the purposes of this tutorial you're not using the automatic code generation, which is why you're setting Codegen to Manual/None. It's useful to understand how to make your own `NSManagedObject` subclass rather than relying on Xcode magic.

Even though you won't be using automatic class generation, Xcode can still lend a helping hand.

➤ From the menu bar, choose **Editor → Create NSManagedObject Subclass**.

The assistant will now ask you for which data model and which entity you wish to create the class.

➤ Select **DataModel** and click **Next**. In the next step, make sure **Location** is selected and click **Next** again.

Select the Location entity

➤ Choose a location to save the source files. Press **Create** to finish.

This adds two new files to the project. The first one is named **Location +CoreDataClass.swift** and looks something like this:

```
import Foundation
import CoreData

public class Location: NSManagedObject {

}
```

As you can see in the `class` line, the `Location` class extends `NSManagedObject` instead of the regular `NSObject`.

The second file that got created is **Location+CoreDataProperties.swift**:

```
import Foundation
import CoreData

extension Location {
 @nonobjc public class func fetchRequest() -> NSFetchRequest<Location> {
    return NSFetchRequest<Location>(entityName: "Location");
```

```
  }

  @NSManaged var latitude: Double
  @NSManaged var longitude: Double
  @NSManaged var date: NSDate?
  @NSManaged var locationDescription: String?
  @NSManaged var category: String?
  @NSManaged var placemark: NSObject?
}
```

In this file, Xcode has created properties for the attributes that you specified in the Data Model editor. But what is this `extension` thing?

With an *extension* you can add additional functionality to an existing object without having to change the source code for that object. This even works when you don't actually have the source code for those objects. Later on in the tutorial you'll see an example of how you can use an extension to add new methods to objects from the iOS frameworks.

Here, the extension is used for another purpose. If you change your Core Data model at some later time and you want to automatically update the code to match those changes, then you can choose **Create NSManagedObject Subclass** again and Xcode will only overwrite what is in **Location+CoreDataProperties.swift** but not anything you added to **Location+CoreDataClass.swift**.

So it's not a good idea to make changes to **Location+CoreDataProperties.swift** if you plan on overwriting this file later. Unfortunately, Xcode made a few small boo-boos in the types of the properties, so you'll have to make some changes to this file anyway.

The first thing to fix is the `placemark` variable. Because you made placemark a Transformable attribute, Xcode doesn't really know what kind of object this will be, so it chose the generic type `NSObject`. You know it's going to be a `CLPlacemark` object, so you can make things easier for yourself by changing it.

➤ First import Core Location into **Location+CoreDataProperties.swift**:

```
import CoreLocation
```

➤ Then change the `placemark` property to:

```
@NSManaged var placemark: CLPlacemark?
```

You're adding a question mark too, because `placemark` is optional.

➤ Also change the date property from `NSDate` to `Date`:

```
@NSManaged var date: Date
```

The `NSDate` class is what Objective-C uses to represent dates but in Swift we work with `Date`, without the "NS". It is also no longer an optional.

➤ Finally, remove the question marks behind the `category` and `locationDescription` properties. Earlier you told Core Data these attributes were not optionals, so they don't need the question mark.

Because this is a *managed* object, and the data lives inside a data store, Swift will handle `Location`'s variables in a special way. The `@NSManaged` 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 place that value into the data store for safekeeping, instead of in a regular instance variable.

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 an SQLite database (pronounced "SQL light"). It's OK if you have no idea what SQLite 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 this data store when the app starts. The code for that is 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.

➤ Open **AppDelegate.swift** and import the Core Data framework at the very top:

```
import CoreData
```

➤ Add the following code inside the `AppDelegate` class:

```
lazy var persistentContainer: NSPersistentContainer = {
  let container = NSPersistentContainer(name: "DataModel")
  container.loadPersistentStores(completionHandler: {
    storeDescription, error in
    if let error = error {
      fatalError("Could load data store: \(error)")
    }
  })
  return container
}()
```

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.

The goal here is to create a so-called `NSManagedObjectContext` object. That is the object you'll use to talk to Core Data. To get that `NSManagedObjectContext` object,

the app needs to do several things:

1. Create an `NSManagedObjectModel` from the Core Data model you created earlier. This object represents the data model during runtime. You can ask it what sort of entities it has, what attributes these entities have, and so on. In most apps you don't need to use the `NSManagedObjectModel` object directly.
2. Create an `NSPersistentStoreCoordinator` object. This object is in charge of the SQLite database.
3. Finally, create the `NSManagedObjectContext` object and connect it to the persistent store coordinator.

Together, these objects are also known as the "Core Data stack".

In iOS 9 you had to perform these steps by hand, which could get a little messy. Fortunately for us there is a new object in iOS 10, the `NSPersistentContainer`, that takes care of everything.

That doesn't mean you may immediately forget what you just learned about the `NSManagedObjectModel` and the `NSPersistentStoreCoordinator`, but it does save you from writing a bunch of code.

The code that you just added creates an instance variable `persistentContainer` of type `NSPersistentContainer`. To get the `NSManagedObjectContext` that we're after, you can simply ask the `persistentContainer` for its `viewContext` property.

➤ For convenience, add a new property to get the `NSManagedObjectContext` from the persistent container:

```
  lazy var managedObjectContext: NSManagedObjectContext =
                                    self.persistentContainer.viewContext
```

Now we're ready to start using Core Data!

➤ Build the app to make sure it compiles without errors. 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 and make tapping Done save a new `Location` object into the Core Data store.

I mentioned the `NSManagedObjectContext` object. 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 the `NSManagedObjectContext` object.

➤ Switch to **LocationDetailsViewController.swift**. Import Core Data into this file and then add a new instance variable:

```
var managedObjectContext: NSManagedObjectContext!
```

The problem is: how do you put the `NSManagedObjectContext` object from the app delegate into that property?

The context object is created by `AppDelegate`, but `AppDelegate` has no reference to the `LocationDetailsViewController`.

That's not so strange as the Location Details view controller doesn't exist until the user taps the Tag Location button. Prior to initiating that segue, there simply is no living `LocationDetailsViewController` object.

The answer is to pass along the `NSManagedObjectContext` object during the segue that presents the `LocationDetailsViewController`. The obvious place for that is `prepare(for:sender:)` in `CurrentLocationViewController`.

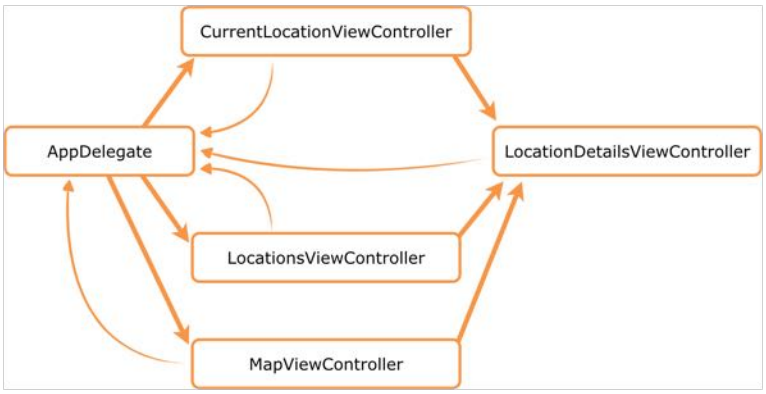
But now you need to find a way to get the `NSManagedObjectContext` object into the `CurrentLocationViewController` in the first place.

I come across a lot of code that does the following:

```
let appDelegate = UIApplication.sharedApplication().delegate
                                                    as! AppDelegate
let context = appDelegate.managedObjectContext

// do something with the context
```

From anywhere in your source code you can get a reference to the context simply by asking the `AppDelegate` for its `managedObjectContext` property. 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.

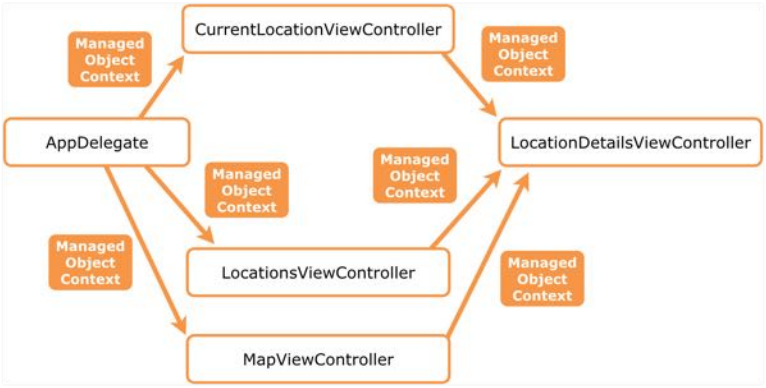


Bad: All classes depend on AppDelegate

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 give the `NSManagedObjectContext` to each object that needs it. Now all the arrows in the diagram go just one way:



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 on to the `LocationDetailsViewController` when it performs the segue. This technique is known as *dependency injection*.

This means `CurrentLocationViewController` needs its own property for the `NSManagedObject` context.

➤ Add the following property to **CurrentLocationViewController.swift**:

```
var managedObjectContext: NSManagedObjectContext!
```

Don't forget to also import Core Data or Xcode won't know what you mean by `NSManagedObjectContext`.

➤ Add the following to `prepare(for:sender:)`, so that it passes on the context to the Tag Location screen:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "TagLocation" {
    . . .

    controller.coordinate = location!.coordinate
    controller.placemark = placemark
    controller.managedObjectContext = managedObjectContext   // add this
```

```
    }
}
```

This should also explain why the `managedObjectContext` variable is declared as an implicitly unwrapped optional with the type `NSManagedObjectContext!`.

You should know by now that variables in Swift must always have a value. If they can be `nil` – which means "not a value" – then the variable must be made optional.

If you were to declare `managedObjectContext` without the exclamation point,

```
var managedObjectContext: NSManagedObjectContext
```

then Swift demands you give it a value inside an `init` method. For objects loaded from a storyboard, such as view controllers, that is `init?(coder)`.

However, `prepare(for:sender:)` happens *after* the new view controller is instantiated, long after the call to `init?(coder)`. As a result, inside `init?(coder)` you can't know yet what the value for `managedObjectContext` is.

You have no choice but to leave the `managedObjectContext` variable `nil` for a short while until the segue happens, and therefore it must be an optional.

You could also have declared it like this:

```
var managedObjectContext: NSManagedObjectContext?
```

The difference between `?` and `!` is that the former requires you to manually unwrap the value with `if let` every time you want to use it.

That gets annoying fast, especially when you know that `managedObjectContext` will get a proper value during the segue and that it will never become `nil` afterwards again. In that case, the exclamation mark is the best type of optional to use.

These rules for optionals may seem very strict – and possibly confusing – when you're coming from another language such as Objective-C, but they are there for a good reason. By only allowing certain variables to have no value, Swift can make your programs safer and reduce the number of programming mistakes.

The fewer optionals you use, the better, but sometimes you can't avoid them – as in this case with `managedObjectContext`.

AppDelegate.swift now needs some way to pass the `NSManagedObjectContext` object to `CurrentLocationViewController`.

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.

➤ Change the `application(didFinishLaunchingWithOptions)` method to:

```
func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions launchOptions:
                                 [UIApplicationLaunchOptionsKey: Any]?)
                 -> Bool {

  let tabBarController = window!.rootViewController
                                                 as! UITabBarController

  if let tabBarViewControllers = tabBarController.viewControllers {
    let currentLocationViewController = tabBarViewControllers[0]
                                     as! CurrentLocationViewController

    currentLocationViewController.managedObjectContext =
                                                   managedObjectContext
  }
  return true
}
```

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 a reference to the `CurrentLocationViewController` object, you give it the `managedObjectContext`. It may not be immediately obvious from looking at the code, but something special happens at this point…

Remember the code for `persistentContainer` you added to AppDelegate earlier? This is what it looked like (I left out most of the actual code):

```
lazy var persistentContainer: NSPersistentContainer = {
  let container = . . .
  . . .
  return container
}()
```

These lines declare an instance variable of type `NSPersistentContainer`. Like a good Swift variable it has an initial value, because it says:

```
lazy var persistentContainer: NSPersistentContainer = something
```

And that something is a block of code in between `{ }` braces:

```
{
  let container = . . .
  . . .
  return container
}()
```

You may recognize this as a closure. You've already seen those a few times now. Normally a closure contains source code that is not performed right away, so you can store it for later use. Here, however, the parentheses at the end of the closure invoke it immediately.

To make this a bit clearer, here is another example:

```
var four: Int = { return 2 + 2 }()
```

The initial value of this variable is the result of the closure `{ return 2 + 2 }`. It's a bit silly to do this when you could have just written `var four = 4`, but in the case of `persistentContainer` it is really handy.

What actually happens inside the closure is fairly straightforward:

```
let container = NSPersistentContainer(name: "DataModel")
container.loadPersistentStores(completionHandler: {
  storeDescription, error in
  if let error = error {
    fatalError("Could load data store: \(error)")
  }
})
return container
```

You instantiate a new `NSPersistentContainer` object with the name of the data model you created earlier, `"DataModel"`. Then you tell it to `loadPersistentStores()`, which loads the data from the database into memory and sets up the Core Data stack.

There is another closure here, given by the `completionHandler` parameter. The code in this closure gets invoked when the persistent container is done loading the data. If something went wrong, you print an error message – useful for debugging! – and terminate the app using the function `fatalError()`.

Now that you know what it does, you may be wondering why you didn't just put all of this code into a regular method and did something like this:

```
var persistentContainer: NSPersistentContainer

init() {
  persistentContainer = createPersistentContainer()
}

func createPersistentContainer() -> NSPersistentContainer {
  // all the initialization code here
  return container
}
```

That would certainly be possible, but now the initialization of `persistentContainer` is spread over three different parts of the code: the declaration of the variable, the method that performs all the initialization logic, and the `init` method to tie it all together.

Isn't it nicer to keep all this stuff in one place, rather than chopped up into three different pieces? Swift lets you perform complex initialization right where you declare the variable. I think that's pretty nifty.

There's another thing going on here:

```
lazy var persistentContainer: NSPersistentContainer = { . . . }()
```

Notice the `lazy` keyword? That means the entire block of code in the `{ . . . }()` closure isn't actually performed right away. The context object won't be created until you ask for it. This is another example of **lazy loading**, just like what you did with `DateFormatter`.

The `managedObjectContext` property was also declared `lazy`:

```
lazy var managedObjectContext: NSManagedObjectContext =
                                    self.persistentContainer.viewContext
```

This is necessary because its initial value comes from `persistentContainer`. It's also necessary to use `self` here to refer to `persistentContainer`. Without, Xcode gives a compiler error.

➤ Run the app. Everything should still be the way it was, but behind the scenes a new database has been created for Core Data.

A peek inside the data store

As mentioned, Core Data stores the data inside an SQLite database. That file is named **DataModel.sqlite** and it lives in the app's Library folder. That's similar to the Documents folder that you've seen in the previous tutorial.

You can see it in Finder if you go to **~/Library/Developer/CoreSimulator** and then to the folder that contains the data for the MyLocations app.

➤ The easiest way to find this folder is to add the following to **Functions.swift**:

```
let applicationDocumentsDirectory: URL = {
  let paths = FileManager.default.urlsForDirectory(
                          .documentDirectory, inDomains: .userDomainMask)
  return paths[0]
}()
```

This creates a new global constant, `applicationDocumentsDirectory`, containing the path to the app's Documents directory. It's a global because you're not putting this inside a class. This constant will exist for the duration of the app; it never goes out of scope. You could have made a method for this as you did in the Checklists tutorial, but using a global constant works just as well.

As before you're using a closure to provide the code that initializes this constant. Like all globals, this is evaluated lazily the very first time it is used.

> **Note:** Globals have a bad reputation. Many programmers avoid them at all costs. The problem with globals is that they create hidden dependencies between the various parts of your program. And dependencies make the program hard to change and hard to debug.

> But used well, globals can be very handy. It's feasible that your app will need to know the path to the Documents directory in several different places. Putting it in a global constant is a great way to solve that design problem.

➤ Add the following line to `application(didFinishLaunchingWithOptions)`:

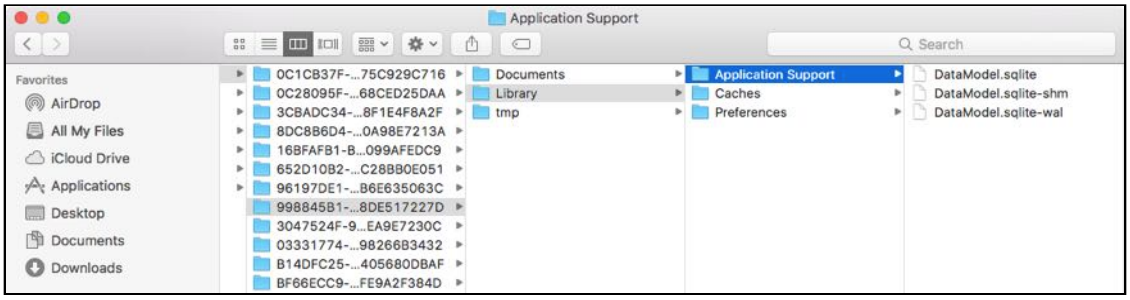
```
print(applicationDocumentsDirectory)
```

On my computer this prints out:

```
file:///Users/matthijs/Library/Developer/CoreSimulator/Devices/66422991-
21E3-4394-8DCE-0584865EA854/data/Containers/Data/Application/998845B1-
8E87-40F7-9C3E-FC8DE517227D/Documents/
```

➤ Open a new Finder window and press **Shift+⌘+G**. Then copy-paste the path without the `file://` bit to go to the Documents folder.

The database is not actually in the Documents folder, so go back one level and enter the **Library** folder, **Application Support**:



The new database in the app's Documents directory

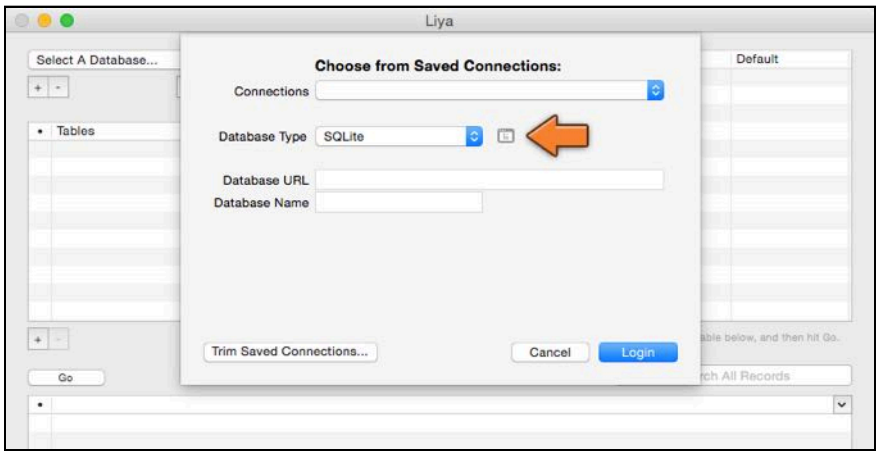
The **DataModel.sqlite-shm** and **-wal** files are also part of the data store.

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 the Mac App Store or www.cutedgesystems.com/software/liya/.

➤ Start Liya. It asks you for a database connection. Under **Database Type** choose **SQLite**.



Liya opens with this dialog box

➤ On the right of the Database Type field is a small icon. Click this to open a file picker.

You can navigate to the **CoreSimulator/.../Library/Application Support** folder, but that's a lot of work (it's a very deeply nested folder).

If you have the Finder window still open, it's easier to drag the DataModel.sqlite file from Finder directly into the open file picker. Click **Choose** when you're done.

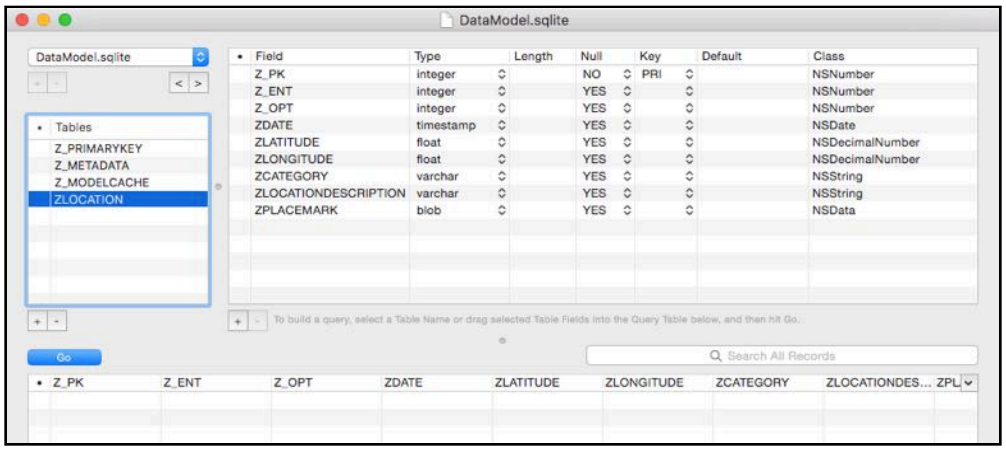
> Tip: You can also right-click the DataModel.sqlite file in Finder and choose **Open With → Liya** from the popup menu.

The **Database URL** field should now contain the app's Document folder and **Database Name** should say DataModel.sqlite:

Connecting to the SQLite database

➤ Click **Login** to proceed.

The screen should look something like this:



The empty DataModel.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 also 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.

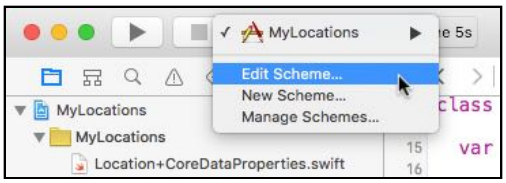
> **Note:** An alternative to Liya is SQLiteStudio, sqlitestudio.pl. You can find more tools, paid and free, on the Mac App Store by searching for "sqlite".

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**.

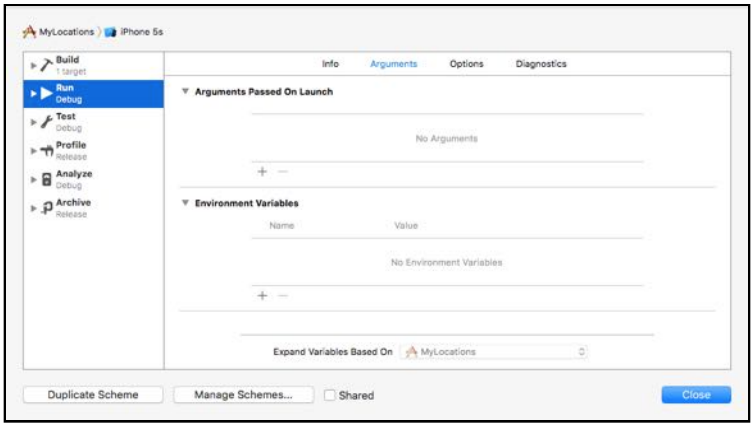
Schemes are how Xcode lets you configure your projects. A scheme is a bunch of settings for building and running your app. Standard projects have just one scheme but you can add additional schemes, which is handy when your project becomes bigger.

➤ Click on the left part of the **MyLocations > iPhone** bar at the top of the screen and choose **Edit Scheme...** from the menu.



The Edit Scheme... option

The following panel should pop up:



The scheme editor

➤ Choose the **Run** option on the left-hand side.

➤ Select the **Arguments** tab.

➤ In the **Arguments Passed On Launch** section, add the following:

```
-com.apple.CoreData.SQLDebug 1
-com.apple.CoreData.Logging.stderr 1
```

Adding the SQLDebug launch argument

➤ Press **Close** to close this dialog, and run the app.

When it starts, you should see something like this in the debug area:

```
CoreData: annotation: Connecting to sqlite database file at "/Users/
matthijs/Library/Developer/CoreSimulator/Devices/66422991-21E3-
4394-8DCE-0584865EA854/data/Containers/Data/Application/998845B1-8E87-
40F7-9C3E-FC8DE517227D/Library/Application Support/DataModel.sqlite"
```

```
CoreData: sql: SELECT TBL_NAME FROM SQLITE_MASTER WHERE TBL_NAME =
'Z_METADATA'
CoreData: sql: pragma journal_mode=wal
CoreData: sql: pragma cache_size=200
CoreData: sql: SELECT Z_VERSION, Z_UUID, Z_PLIST FROM Z_METADATA
```

This is the debug output from Core Data. If you speak SQL, some of this will look familiar. 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 put a new `Location` object into the data store when the Done button is pressed.

➤ Add the following instance variable to **LocationDetailsViewController.swift**:

```
var date = Date()
```

You're adding this variable because you need to store the current date in the new `Location` object. You only want to make that `Date` object once.

➤ In `viewDidLoad()`, change the line that sets the `dateLabel`'s text to:

```
dateLabel.text = format(date: date)
```

This now uses the new instance variable.

➤ Change the `done()` method to the following:

```
@IBAction func done() {
  let hudView = HudView.hud(inView: navigationController!.view,
                            animated: true)
  hudView.text = "Tagged"

  // 1
  let location = Location(context: managedObjectContext)

  // 2
  location.locationDescription = descriptionTextView.text
  location.category = categoryName
  location.latitude = coordinate.latitude
  location.longitude = coordinate.longitude
  location.date = date
  location.placemark = placemark

  // 3
  do {
    try managedObjectContext.save()

    afterDelay(0.6) {
      self.dismiss(animated: true, completion: nil)
    }
```

```
    } catch {
      fatalError("Error: \(error)")
    }
  }
```

This is where you do all the work:

1. First, you create a new `Location` instance. Because this is a managed object, you have to use its `init(context:)` method. You can't just write `Location()` because then the `managedObjectContext` won't know about the new object.
2. Once you have created the `Location` instance, you can use it like any other object. Here you set its properties to whatever the user entered in the screen.
3. You now have a new `Location` object whose properties are all filled in, 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.

Saving takes any objects that were added to the context, or any managed objects that had their contents changed, and permanently writes these changes into the data store. That's why they call the context the "scratchpad"; its changes aren't persisted until you save them.

The `save()` method can fail for a variety of reasons and therefore you need to catch that potential error. That's done using the `do-try-catch` feature of Swift.

do-try-catch

Sometimes operations from the iOS SDK can fail and return an error of some kind. You've already seen the `Error` (and `NSError`) object that is used to describe such errors. But working with `Error` can be cumbersome, as it's really something that is borrowed from Objective-C.

Swift has a nicer way to deal with errors. It comes down to this: any method that can potentially fail must have the `try` keyword in front of it. And that method call with the `try` keyword must be inside a `do-catch` block.

Saving the managed object context is an operation that can potentially fail. That's why you wrote it like this:

```
do {
  try managedObjectContext.save()

  // code that runs when the "try" succeeds . . .
} catch {
  // code that runs when the "try" fails . . .
```

```
  }
```

If something goes wrong and the method fails – or "throws an error" as we say – the app skips the rest of the `do` section and immediately jumps to the `catch` section to handle the error.

If you've programmed in languages that use exceptions this may look familiar, but there's a… catch. Any method call that may potentially throw an error must be written as `try methodName()`. That makes it easy to spot which method invocations can throw errors.

You'll see several more examples of `do-try-catch` in the rest of this tutorial and the next one. It's a very important topic in Swift because nobody likes errors. They must be caught!

➤ Run the app and tag a location. Enter a description and press the Done button.

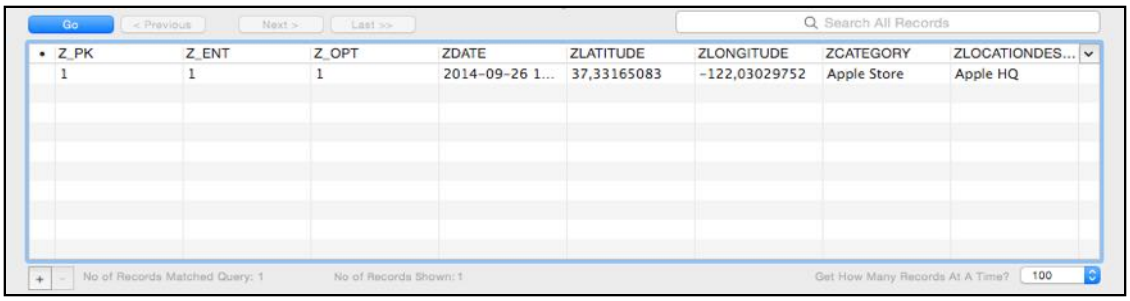
If everything went well, Core Data will dump a whole bunch of debug information into the debug area:

```
CoreData: sql: BEGIN EXCLUSIVE

. . .

CoreData: sql: INSERT INTO ZLOCATION(Z_PK, Z_ENT, Z_OPT, ZCATEGORY,
ZDATE, ZLATITUDE, ZLOCATIONDESCRIPTION, ZLONGITUDE, ZPLACEMARK) VALUES(?,
?, ?, ?, ?, ?, ?, ?, ?)

CoreData: sql: COMMIT

. . .

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.

> **Note:** Again, in Xcode 8.0 the Core Data debug output does not work. Sorry about that.

➤ In Liya, refresh the contents of the ZLOCATION table (press the Go button). There should now be one row in that table:



A new row was added to the table

(If you don't see any rows in the table, press the Stop button in Xcode first to exit the app. You can also try closing the Liya window and opening a new connection to the database.)

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:

```
do {
  try managedObjectContext.save()
  . . .
} catch {
  fatalError("Error: \(error)")
}
```

What if something goes wrong with the save? In that case, the `try` jumps into the `catch` section and you call the `fatalError()` function. That 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.

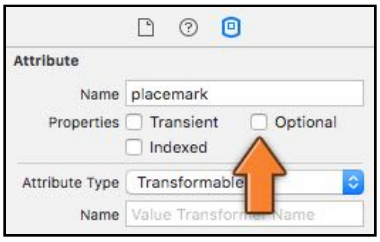
The bad news is that there isn't much else to do but crash when Core Data does give an error. Something went horribly wrong somewhere and now you're stuck with invalid data. If the app were allowed to continue, things would likely only get worse, as there is no telling what state the app is in. The last thing you want to do is to corrupt the user's data.

Instead of making the app crash hard with `fatalError()`, it will be nice to tell the user about it first so at least they know what is happening. The crash is still inevitable, but now your users will know why the app suddenly stopped working.

In this section, you'll add a popup alert for handling such situations. Again, these errors should happen only during development, but just in case they do 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, just to test your error handling code and to make sure the app fails gracefully.

➤ Run the app. It is possible that the app crashes right away…

What happened is that you have just changed the data model by making changes to the `placemark` attribute. When you launch the app, the `NSPersistentContainer` notices this and tries to perform a so-called "migration" of the SQLite database to the new, updated data model.

The migration may succeed… or not… depending on what is currently in your data store. If you previously tagged a location that did not have a valid address – i.e. whose `placemark` is `nil` – then the migration to the new data model fails. After all, the new data model does not allow for `placemark`s that are `nil`.

If the app also crashed for you, then the debug area says why:

```
reason=Cannot migrate store in-place: Validation error missing attribute
values on mandatory destination attribute, . . . {entity=Location,
attribute=placemark, . . .}
```

The DataModel.sqlite file is out of date with respect to the changed data model, and Core Data can't automatically resolve this issue.

There are two ways to fix this: 1) simply throw away the DataModel.sqlite file from the Library directory; 2) remove the entire app from the Simulator.

➤ Remove the DataModel.sqlite file, as well as the –shm and –wal files, 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 may require you to throw away the database file or Core Data cannot be initialized properly.

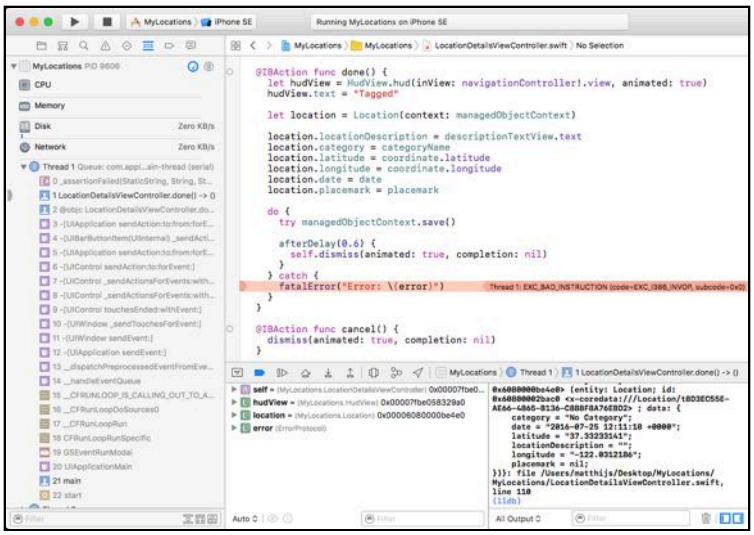
> **Note:** Not all is lost if `NSPersistentContainer`'s migration fails. Core Data allows you to perform your own migrations when you release an update to your app with a new data model. Instead of crashing, this mechanism allows you to convert the contents of the user's existing data store to the new model. However, when developing it is 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, you can fake this by temporarily commenting out the line `self.placemark = p.last!` in `locationManager(didUpdateLocations)` inside **CurrentLocationViewController.swift**. This will make it seem no address was found and the value of `placemark` stays `nil`.

➤ Tap the Done button to save the new `Location` object.

The app will crash on the call to `fatalError()`:



The app crashes after a Core Data error

In the debug area, you can see that it says:

```
The operation couldn't be completed . . . 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`.

Of course, what you've just seen only 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 when this happens. After the user dismisses that alert, you'll still 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.)

➤ Add the following code to **Functions.swift**:

```
let MyManagedObjectContextSaveDidFailNotification = Notification.Name(
              rawValue: "MyManagedObjectContextSaveDidFailNotification")

func fatalCoreDataError(_ error: Error) {
  print("*** Fatal error: \(error)")
  NotificationCenter.default.post(
       name: MyManagedObjectContextSaveDidFailNotification, object: nil)
```

```
  }
```

This defines a new global function for handling fatal Core Data errors.

➤ Replace the error handling code in the done() action with:

```
  do {
    try managedObjectContext.save()
    . . .
  } catch {
    fatalCoreDataError(error)
  }
```

The call to `fatalCoreDataError()` has taken the place of `fatalError()`. So what does that new function do, actually?

It first outputs the error message to the Debug Area using `print()` because it's always useful to log such errors. After dumping the debug info, the function does the following:

```
  NotificationCenter.default.post(
        name: MyManagedObjectContextSaveDidFailNotification, object: nil)
```

I've been using the term "notification" to mean any generic event or message being delivered, but the iOS SDK also has an object called the `NotificationCenter` (not to be confused with Notification Center on your phone).

The code above uses `NotificationCenter` to post a notification. Any object in your app can subscribe to such notifications and when these occur, `NotificationCenter` 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. The sender just sends the notification but doesn't really care what happens to it. If anyone is listening, great. If not, then that's cool too.

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 when the user taps the Home button.

You can also define your own notifications, and that is what you've done here. The new notification is called `MyManagedObjectContextSaveDidFailNotification`.

The idea is that there is one place in the app that listens for this notification, pops up an alert view, and terminates. The great thing about using `NotificationCenter` 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 `fatalCoreDataError()` function 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 methods to **AppDelegate.swift**:

```
func listenForFatalCoreDataNotifications() {
  // 1
  NotificationCenter.default.addObserver(
       forName: MyManagedObjectContextSaveDidFailNotification,
       object: nil, queue: OperationQueue.main, using: { notification in

    // 2
    let alert = UIAlertController(
      title: "Internal Error",
      message:
        "There was a fatal error in the app and it cannot continue.\n\n"
        + "Press OK to terminate the app. Sorry for the inconvenience.",
      preferredStyle: .alert)

    // 3
    let action = UIAlertAction(title: "OK", style: .default) { _ in
      let exception = NSException(
                  name: NSExceptionName.internalInconsistencyException,
                  reason: "Fatal Core Data error", userInfo: nil)
      exception.raise()
    }

    alert.addAction(action)

    // 4
    self.viewControllerForShowingAlert().present(alert, animated: true,
                                                 completion: nil)
  })
}

// 5
func viewControllerForShowingAlert() -> UIViewController {
  let rootViewController = self.window!.rootViewController!
  if let presentedViewController =
                            rootViewController.presentedViewController {
    return presentedViewController
  } else {
    return rootViewController
  }
}
```

Here's how this works step-by-step:

1. Tell `NotificationCenter` that you want to be notified whenever a `MyManagedObjectContextSaveDidFailNotification` is posted. The actual code that is performed when that happens sits in a closure following `using:`.
2. Create a `UIAlertController` to show the error message.
3. Add an action for the alert's OK button. The code for handling the button press is again a closure (these things are everywhere!). Instead of calling

`fatalError()`, the closure creates an `NSException` object to terminate the app. That's a bit nicer and it provides more information to the crash log.

4. Finally, you present the alert.

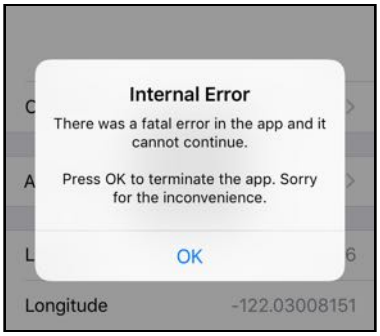
5. To show the alert with `present(animated, completion)` you need a view controller that is currently visible, so this helper method finds one that is. Unfortunately you can't simply use the window's `rootViewController` – in this app that is the tab bar controller – because it will be hidden when the Location Details screen is open.

All that remains is calling this new `listenForFatalCoreDataNotifications()` method so that the notification handler is registered with `NotificationCenter`.

➤ Add the following to `application(didFinishLaunchingWithOptions)`, just before the `return true` statement:

```
listenForFatalCoreDataNotifications()
```

➤ Run the app again and try to tag a location before the street address has been obtained. Now as 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 throws an error, then whatever value the user entered was invalid somehow and you should handle it.

➤ In the data model, set the **placemark** attribute back to optional.

Run the app just to make sure everything works as it should.

(If you commented out the line in `locationManager(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 **03 - 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 Scene**. 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.)

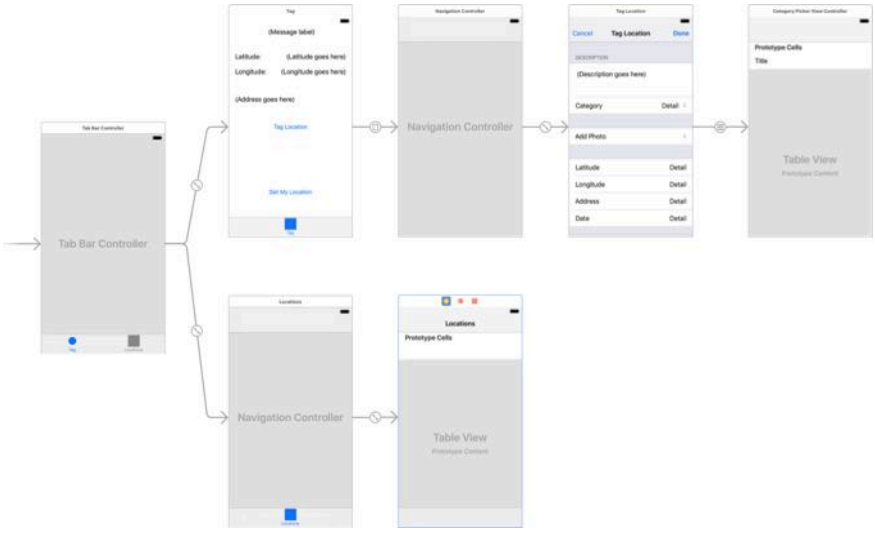
➤ **Ctrl-drag** from the Tab Bar Controller to this new Navigation Controller and select **Relationship Segue - view controllers**. This adds the navigation controller to the tab bar.

➤ 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**. (If Xcode gives you trouble, use the Attributes inspector on the Navigation Item instead.)

➤ In the **Identity inspector**, change the **Class** of this 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

➤ Run the app and activate the Locations tab. 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 **LocationCell**.

➤ In the **Size inspector**, change **Row Height** to 57.

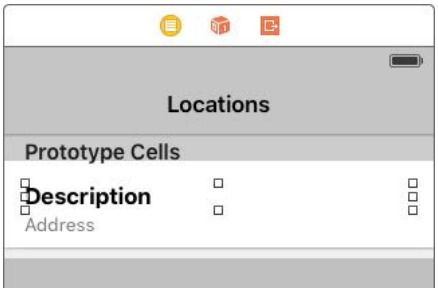
➤ Drag two Labels into the cell. Give the top one the text **Description** and the bottom one the text **Address**. This is just so you know what they are for.

➤ Set the font of the Description label to **System Bold**, size **17**. Give this label a tag of 100.

➤ Set the font of the Address label to **System**, size **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:

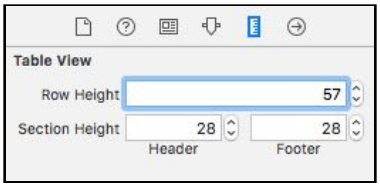


The prototype cell

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 a good idea to make sure that the prototype cell works before you have to deal with Core Data.

➤ Add a new file to the project and name it **LocationsViewController.swift**.

Tip: If you want to keep your list of source files neatly sorted by name in the project navigator, then right-click the MyLocations group (the yellow folder icon) and choose **Sort by Name** from the menu.

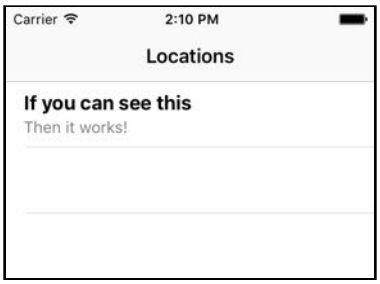
➤ Change the contents of **LocationsViewController.swift** to:

```
import UIKit
import CoreData
import CoreLocation

class LocationsViewController: UITableViewController {
  var managedObjectContext: NSManagedObjectContext!
```

```
  // MARK: - UITableViewDataSource

  override func tableView(_ tableView: UITableView,
                          numberOfRowsInSection section: Int) -> Int {
    return 1
  }

  override func tableView(_ tableView: UITableView,
                 cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
                         withIdentifier: "LocationCell", for: indexPath)

    let descriptionLabel = cell.viewWithTag(100) as! UILabel
    descriptionLabel.text = "If you can see this"

    let addressLabel = cell.viewWithTag(101) as! UILabel
    addressLabel.text = "Then it works!"

    return cell
  }
}
```

You're faking a single row with some placeholder text in the labels. You're also already giving this class an `NSManagedObjectContext` property even though you won't be using it yet.

➤ Run the app to make sure the table view works.



The table view with fake data

Note: If the table remains empty, then go back to the storyboard. In the **Identity inspector** for the view controller, the **Module** field is most likely empty or says "None". Click the blue arrow to change it to **MyLocations**.

Swift apps consist of one or more *modules*. Each target in your project is compiled into its own module with its own namespace, and Interface Builder needs to know in which module your view controller class lives. Because you filled in the Class field before `LocationsViewController` existed, Xcode got confused. Press **⌘+S** to save the changes and run the app again.

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 new 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.swift**:

```
var locations = [Location]()
```

This array contains the list of `Location` objects.

➤ Add the `viewDidLoad()` method:

```
override func viewDidLoad() {
  super.viewDidLoad()
  // 1
  let fetchRequest = NSFetchRequest<Location>()
  // 2
  let entity = Location.entity()
  fetchRequest.entity = entity
  // 3
  let sortDescriptor = NSSortDescriptor(key: "date", ascending: true)
  fetchRequest.sortDescriptors = [sortDescriptor]
  do {
    // 4
    locations = try managedObjectContext.fetch(fetchRequest)
  } catch {
    fatalCoreDataError(error)
  }
}
```

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. Here you tell 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 `fetch()` method returns an array with the sorted objects, or throws an error in case something went wrong. That's why this happens inside a `do-try-catch` block.

If everything went well, you assign the results of the fetch to the `locations` instance variable.

> **Note:** To create the fetch request you wrote `NSFetchRequest<Location>`.
>
> The `< >` mean that `NSFetchRequest` is a *generic*. Recall that arrays are also generics, because to create an array you specify the type of objects that go into the array, either using the shorthand notation `[Location]`, or the longer `Array<Location>`.
>
> To use an `NSFetchRequest`, you need to tell it what type of objects you're going to be fetching. Here, you create an `NSFetchRequest<Location>` so that the result of `fetch()` is an array of `Location` objects.

Now that you've loaded the list of `Location` objects into an instance variable, you can change the table view's data source methods.

➤ Change the data source methods to:

```
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
  return locations.count
}
```

```
override func tableView(_ tableView: UITableView,
                 cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(
                         withIdentifier: "LocationCell", for: indexPath)

  let location = locations[indexPath.row]

  let descriptionLabel = cell.viewWithTag(100) as! UILabel
  descriptionLabel.text = location.locationDescription

  let addressLabel = cell.viewWithTag(101) as! UILabel
  if let placemark = location.placemark {
    var text = ""
    if let s = placemark.subThoroughfare {
      text += s + " "
    }
    if let s = placemark.thoroughfare {
      text += s + ", "
    }
    if let s = placemark.locality {
      text += s
    }
    addressLabel.text = text
  } else {
```

```
      addressLabel.text = ""
    }
    return cell
  }
```

This should have no surprises for you. You get the `Location` object for the row from the array and then use its properties to fill up the labels. Because `placemark` is an optional, you use `if let` to unwrap it.

➤ Run the app. Now switch to the Locations tab and… crap! It crashes.

The text in the debug area says something like:

```
fatal error: unexpectedly found nil while unwrapping an Optional value
```

Exercise. What did you forget? ■

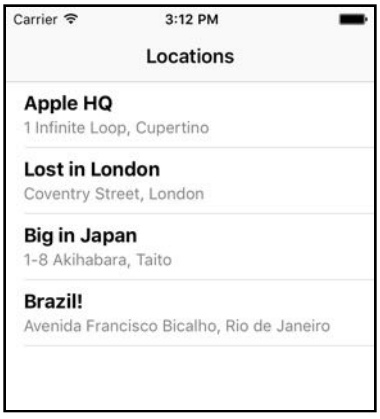
Answer: You added a `managedObjectContext` property to `LocationsViewController`, but never gave this property a value. Therefore, there is nothing to fetch `Location` objects from.

➤ Switch to **AppDelegate.swift**. In `application(didFinishLaunchingWithOptions)`, inside the `if let tabBarViewControllers` block, add the following:

```
let navigationController = tabBarViewControllers[1]
                                                as! UINavigationController
let locationsViewController = navigationController.viewControllers[0]
                                                as! LocationsViewController
locationsViewController.managedObjectContext = 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 using the **Cocoa Touch Class** template. Name it **LocationCell** and make it a subclass of **UITableViewCell**.

➤ Add the following outlets to **LocationCell.swift**, inside the class definition:

```
@IBOutlet weak var descriptionLabel: UILabel!
@IBOutlet weak var addressLabel: UILabel!
```

➤ 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 use the `LocationCell`'s Connections inspector to connect the `descriptionLabel` and `addressLabel` outlets.

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.swift**, replace `tableView(cellForRowAt)` with the following:

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(
        withIdentifier: "LocationCell", for: indexPath) as! LocationCell

  let location = locations[indexPath.row]
  cell.configure(for: location)

  return cell
}
```

As before, this asks for a cell using `dequeueReusableCell(withIdentifier, for)`, but now this will always be a `LocationCell` object instead of a regular `UITableViewCell`. That's why you've added the type cast.

Note that `"LocationCell"` is the re-use identifier from the placeholder cell, but `LocationCell` is the class of the actual cell object that you're getting. They have the same name but one is a `String` and the other is a `UITableViewCell` subclass with extra properties. I hope that's not too confusing.

Once you have the cell reference, you call a new method, `configure(for)` to put the `Location` object into the table view cell.

➤ Add this new method to **LocationCell.swift**:

```
func configure(for location: Location) {
  if location.locationDescription.isEmpty {
    descriptionLabel.text = "(No Description)"
  } else {
    descriptionLabel.text = location.locationDescription
  }

  if let placemark = location.placemark {
    var text = ""
    if let s = placemark.subThoroughfare {
      text += s + " "
    }
    if let s = placemark.thoroughfare {
      text += s + ", "
    }
    if let s = placemark.locality {
      text += s
    }
    addressLabel.text = text
  } else {
    addressLabel.text = String(format:
      "Lat: %.8f, Long: %.8f", location.latitude, location.longitude)
  }
}
```

Instead of using `viewWithTag()` to find the description and address labels, you now simply use the `descriptionLabel` and `addressLabel` properties of the cell.

➤ Run the app to make sure everything still works. If you have a location without a description the table cell will now say "(No Description)". If there is no placemark, the address label contains the GPS coordinates.

Using a custom subclass for your table view cells, there is no limit to how complex you can make them.

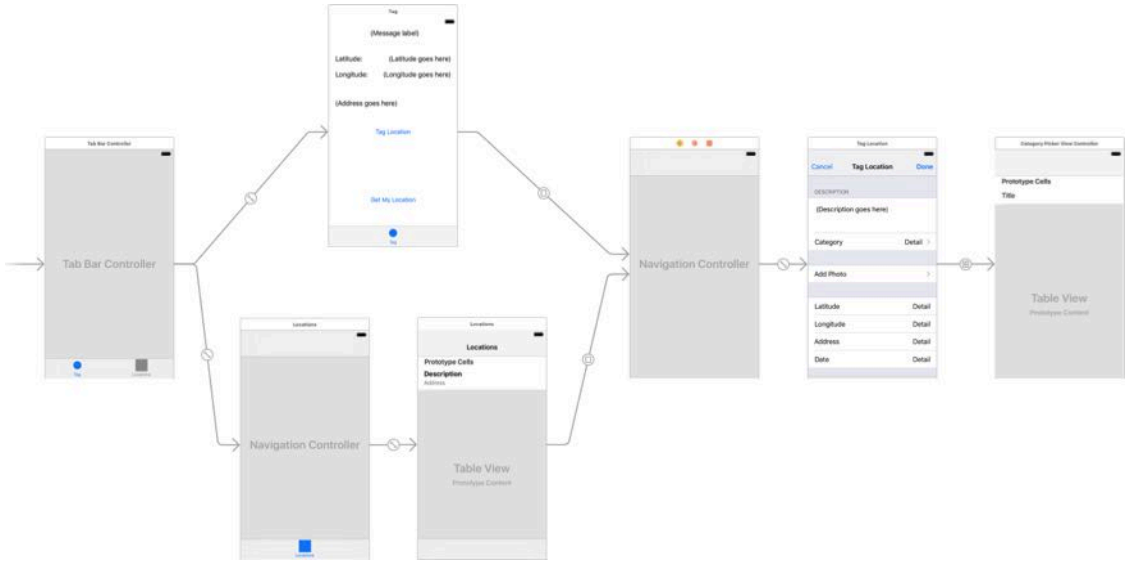
Editing the locations

You will now connect the `LocationsViewController` to the Location Details screen, so that when you tap a row in the table, it lets you edit that location's description and category.

You'll be re-using the `LocationDetailsViewController` but have it edit an existing `Location` object rather than add a new one.

➤ Go to the storyboard. Select the prototype cell (the one with the "Description" and "Address" labels) and **Ctrl-drag** to the Navigation Controller that is connected to the Location Details screen. Add a **Present Modally** selection 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 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.

➤ Go to **LocationsViewController.swift** and add the prepare-for-segue method:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "EditLocation" {
    let navigationController = segue.destination
                                          as! UINavigationController
    let controller = navigationController.topViewController
                                        as! LocationDetailsViewController
    controller.managedObjectContext = managedObjectContext

    if let indexPath = tableView.indexPath(
                                       for: sender as! UITableViewCell) {
      let 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 the row and puts it in the new `locationToEdit` property of `LocationDetailsViewController`. This property doesn't exist yet but

you'll add it in a moment.

The Any type

The type of the `sender` parameter is `Any`. You have seen this type in a few places before. What is it?

Objective-C has a special type, `id`, 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` doesn't have any methods, properties or instance variables, it's a completely naked object reference.

All objects in an Objective-C program can be treated as having type `id`. As a result, a lot of the APIs from the iOS frameworks depend on this special `id` type. This is a powerful feature of Objective-C but unfortunately a dynamic type like `id` doesn't really fit in a *strongly typed* language such as Swift.

Still, we can't avoid `id` completely because it's so prevalent in the iOS frameworks. The Swift equivalent of `id` is the type `Any`.

The `sender` parameter from `prepare(for:sender:)` can be any kind of object and therefore has type `Any` (thanks to the question mark it can also be `nil`).

If the segue is triggered from a table view, `sender` has type `UITableViewCell`. If triggered from a `button`, sender has type `UIBarButtonItem`, and so on.

Objects that appear as type `Any` are not very useful in that form, and you'll have to tell Swift what sort of object it really is. In the code that you just wrote, `indexPath(for)` expects a `UITableViewCell` object, not an `Any` object.

You and I both know that `sender` in this case really is a `UITableViewCell` because the only way to trigger this segue is to tap a table view cell. With the `as!` type cast you're giving Swift your word (Scout's honor!) that it can safely interpret `sender` as a `UITableViewCell`.

(Of course, if you were to hook up this segue to something else, such as a button, then this assumption is no longer valid and the app will crash.)

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 these properties to **LocationDetailsViewController.swift**:

```
var locationToEdit: Location?
var descriptionText = ""
```

`locationToEdit` needs to be an optional because in "adding" mode it will be `nil`. That's how the `LocationDetailsViewController` tells the difference.

➤ Expand `viewDidLoad()` to check whether `locationToEdit` is set:

```
override func viewDidLoad() {
  super.viewDidLoad()

  if let location = locationToEdit {
    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".

Note: Xcode gives a warning on the line `if let location = locationToEdit` because you're not using the value of `location` anywhere. If you click the yellow icon, Xcode suggests that you replace it with `if locationToEdit != nil`. In a little while you *will* use `location`, so ignore Xcode's suggestion.

➤ Also change this line in `viewDidLoad()`:

```
descriptionTextView.text = descriptionText
```

You're loading the value of the new `descriptionText` variable into the text view.

Now how do you get the values from that `Location` object into the text view and labels of this view controller? Swift has a really cool **property observer** feature that is perfect for this.

➤ Change the declaration of the `locationToEdit` property to the following:

```
var locationToEdit: Location? {
  didSet {
    if let location = locationToEdit {
      descriptionText = location.locationDescription
      categoryName = location.category
      date = location.date
```

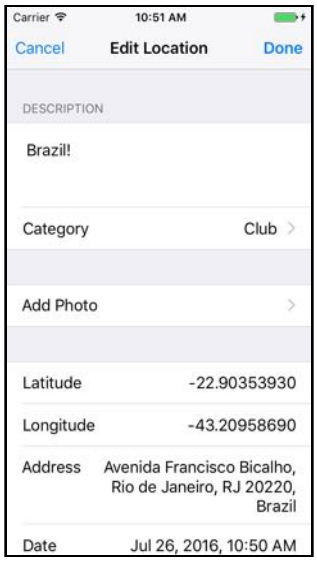
```
        coordinate = CLLocationCoordinate2DMake(
                                    location.latitude, location.longitude)
        placemark = location.placemark
      }
    }
  }
```

If a variable has a `didSet` block, then the code in this block is performed whenever you put a new value into the variable. Very handy!

Here you take the opportunity to fill in the view controller's instance variables with the `Location` object's values.

Because `prepare(for:sender)` – and therefore `locationToEdit`'s `didSet` – is called before `viewDidLoad()`, this puts the right values on the screen when it becomes visible.

➤ 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,
2. the screen doesn't update to reflect any changes in the data model.

The first fix is easy.

➤ Still in **LocationDetailsViewController.swift**, change the top part of `done()`:

```
@IBAction func done() {
  let hudView = HudView.hud(inView: . . .)

  let location: Location
  if let temp = locationToEdit {
    hudView.text = "Updated"
    location = temp
  } else {
    hudView.text = "Tagged"
    location = Location(context: managedObjectContext)
  }

  location.locationDescription = descriptionTextView.text
  . . .
```

The change is 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`.

> **Note:** I've been harping on about the fact that Swift requires all non-optional variables and constants to always have a value. But here you declare `let location` without giving it an initial value. What gives?
>
> Well, the if-statement that follows this declaration always puts a value into `location`, either the unwrapped value of `locationToEdit`, or a new `Location` object obtained from Core Data. After the if-statement, `location` is guaranteed to have a value. Swift is cool with that.

➤ 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: Recall that the table view also doesn't update when you tag new locations. ■

Answer: You fetch the `Location` objects in `viewDidLoad()`. But `viewDidLoad()` is only performed once, 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 better way to do this, and that's the topic of the next section.

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 live 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.swift**, replace the `locations` instance variable with the new `fetchedResultsController` variable:

```
lazy var fetchedResultsController:
                                NSFetchedResultsController<Location> = {
  let fetchRequest = NSFetchRequest<Location>()

  let entity = Location.entity()
  fetchRequest.entity = entity

  let sortDescriptor = NSSortDescriptor(key: "date", ascending: true)
  fetchRequest.sortDescriptors = [sortDescriptor]

  fetchRequest.fetchBatchSize = 20

  let fetchedResultsController = NSFetchedResultsController(
    fetchRequest: fetchRequest,
    managedObjectContext: self.managedObjectContext,
    sectionNameKeyPath: nil,
    cacheName: "Locations")

  fetchedResultsController.delegate = self
  return fetchedResultsController
}()
```

This again uses the lazy initialization pattern with a closure that sets everything up. 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.

The code in the closure does the same thing that you used to do in `viewDidLoad()`: it makes an `NSFetchRequest` and gives it an entity and a sort descriptor.

> **Note:** One small difference is that you need to use `self` here to access the `managedObjectContext` variable. You're inside a closure, after all.
>
> Also note that the new variable is not just `NSFetchedResultsController` but `NSFetchedResultsController<Location>`, since it's a generic. You need to tell the fetched results controller what type of objects to fetch.

This is new:

```
fetchRequest.fetchBatchSize = 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:

```
let fetchedResultsController = NSFetchedResultsController(
  fetchRequest: fetchRequest,
  managedObjectContext: self.managedObjectContext,
  sectionNameKeyPath: nil,
  cacheName: "Locations")
```

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 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.

The line that sets `fetchedResultsController.delegate` to `self` currently gives an error message because `LocationsViewController` does not conform to the right delegate protocol yet. You'll fix that in minute.

Now that you have a fetched results controller, you can simplify `viewDidLoad()`.

➤ Change `viewDidLoad()` to:

```
override func viewDidLoad() {
  super.viewDidLoad()
  performFetch()
}

func performFetch() {
  do {
```

```
    try fetchedResultsController.performFetch()
  } catch {
    fatalCoreDataError(error)
  }
}
```

You still perform the initial fetch in `viewDidLoad()`, using the new `performFetch()` helper method. However, if any `Location` objects change after that initial fetch, the `NSFetchedResultsController`'s delegate methods are called to let you know about these changes. 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 `deinit` method:

```
deinit {
  fetchedResultsController.delegate = nil
}
```

The `deinit` 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. Still, it's good to get into the habit of writing `deinit` methods.)

Because you removed the `locations` array, you should also change the table's data source methods.

➤ Change `tableView(numberOfRowsInSection)` to:

```
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
  let sectionInfo = fetchedResultsController.sections![section]
  return sectionInfo.numberOfObjects
}
```

The fetched results controller's `sections` property returns an array of `NSFetchedResultsSectionInfo` objects that describe each section of the table view. The number of rows is found in the section info's `numberOfObjects` property.

(Currently there is only one section but in a little while you'll split up the locations by category and then each category gets its own section.)

➤ Change `tableView(cellForRowAt)` to:

```
override func tableView(_ tableView: UITableView,
                 cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(
        withIdentifier: "LocationCell", for: indexPath) as! LocationCell
```

```
    let location = fetchedResultsController.object(at: indexPath)
    cell.configure(for: location)

    return cell
  }
```

Instead of looking into the `locations` array like you did before, 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.

➤ Make the same change in `prepare(for:sender:)`.

There is still one piece of the puzzle missing. You need to implement the delegate methods for `NSFetchedResultsController` in `LocationsViewController`. Let's use an *extension* for that, to keep these methods organized.

An extension lets you add code to an existing class, without having to modify that class in the source code. When you make an extension you say, "here are a bunch of extra methods that also need to go into that class", and you can do that even if you didn't write the original class to begin with.

You've seen an extension used in Location+CoreDataProperties.swift. That was done to make it easier for Xcode to regenerate this file without overwriting the contents of Location+CoreDataClass.swift.

You can also use extensions to organize your source code. Here you'll use an extension just for the `NSFetchedResultsControllerDelegate` methods, so they are not all tangled up with `LocationsViewController`'s other code. By putting it in its own thing, it keeps the responsibilities separate.

This makes it easy to spot which part of `LocationsViewController` plays the role of the delegate. All the fetched results controller delegate stuff happens just in this extension, not in the main body of the class. (You could even place this extension in a separate Swift file if you wanted.)

➤ Add the following code to the bottom of the source file, outside of the class:

```
extension LocationsViewController: NSFetchedResultsControllerDelegate {

  func controllerWillChangeContent(_ controller:
                    NSFetchedResultsController<NSFetchRequestResult>) {
    print("*** controllerWillChangeContent")
    tableView.beginUpdates()
  }

  func controller(
      _ controller: NSFetchedResultsController<NSFetchRequestResult>,
      didChange anObject: Any, at indexPath: IndexPath?,
      for type: NSFetchedResultsChangeType, newIndexPath: IndexPath?) {

    switch type {
```

```
      case .insert:
        print("*** NSFetchedResultsChangeInsert (object)")
        tableView.insertRows(at: [newIndexPath!], with: .fade)

      case .delete:
        print("*** NSFetchedResultsChangeDelete (object)")
        tableView.deleteRows(at: [indexPath!], with: .fade)

      case .update:
        print("*** NSFetchedResultsChangeUpdate (object)")
        if let cell = tableView.cellForRow(at: indexPath!)
                                                    as? LocationCell {
          let location = controller.object(at: indexPath!) as! Location
          cell.configure(for: location)
        }

      case .move:
        print("*** NSFetchedResultsChangeMove (object)")
        tableView.deleteRows(at: [indexPath!], with: .fade)
        tableView.insertRows(at: [newIndexPath!], with: .fade)
      }
    }

    func controller(
          _ controller: NSFetchedResultsController<NSFetchRequestResult>,
          didChange sectionInfo: NSFetchedResultsSectionInfo,
          atSectionIndex sectionIndex: Int,
          for type: NSFetchedResultsChangeType) {
      switch type {
      case .insert:
        print("*** NSFetchedResultsChangeInsert (section)")
        tableView.insertSections(IndexSet(integer: sectionIndex),
                                 with: .fade)
      case .delete:
        print("*** NSFetchedResultsChangeDelete (section)")
        tableView.deleteSections(IndexSet(integer: sectionIndex),
                                 with: .fade)
      case .update:
        print("*** NSFetchedResultsChangeUpdate (section)")
      case .move:
        print("*** NSFetchedResultsChangeMove (section)")
      }
    }

    func controllerDidChangeContent(
        _ controller: NSFetchedResultsController<NSFetchRequestResult>) {
      print("*** controllerDidChangeContent")
      tableView.endUpdates()
    }
  }
```

Yowza, that's a lot of code. 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. Look it over for a few minutes to see if this code makes sense to you. You've made it this far, so I'm sure it won't be too hard.

`NSFetchedResultsController` will invoke 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 `print()` 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:

```
*** controllerWillChangeContent
*** NSFetchedResultsChangeUpdate (object)
*** controllerDidChangeContent
```

`NSFetchedResultsController` noticed that an existing object was updated and, through updating the table, called your `cell.configure(for)` 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:

```
*** controllerWillChangeContent
*** NSFetchedResultsChangeInsert (object)
*** controllerDidChangeContent
```

This time it's an "insert" notification. The delegate methods told the table view to do `insertRows(at:with:)` 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 in response.

It doesn't matter where in the app you make these changes, they can happen on any screen. When that screen saves the changes to the managed object context, the fetched results controller picks up on them right away.

"It's not a bug, it's an undocumented feature"

There is a nasty bug with Core Data that still appears to be present in iOS 10. 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.

It's even possible that 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:
```

We did no such thing! Interestingly, this problem does not occur when you switch to the Locations tab before you tag the new location.

There are two possible fixes:

1 - You can delete the cache of the `NSFetchedResultsController`. To do this, add the following line to `viewDidLoad()` before the call to `performFetch()`:

```
NSFetchedResultsController<Location>.deleteCache(withName: "Locations")
```

This is not a great solution because it negates the point of having a cache in the first place.

2 - You can force the `LocationsViewController` to load its view immediately when the app starts up. Without this, it delays loading the view until you switch tabs, causing Core Data to get confused. To apply this fix, add the following to `application(didFinishLaunchingWithOptions)`, immediately below the line that sets `locationsViewController.managedObjectContext`:

```
let _ = locationsViewController.view
```

If this problem also affects you, then implement one of the above solutions (my suggestion is #2). Then throw away DataModel.sqlite and run the app again. Verify that the bug no longer occurs.

iOS is pretty great but unfortunately it's not free of bugs (what software is?). If you encounter what you perceive to be a bug in one of the iOS frameworks, then report it at bugreport.apple.com. Feel free to report this Core Data bug as practice. :-)

Speaking of nasty bugs, iOS 10.0 introduces one that's a doozy. Parts of Core Data were rewritten for Swift 3.0 and I'm afraid some new bugs slipped into the code.

Here's how to reproduce this error:

1. Tag two new locations and give them both the same category.
2. Go to the Locations tab.
3. Tap the first of these new locations and change its category to something else.

You will now get an error like the following:

```
Serious application error . . . Invalid update: invalid number of rows
in section . . .
```

This error message comes from `UITableView`. It expected the app to move a row from one section into another section, but according to the error message that didn't happen. If you trace back through what *did* happen, you'll find that the `NSFetchedResultsController` sent a `NSFetchedResultsChangeUpdate` notification to the delegate (the view controller) instead of a `NSFetchedResultsChangeMove`.

Because `NSFetchedResultsController` sends the wrong message, the app gets confused and crashes. Yikes! This bug is pretty serious. Until this is fixed, I cannot recommend you use `NSFetchedResultsController` in your own apps if you're using multiple sections.

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.swift**:

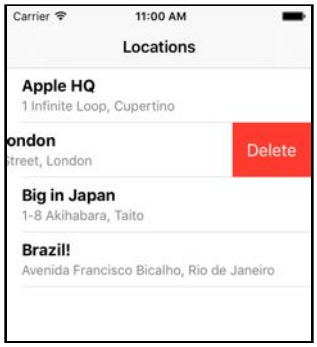
```
override func tableView(_ tableView: UITableView,
                        commit editingStyle: UITableViewCellEditingStyle,
                        forRowAt indexPath: IndexPath) {
  if editingStyle == .delete {
    let location = fetchedResultsController.object(at: indexPath)
    managedObjectContext.delete(location)

    do {
      try managedObjectContext.save()
    } catch {
      fatalCoreDataError(error)
```

```
      }
    }
  }
```

You've seen `tableView(commit, forRowAt)` before. It's part of the table view's data source protocol. As soon as you implement this method in your view controller, it enables 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.



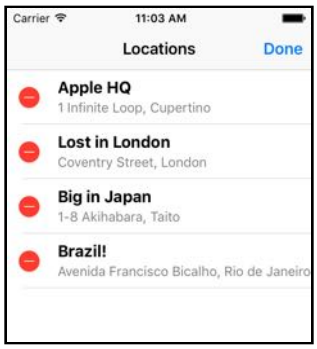
Swipe to delete rows from the table

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()`:

```
navigationItem.rightBarButtonItem = 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:



The table view in edit mode

➤ Run the app and verify that you can now also delete rows by pressing the Edit button.

Pretty sweet, 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` initialization block:

```
lazy var fetchedResultsController: NSFetchedResultsController<...> = {
  . . .
  let sortDescriptor1 = NSSortDescriptor(key: "category",
                                         ascending: true)
  let sortDescriptor2 = NSSortDescriptor(key: "date", ascending: true)
  fetchRequest.sortDescriptors = [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`.

➤ Also change the initialization of the `NSFetchedResultsController` object:

```
  let fetchedResultsController = NSFetchedResultsController(
    fetchRequest: fetchRequest,
    managedObjectContext: self.managedObjectContext,
    sectionNameKeyPath: "category",                    // change this
    cacheName: "Locations")
```

The only difference here is that the `sectionNameKeyPath` parameter changed to `"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:

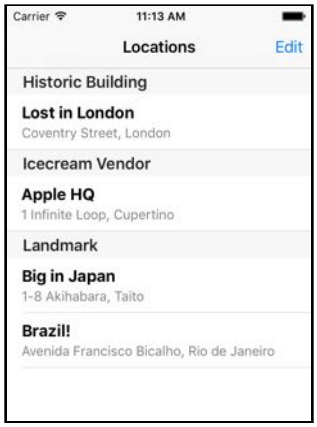
```
override func numberOfSections(in tableView: UITableView) -> Int {
  return fetchedResultsController.sections!.count
}

override func tableView(_ tableView: UITableView,
                        titleForHeaderInSection section: Int) -> String? {
  let sectionInfo = 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 array of `NSFetchedResultsSectionInfo` objects, and then look inside that array to find out how many sections there are and what their names are.

Exercise. Why do you need to write `sections!` with an exclamation point? ■

Answer: the `sections` property is an optional, so it needs to be unwrapped before you can use it. Here you know for sure that `sections` will never be `nil` – after all, you just told `NSFetchedResultsController` to group the search results based on the value of their "category" field – so you can safely force unwrap it using the exclamation mark. Are you starting to get the hang of these optionals already?

➤ 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 **04 – Locations Tab** in the

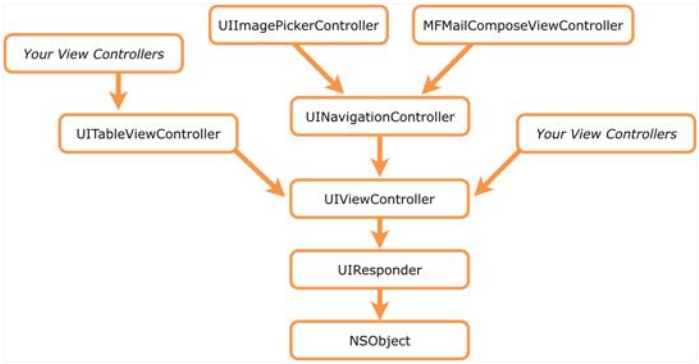
tutorial's Source Code folder.

Hierarchies

Programmers love to organize stuff into hierarchies – or **tree structures** as they like to call them.

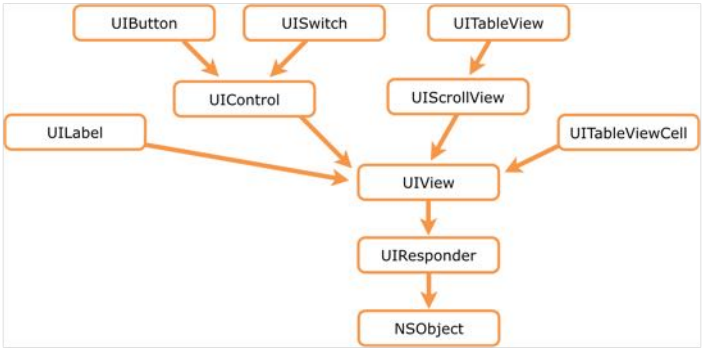
You have to admit that the following pictures do look at bit like trees. The thing at the bottom is often called the root, all the other items are branches. Keeping with the tree analogy, the items at the very ends are known as the leaves.

There is the inheritance hierarchy of view controller classes:



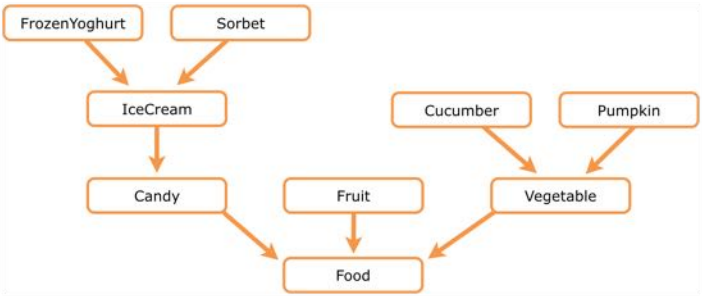
UIViewController inheritance tree (partial)

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.

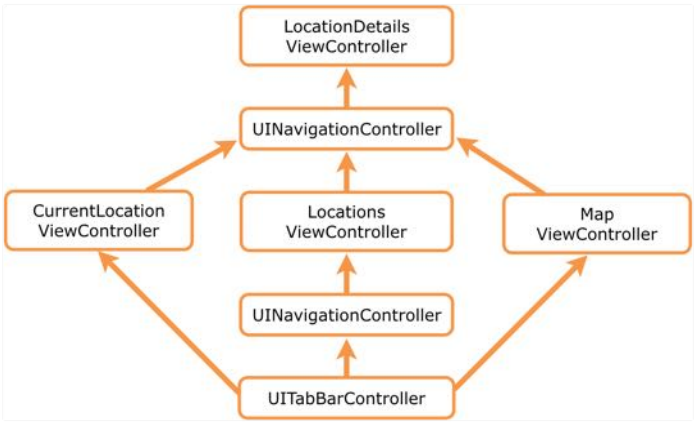
A `UITableViewController` *is* a `UIViewController`, but with extra stuff. Likewise, a `UIButton` is a `UIView` that also knows how to respond to touches and can initiate actions when tapped.

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 `UIControl`, which is the child of `UIView`.

The above hierarchies are between data types. 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 sort of hierarchy between view controllers, with the `UITabBarController` as the root at the bottom:

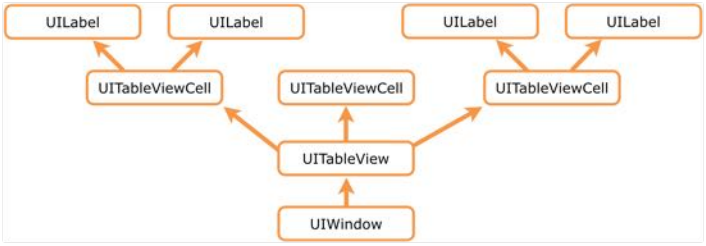


The flow between the screens in the app is also a hierarchy

This is essentially what your storyboard represents. Here the terms parent and child

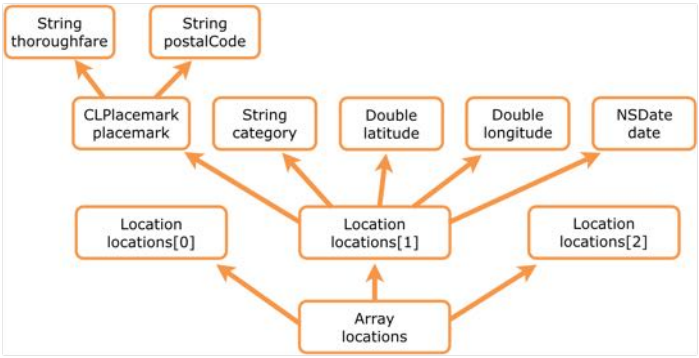
are used also: the `UITabBarController` is the parent of the view controllers for the Tag, Locations and Map screens (which you'll be adding next).

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 `Array` 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.

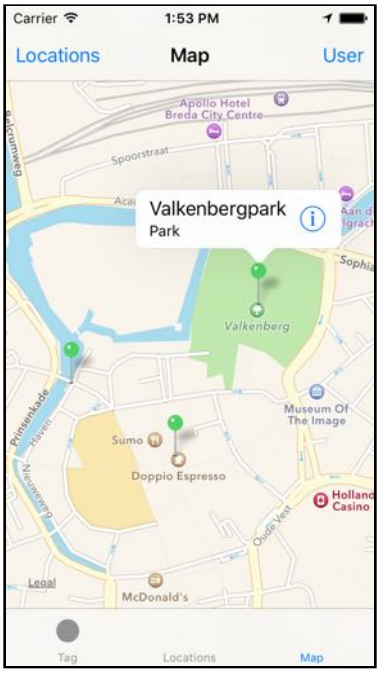
The world of computing is rife with examples of hierarchies and tree structures. For example, the contents of a plist or XML file, the organization of your source files into groups in the Xcode project, the computer's file system with its infinitely nested directories, get-rich-quick Ponzi schemes, you 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 a third tab to the app that looks like this:



The Map screen with some locations in my hometown

First visit: the storyboard.

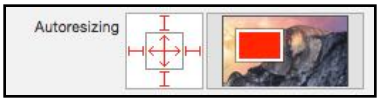
➤ From the Objects Library, drag a regular **View Controller** into the canvas (put it below the others).

➤ Ctrl-drag from the Tab Bar Controller to this new View Controller to add it to the tabs (choose **Relationship segue – view controllers**).

➤ The new view controller now has a **Tab Bar Item**. Rename it to say **Map**.

➤ Drag a **Map Kit View** into the view controller. Make it cover the entire area of the screen, so that it sits partially below the tab bar. (The size of the Map View should be 320 × 568 points.)

➤ In the **Size inspector**, change the autoresizing settings for the Map View to:



The autoresizing settings for the Map View

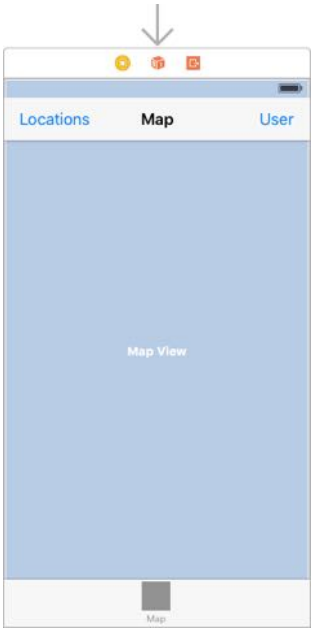
➤ In the **Attributes inspector** for the Map View, enable **Shows: User Location**.

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 at Y = 20, against the bottom of the status bar. Change its title to **Map**. The navigation bar should also partially overlap 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:



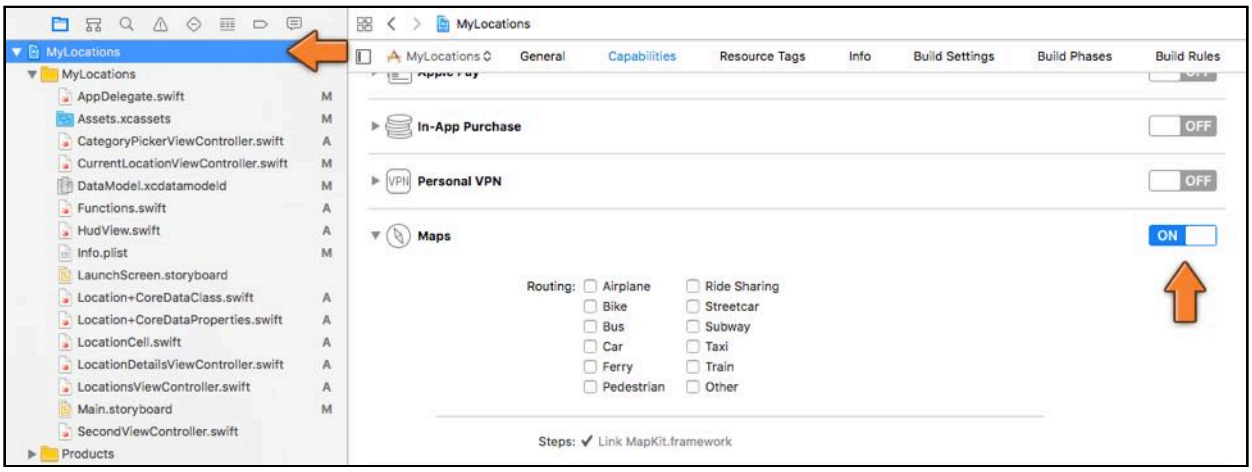
The design of the Map screen

> **Note:** For some reason the Map Kit View appears as a white rectangle on macOS Sierra. The above screenshot is from OS X El Capitan where the Map Kit View is blue and a lot easier to see.

The app won't run as-is. The app will compile without problems but crashes when you switch to the Map screen. `MKMapView` is not part of UIKit but of MapKit, so you need to add the MapKit framework first.

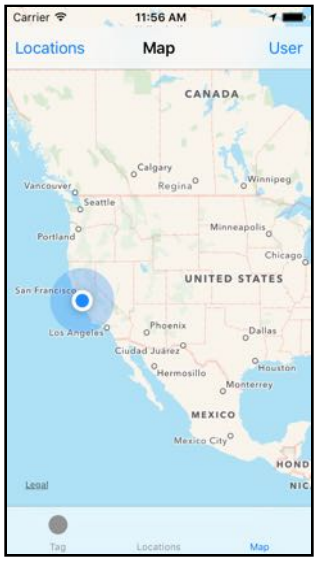
For some reason adding `import MapKit` to the source of the app is not sufficient to solve this problem. An extra step is required.

➤ Go to the **Project Settings** screen and select the **Capabilities** tab. Scroll down to where it says **Maps** and toggle the switch to ON.



Enabling the app to use maps

➤ 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 blue dot shows the current location:



The map shows the user's location

You can pan the map by clicking the mouse and dragging it across the Simulator window. To zoom in or out, hold down the Alt/Option key while dragging the mouse.

> **Note:** There is a gap between the top of the screen and the navigation bar. That happens because the status bar is not 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 that blue dot could be almost anywhere in California!

➤ Add a new source file to the project and name it **MapViewController**.

➤ Replace the contents of **MapViewController.swift** with the following:

```
import UIKit
import MapKit
import CoreData

class MapViewController: UIViewController {
  @IBOutlet weak var mapView: MKMapView!

  var managedObjectContext: NSManagedObjectContext!

  @IBAction func showUser() {
    let region = MKCoordinateRegionMakeWithDistance(
                              mapView.userLocation.coordinate, 1000, 1000)
    mapView.setRegion(mapView.regionThatFits(region), animated: true)
  }

  @IBAction func showLocations() {
  }
}

extension MapViewController: MKMapViewDelegate {
}
```

This is a regular view controller, not a table view controller. It has an outlet 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, courtesy of the extension.

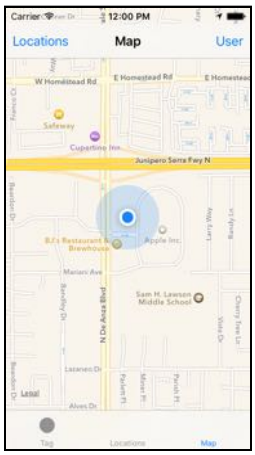
➤ In the storyboard, select the view controller and in the **Identity inspector** set its **Class** to **MapViewController**.

➤ Connect the Locations bar button item to the `showLocations` action and the User button to the `showUser` action. (In case you forgot how, Ctrl-drag from the bar button items to the view controller.)

➤ Connect the Map View with the `mapView` outlet (Ctrl-drag from the view controller to the Map View), and its delegate with the view controller (Ctrl-drag the other way around).

Currently the view controller only implements the `showUser()` action method. 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.

Try it out:



Pressing the User button zooms in to the user's location

The other button, Locations, is going to 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.swift**:

```
var locations = [Location]()
```

➤ Also add the `updateLocations()` method:

```
func updateLocations() {
  mapView.removeAnnotations(locations)

  let entity = Location.entity()

  let fetchRequest = NSFetchRequest<Location>()
  fetchRequest.entity = entity

  locations = try! managedObjectContext.fetch(fetchRequest)
  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.

You've seen how to handle errors with a `do–try–catch` block. If you're certain that a particular method call will never fail, you can dispense with the `do` and `catch` and just write `try!` with an exclamation point. As with other things in Swift that have exclamation points, if it turns out that you were wrong, the app will crash without mercy. But in this case there isn't much that can go wrong so you can choose to

live a little more dangerously.

Once you've obtained the `Location` objects, you call `mapView.addAnnotations()` 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 when this happens the `locations` array may already exist and may contain `Location` objects. If so, you first remove the pins for these old objects with `removeAnnotations()`.

Xcode says the lines with `mapView.addAnnotations()` and `removeAnnotations()` have errors. This is to be expected and you'll fix it in a minute.

➤ First, add the `viewDidLoad()` method:

```
override func viewDidLoad() {
  super.viewDidLoad()
  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 reference to that object first. As before, that happens in `AppDelegate`.

➤ In **AppDelegate.swift**, extend `application(didFinishLaunchingWithOptions)` to pass the context object to the `MapViewController` as well. This goes inside the `if let` statement:

```
let mapViewController = tabBarViewControllers[2] as! MapViewController
mapViewController.managedObjectContext = managedObjectContext
```

You're not quite done yet. In `updateLocations()` you told 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 `class` line from **Location+CoreDataClass.swift** to:

```
public class 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!

Exercise. Xcode now says "Use of undeclared type MKAnnotation". Why is that? ■

Answer: You still need to `import MapKit`. Add that line at the top of the file.

Exercise. Xcode still gives a compiler error when you try to build the project. What

is wrong now? ■

Answer: Because you said `Location` conforms to the protocol `MKAnnotation`, you have to provide all the required features from that protocol in the `Location` class.

The `MKAnnotation` protocol requires the class to implement 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 code to **Location+CoreDataClass.swift**:

```
var coordinate: CLLocationCoordinate2D {
  return CLLocationCoordinate2DMake(latitude, longitude)
}

var title: String? {
  if locationDescription.isEmpty {
    return "(No Description)"
  } else {
    return locationDescription
  }
}

var subtitle: String? {
  return category
}
```

Do you notice anything special here? All three items are instance variables (because of `var`), but they also have a block of source code associated with them.

These variables are **read-only computed properties**. That means they don't actually store a value into a memory location. Whenever you access the `coordinate`, `title`, or `subtitle` variables, they perform the logic from their code blocks. That's why they are *computed* properties: they compute something.

These properties are read-only because they only return a value – you can't give them a new value using the assignment operator.

The following is OK because it reads the value of the property:

```
let s = location.title
```

But you cannot do this:

```
location.title = "Time for a change"
```

The only way the `title` property can change is if the `locationDescription` value changes. You could also have written this as a method:

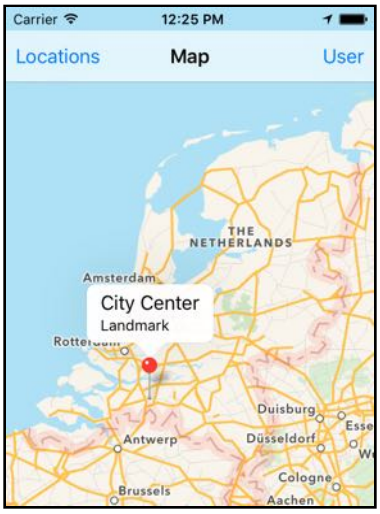
```
func title() -> String? {
```

```
    if locationDescription.isEmpty {
      return "(No Description)"
    } else {
      return locationDescription
    }
  }
```

This is equivalent to using the computed property. Whether to use a method or a computed property is often a matter of taste and you'll see both ways used throughout the iOS frameworks.

(By the way, it is also possible to make *read-write* computed properties that *can* be changed, but the `MKAnnotation` protocol doesn't use those.)

➤ Run the app and switch to the Map screen. It should now show pins for all 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

> **Note:** So far all the protocols you've seen were used for making delegates, but that's not the case here. `Location` is not a delegate of anything.
>
> The `MKAnnotation` protocol simply lets you pretend that `Location` is an annotation that can be placed on a map view. You can use this trick with any object you want; as long as the object implements the `MKAnnotation` protocol it can be shown on a map.
>
> Protocols let objects wear different hats.

Pressing the User button makes the app zoom to the user's current coordinates 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.swift**, add a new method, `region(for)`:

```
func region(for annotations: [MKAnnotation]) -> MKCoordinateRegion {
  let region: MKCoordinateRegion

  switch annotations.count {
  case 0:
    region = MKCoordinateRegionMakeWithDistance(
                            mapView.userLocation.coordinate, 1000, 1000)

  case 1:
    let annotation = annotations[annotations.count - 1]
    region = MKCoordinateRegionMakeWithDistance(
                                      annotation.coordinate, 1000, 1000)

  default:
    var topLeftCoord = CLLocationCoordinate2D(latitude: -90,
                                              longitude: 180)
    var bottomRightCoord = CLLocationCoordinate2D(latitude: 90,
                                                  longitude: -180)

    for annotation in annotations {
      topLeftCoord.latitude = max(topLeftCoord.latitude,
                                  annotation.coordinate.latitude)
      topLeftCoord.longitude = min(topLeftCoord.longitude,
                                   annotation.coordinate.longitude)
      bottomRightCoord.latitude = min(bottomRightCoord.latitude,
                                      annotation.coordinate.latitude)
      bottomRightCoord.longitude = max(bottomRightCoord.longitude,
                                       annotation.coordinate.longitude)
    }

    let center = CLLocationCoordinate2D(
      latitude: topLeftCoord.latitude -
               (topLeftCoord.latitude - bottomRightCoord.latitude) / 2,
      longitude: topLeftCoord.longitude -
              (topLeftCoord.longitude - bottomRightCoord.longitude) / 2)

    let extraSpace = 1.1
    let span = MKCoordinateSpan(
      latitudeDelta: abs(topLeftCoord.latitude -
                         bottomRightCoord.latitude) * extraSpace,
      longitudeDelta: abs(topLeftCoord.longitude -
                          bottomRightCoord.longitude) * extraSpace)

    region = MKCoordinateRegion(center: center, span: span)
  }

  return mapView.regionThatFits(region)
}
```

`region(for)` has three situations to handle. It uses a `switch` statement to look at the number of annotations and then chooses the corresponding case:

- There are no annotations. You'll center the map on the user's current position.
- There is only one annotation. You'll center the map on that one annotation.
- There are two or more annotations. You'll calculate the extent of their reach and add a little padding. See if you can make sense of those calculations. The `max()` function looks at two values and returns the larger of the two; `min()` returns the smaller; `abs()` always makes a number positive (absolute value).

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 `region(for)` is concerned, what it deals with are annotations. It just so happens that these annotations are described by your `Location` objects.

That is the power of using protocols. It also allows you to use this method in any app that uses Map Kit, without modifications. Pretty neat.

➤ Change the `showLocations()` action method to:

```
@IBAction func showLocations() {
  let theRegion = region(for: locations)
  mapView.setRegion(theRegion, animated: true)
}
```

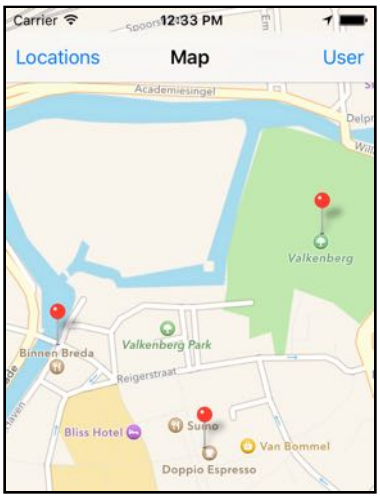
This calls `region(for)` to calculate a reasonable region that fits all the `Location` objects and then sets that region on the map view.

➤ Finally, change `viewDidLoad()`:

```
override func viewDidLoad() {
  super.viewDidLoad()
  updateLocations()

  if !locations.isEmpty {
    showLocations()
  }
}
```

It's a good idea to show the user's locations the first time you switch to the Map tab, so `viewDidLoad()` calls `showLocations()` too.

➤ 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

Making your own pins

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.

➤ Add the following code to **MapViewController.swift**, inside the extension at the bottom:

```
func mapView(_ mapView: MKMapView,
             viewFor annotation: MKAnnotation) -> MKAnnotationView? {
  // 1
  guard annotation is Location else {
    return nil
  }
  // 2
  let identifier = "Location"
  var annotationView = mapView.dequeueReusableAnnotationView(
                                          withIdentifier: identifier)
  if annotationView == nil {
    let pinView = MKPinAnnotationView(annotation: annotation,
                                      reuseIdentifier: identifier)
    // 3
    pinView.isEnabled = true
    pinView.canShowCallout = true
    pinView.animatesDrop = false
    pinView.pinTintColor = UIColor(red: 0.32, green: 0.82,
                                   blue: 0.4, alpha: 1)

    // 4
    let rightButton = UIButton(type: .detailDisclosure)
    rightButton.addTarget(self,
                          action: #selector(showLocationDetails),
                          for: .touchUpInside)
    pinView.rightCalloutAccessoryView = rightButton
```

```
      annotationView = pinView
    }

    if let annotationView = annotationView {
      annotationView.annotation = annotation

      // 5
      let button = annotationView.rightCalloutAccessoryView as! UIButton
      if let index = locations.index(of: annotation as! Location) {
        button.tag = index
      }
    }

    return annotationView
  }
```

This is very similar to what a table view data source does in "`cellForRowAt`", 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 apart from 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 special "`is`" type check operator to determine whether the annotation is really a `Location` object. If it isn't, you return `nil` to signal that you're not making an annotation for this other kind of object. The `guard` statement you're using here works like an `if`: it only continues if the condition – `annotation is Location` – is true.

2. This looks similar 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 using `MKPinAnnotationView` for your annotations. 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 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 where it gets interesting. 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 a new method `showLocationDetails()`, 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

`showLocationDetails()` when the button is pressed.

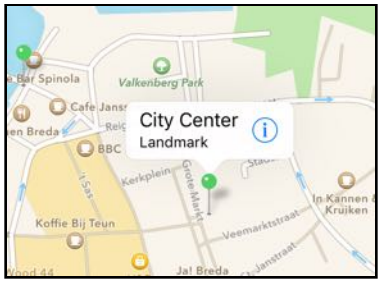
➤ Add the `showLocationDetails()` method but leave it empty for now. Put it inside the main class, not the extension.

```
func showLocationDetails(_ sender: UIButton) {
}
```

Because you've told the button its `#selector` is `showLocationDetails`, the app won't compile unless you add at least an empty version of this method.

The method takes one parameter, `sender`, that refers to the control that sent the action message. In this case the sender will be the (i) button. That's why the type of the `sender` parameter is `UIButton`.

➤ 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 in the storyboard.)



The annotations use your own view

Guard

In the map view delegate method, you wrote the following:

```
guard annotation is Location else {
  return nil
}
```

The `guard` statement lets you try something. If the result is `nil` or `false`, the code from the `else` block is performed.

If everything works like it's supposed to, the code simply skips the `else` block and continues.

You could also have written it as follows:

```
if annotation is Location {
```

```
    // do all the other things
    . . .
  } else {
    return nil
  }
```

This uses the familiar `if` statement. But notice how the code that handles the situation when `annotation` is *not* a `Location` is now all the way at the bottom of the method. If you have several of these if-statements, your code ends up looking like this:

```
if condition1 {
  if condition2 {
    if condition3 {
      . . .
    } else {
      return nil  // condition3 is false
    }
  } else {
    return nil    // condition2 is false
  }
} else {
  return nil      // condition1 is false
}
```

This kind of structure is known as the "Pyramid of Doom". There's nothing wrong with it per se, but it can make the program flow hard to decipher. With `guard` you can write this as:

```
guard condition1 else {
  return nil             // condition1 is false
}
guard condition2 else {
  return nil             // condition2 is false
}
guard condition3 else {
  return nil             // condition3 is false
}
. . .
```

Now all the conditions are checked first and any errors or unexpected situations are handled straight away. Many programmers find this easier to read.

Tapping a pin on the map now brings up a callout with a blue (i) button. What should this button do? Show the Edit Location screen, of course!

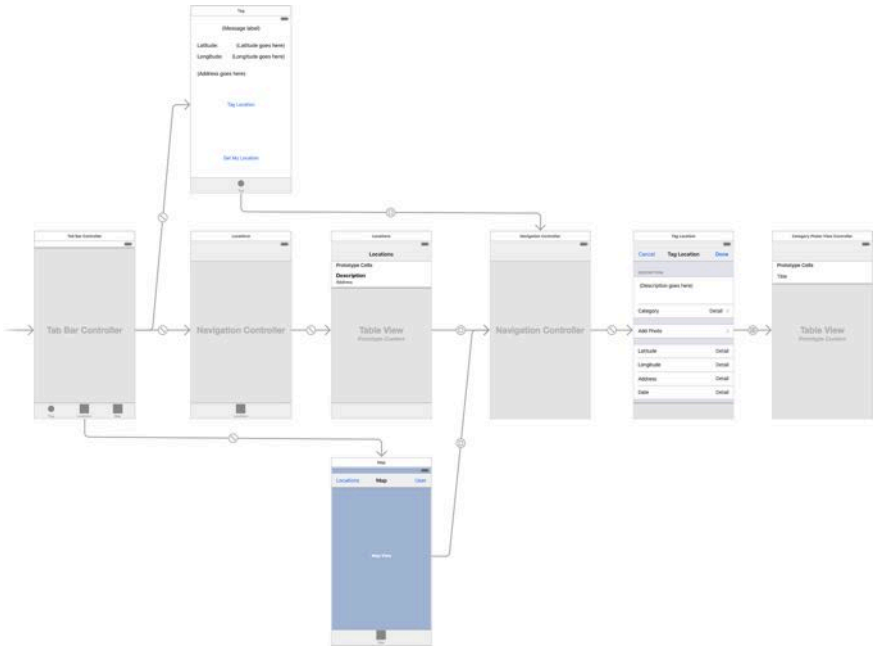
➤ Open the storyboard. Select the Map View Controller (the actual view controller, not some view inside it) and **Ctrl-drag** to the Navigation Controller that contains

the Location Details View Controller.

Make this a new **Present Modally** segue named **EditLocation**.

Tip: If making this connection gives you problems because the storyboard won't fit on your screen, then try Ctrl-dragging from the outline pane. You can also zoom out and make the segue.

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 (or at least to its navigation controller).

➤ Back in **MapViewController.swift**, change the `showLocationDetails()` method to trigger the segue:

```
func showLocationDetails(sender: UIButton) {
  performSegue(withIdentifier: "EditLocation", sender: sender)
}
```

Because the segue isn't connected to any particular control in the view controller, you have to perform it manually. You pass along the button object as the `sender`, so you can read its `tag` property in *prepare-for-segue*.

➤ Add the `prepare(for:sender:)` method:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "EditLocation" {
```

```
      let navigationController = segue.destination
                                              as! UINavigationController

      let controller = navigationController.topViewController
                                           as! LocationDetailsViewController

      controller.managedObjectContext = managedObjectContext

      let button = sender as! UIButton
      let 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 `sender` button as the index in that array.

➤ Run the app, tap on a pin and edit the location.

It works, except that the annotation's callout doesn't change until you tap the pin again. 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 that happen afterwards are overlooked.

The way you're going to fix this for the Map screen is by using notifications. Recall that you have already put `NotificationCenter` to use for dealing with Core Data save errors.

As it happens, Core Data also sends out a bunch of notifications when changes are made to the data store. You can subscribe to these notifications and update the map view when you receive them.

➤ In **MapViewController.swift**, change the `managedObjectContext` property declaration to:

```
  var managedObjectContext: NSManagedObjectContext! {
    didSet {
      NotificationCenter.default.addObserver(forName:
               Notification.Name.NSManagedObjectContextObjectsDidChange,
               object: managedObjectContext,
               queue: OperationQueue.main) { notification in
        if self.isViewLoaded {
          self.updateLocations()
        }
      }
    }
  }
```

This is another example of a property observer put to good use.

As soon as `managedObjectContext` is given a value – which happens in `AppDelegate` during app startup – the `didSet` block tells the `NotificationCenter` to add an observer for the `NSManagedObjectContextObjectsDidChange` notification.

This notification with the very long name is sent out by the `managedObjectContext` whenever the data store changes. In response you would like the following closure to be called. For clarity, here's what happens in the closure:

```
{ notification in
  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 use `if self.isViewLoaded` to make sure `updateLocations()` only gets called when the map view is loaded.

Because this screen sits in a tab, the view from `MapViewController` does not actually get loaded from the storyboard until the user switches to the Map tab.

So the view may not have been loaded yet when the user tags a new location. In that case it makes no sense to call `updateLocations()` – it could even crash the app because the `MKMapView` object doesn't exist yet at that point!

➤ Run the app. First go to the Map screen to see your existing location pins. 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.

Have another look at that closure. The "`notification in`" bit is the parameter for the closure. Like functions and methods, closures can take parameters.

Because this particular closure gets called by `NotificationCenter`, you're given an `Notification` object in the `notification` parameter. Since you're not using this `notification` object anywhere in the closure, you could also write it like this:

```
{ _ in
  if self.isViewLoaded {
    self.updateLocations()
  }
}
```

You've already seen the _ underscore used in a few places in the code. This symbol is called the **wildcard** and you can use it whenever a name is expected but you don't really care about it.

Here, the _ tells Swift you're not interested in the closure's parameter. It also helps to reduce visual clutter in the source code; it's obvious at a glance that this parameter – whatever it may be – isn't being used in the closure.

So whenever you see the _ used in Swift source code it just means, "there's something here but the programmer has chosen to ignore it".

Exercise. The `Notification` object has a `userInfo` dictionary. From that dictionary it is possible to figure out which objects were inserted/deleted/updated. For example, use the following `print()`s to examine this dictionary:

```
if let dictionary = notification.userInfo {
  print(dictionary["inserted"])
  print(dictionary["deleted"])
  print(dictionary["updated"])
}
```

This will print out an (optional) array of `Location` objects or `nil` if there were no changes. Your mission, should you choose to accept it: 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.

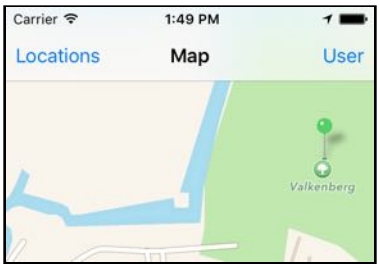
➤ Add a new extension at the bottom of **MapViewController.swift**:

```
extension MapViewController: UINavigationBarDelegate {
  func position(for bar: UIBarPositioning) -> UIBarPosition {
    return .topAttached
  }
}
```

This tells the navigation bar to extend under the status bar area.

➤ Finally, in the storyboard **Ctrl-drag** from the navigation bar to the view controller to make the delegate connection.

Now the gap between the navigation bar and the top of the screen is gone:



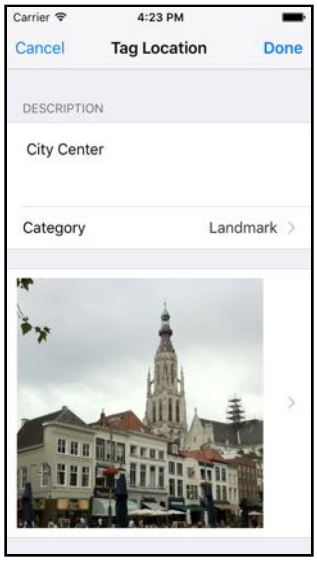
The navigation bar extends to the top of the screen

You can find the project files for the app up to this point under **05 - 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 save the photo along with the location so the user has a nice picture to look at.



A photo in the Tag Location screen

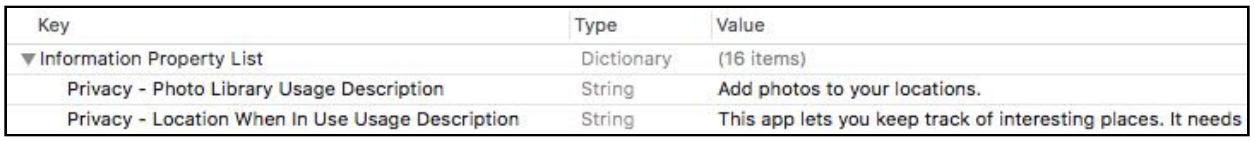
Just as you need to ask the user for permission before you can read the device's GPS coordinates, so does accessing the user's photo library.

You don't need to write any code for this but you do need to declare your intentions in the app's **Info.plist**. If you don't do this, the app will crash as soon as you try to use the `UIImagePickerController`.

➤ Open **Info.plist** and add a new row (either right-click and select **Add Row** or use the **Editor → Add Item** menu).

For the key, use **NSPhotoLibraryUsageDescription**, or choose **Privacy - Photo Library Usage Description** from the list.

For the value, type: **Add photos to your locations.**



Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
Privacy - Photo Library Usage Description	String	Add photos to your locations.
Privacy - Location When In Use Usage Description	String	This app lets you keep track of interesting places. It needs

Adding a usage description in Info.plist

Now when the app opens the photo picker for the first time, iOS will tell the user what the app intends to use the photos for, using the description you just added to Info.plist.

➤ In **LocationDetailsViewController.swift**, add the following extension to the bottom of the source file:

```
extension LocationDetailsViewController:
      UIImagePickerControllerDelegate, UINavigationControllerDelegate {

  func takePhotoWithCamera() {
    let imagePicker = UIImagePickerController()
    imagePicker.sourceType = .camera
    imagePicker.delegate = self
    imagePicker.allowsEditing = true
    present(imagePicker, animated: true, completion: nil)
  }
}
```

The `UIImagePickerController` is a view controller like any other, but it is built into 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.)

Note: You're doing this in an extension because it allows you to group all the photo-picking related functionality together.

If you wanted to, you could put these methods into the main class body. That would work fine too, but view controllers tend to become very big with many methods that all do different things.

As a way to preserve your sanity, it's nice to extract conceptually related methods – such as everything that has to do with picking photos – and place them together in their own extension.

You could even move each of these extensions to their own source file, for example "LocationDetailsViewController+PhotoPicking.swift".

➤ Add the following methods to the extension:

```
func imagePickerController(_ picker: UIImagePickerController,
                    didFinishPickingMediaWithInfo info: [String : Any]) {

  dismiss(animated: true, completion: nil)
}

func imagePickerControllerDidCancel(_ picker: UIImagePickerController) {
  dismiss(animated: true, completion: nil)
}
```

Currently these delegate methods simply remove the image picker from the screen. Soon you'll take the image the user picked and add it to the `Location` object, but for now you just want to make the image picker show up.

Note that the view controller (in this case the extension) must conform to both `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate` for this to work, but you don't have to implement any of the `UINavigationControllerDelegate` methods.

➤ Now change `tableView(didSelectRowAt)` in the class:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  if indexPath.section == 0 && indexPath.row == 0 {
    . . .
  } else if indexPath.section == 1 && indexPath.row == 0 {
    takePhotoWithCamera()
  }
}
```

Add Photo is the first row in the second section. When it's tapped, you call the `takePhotoWithCamera()` method that you just added.

➤ Run the app, tag a new location or edit an existing one, and press Add Photo.

If you're running the app in the Simulator, then bam! It crashes. The error message is this:

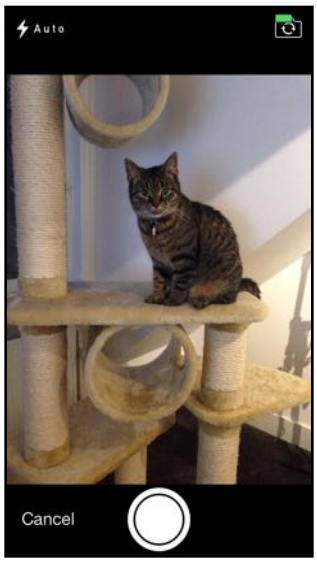
```
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: 'Source type 1 not available'
```

The culprit for the crash is the line:

```
imagePicker.sourceType = .camera
```

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 to the extension:

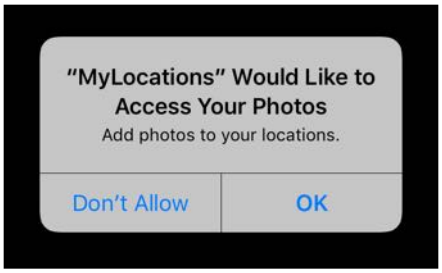
```
func choosePhotoFromLibrary() {
  let imagePicker = UIImagePickerController()
  imagePicker.sourceType = .photoLibrary
  imagePicker.delegate = self
  imagePicker.allowsEditing = true
  present(imagePicker, animated: true, completion: nil)
}
```

These two methods do essentially the same, except now you set the `sourceType` to `.photoLibrary`.

➤ Change "`didSelectRowAt`" to call `choosePhotoFromLibrary()` instead of `takePhotoWithCamera()`.

➤ Run the app in the Simulator and press Add Photo.

First, the user needs to give the app permission to access the photo library:



The user needs to allow the app access to 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** to allow the app to use the photo library.

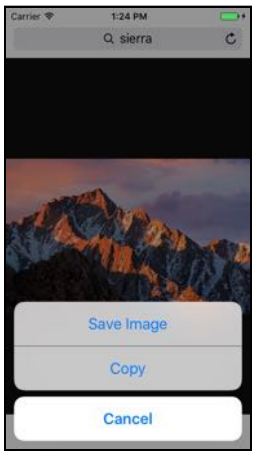
At this point you might see this:

There are no photos in the library

If this happens, the simulator's photo library has no photos in it yet. It's also possible you already see a handful of stock images; this differs between Xcode versions.

➤ If you don't see any images, stop the app and click on the built-in **Photos** app in the Simulator. This should display a handful of sample photos. Run the app again and try picking a photo. You may or may not see these sample photos now. If not, you'll have to add your own.

There are several 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 an image file on top of the Simulator window. This adds the picture to your library in the Photos app.

Finally, you can use the Terminal and the `simctl` command. Type all on one line:

```
/Applications/Xcode.app/Contents/Developer/usr/bin/simctl addphoto booted
~/Desktop/MyPhoto.JPG
```

The `simctl` tool can be used to manage your Simulators (type `simctl help` for a list of options). The command "`addphoto booted`" adds the specified image to the active Simulator's photo library.

➤ 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 `true`. With this setting enabled, the user can do some quick editing on the photo before making his final choice. (In the Simulator you can hold down Alt/Option while dragging to rotate and zoom the photo.)

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 little smarter and allow the user to choose the camera when it is 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 following methods to **LocationDetailsViewController.swift**, in the photo picking extension:

```
func pickPhoto() {
  if UIImagePickerController.isSourceTypeAvailable(.camera) {
    showPhotoMenu()
  } else {
    choosePhotoFromLibrary()
  }
}

func showPhotoMenu() {
  let alertController = UIAlertController(title: nil, message: nil,
                                          preferredStyle: .actionSheet)

  let cancelAction = UIAlertAction(title: "Cancel", style: .cancel,
                                   handler: nil)
  alertController.addAction(cancelAction)

  let takePhotoAction = UIAlertAction(title: "Take Photo",
                                      style: .default, handler: nil)
  alertController.addAction(takePhotoAction)
```

```
    let chooseFromLibraryAction = UIAlertAction(title:
                    "Choose From Library", style: .default, handler: nil)
    alertController.addAction(chooseFromLibraryAction)

    present(alertController, animated: true, completion: nil)
  }
```

You're using `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 `UIAlertController` on the screen.

Unlike the alert controllers you've used before, this one has the `.actionSheet` style. An action sheet works very much like an alert view, except that it slides in from the bottom of the screen.

➤ In "`didSelectRowAt`", change the call to `choosePhotoFromLibrary()` to `pickPhoto()` instead. This is the last time you're changing this line, honest.

➤ 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 the action sheet but doesn't do anything else yet.

By the way, if you want to test this action sheet on the Simulator, then you can fake the availability of the camera by writing the following in `pickPhoto()`:

```
  if true || UIImagePickerController.isSourceTypeAvailable(.camera) {
```

That will always show the action sheet because the condition is now always true.

The choices in the action sheet are provided by `UIAlertAction` objects. The `handler:` parameter determines what happens when you press the corresponding button in the action sheet.

Right now the handlers for all three choices – Take Photo, Choose From Library, Cancel – are `nil`, so nothing will happen.

➤ Change these lines to the following:

```
let takePhotoAction = UIAlertAction(title: "Take Photo",
          style: .default, handler: { _ in self.takePhotoWithCamera() })
```

```
let chooseFromLibraryAction = UIAlertAction(
                          title: "Choose From Library", style: .default,
                          handler: { _ in self.choosePhotoFromLibrary() })
```

This gives `handler:` a closure that calls the corresponding method from the extension. You use the _ wildcard to ignore the parameter that is passed to this closure (which is a reference to the `UIAlertAction` itself).

➤ Run the app make sure the buttons from the action sheet work properly.

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.

Notice that the Add Photo cell remains selected (dark gray background) when you cancel the action sheet. That doesn't look so good.

➤ In `tableView(didSelectRowAt)`, add the following line before the call to `pickPhoto()`:

```
tableView.deselectRow(at: indexPath, animated: true)
```

This first deselects 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.

By the way, if you still have the Core Data debug output enabled, then you should see a whole bunch of output in the Xcode Debug Area when the image picker is active. Apparently the `UIImagePickerController` uses Core Data as well!

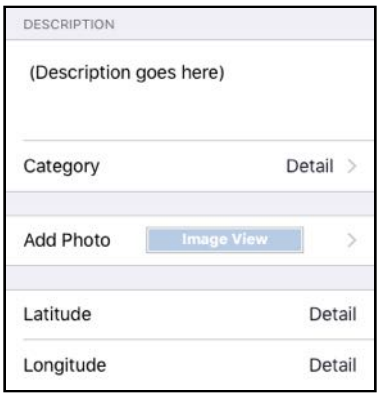
Showing the image

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 the class in **LocationDetailsViewController.swift**:

```
@IBOutlet weak var imageView: UIImageView!
@IBOutlet weak var addPhotoLabel: UILabel!
```

➤ 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

> **Note:** On macOS Sierra the image view appears as a white rectangle instead of a blue rectangle.

➤ Connect the Image View to the view controller's `imageView` outlet. Also connect the Add Photo label to the `addPhotoLabel` 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.swift**:

```
var image: UIImage?
```

If no photo is picked yet, `image` is `nil`, so this must be an optional.

➤ Add the `show(image)` method to the class:

```
func show(image: UIImage) {
  imageView.image = image
  imageView.isHidden = false
  imageView.frame = CGRect(x: 10, y: 10, width: 260, height: 260)
  addPhotoLabel.isHidden = true
}
```

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 `imagePickerController(didFinishPickingMediaWithInfo)` method from the photo picking extension to the following:

```
func imagePickerController(_ picker: UIImagePickerController,
                   didFinishPickingMediaWithInfo info: [String : Any]) {
```

```
    image = info[UIImagePickerControllerEditedImage] as? UIImage

    if let theImage = image {
      show(image: theImage)
    }

    dismiss(animated: true, completion: nil)
  }
```

This is the method that gets called when the user has selected a photo in the image picker.

You can tell by the notation `[String : Any]` that the `info` parameter is a dictionary. Whenever you see `[ A : B ]` you're dealing with a dictionary that has keys of type "A" and values of type "B".

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 user moved and/or scaled it. (You can also get the original image if you wish, using a different key.)

Once you have the photo, you store it in the `image` instance variable so you can use it later.

Dictionaries always return optionals, because there is a theoretical possibility that the key you asked for – `UIImagePickerControllerEditedImage` in this case – doesn't actually exist in the dictionary.

Under normal circumstances you'd unwrap this optional but here the `image` instance variable is an optional itself so no unwrapping is necessary.

If `info[UIImagePickerControllerEditedImage]` is `nil`, then `image` will be `nil` too. You do need to cast the value from the meaningless `Any` to `UIImage` using the `as?` operator. In this case you need to use the optional cast, `as?` instead of `as!`, because `image` is an optional instance variable.

Once you have the image and it is not `nil`, the call to `show(image)` puts it in the Add Photo cell.

Exercise. See if you can rewrite the above logic to use a `didSet` property observer on the `image` instance variable. If you succeed, then placing the photo into image will automatically update the `UIImageView`, without needing to call `show(image)`. ■

➤ Run the app and choose a photo. Whoops, it looks like you have a small problem here:

The photo gets cut off

(It's also possible that the photo overlaps the rows below it. In any case, it doesn't look good…)

The `show(image)` 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 "`heightForRowAt`" table view method to make the table view cell resize.

➤ Change the `tableView(heightForRowAt)` method:

```
override func tableView(_ tableView: UITableView,
                        heightForRowAt indexPath: IndexPath) -> CGFloat {
  if indexPath.section == 0 && indexPath.row == 0 {
    return 88

  } else if indexPath.section == 1 {           // this else if is new
    if imageView.isHidden {
      return 44
    } else {
      return 280
    }

  } else if indexPath.section == 2 && indexPath.row == 2 {
    . . .
```

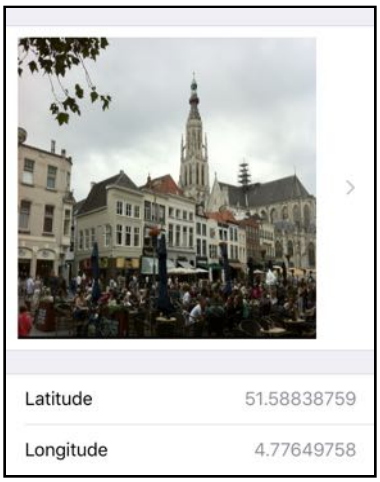
If there is no image, then the height for the Add Photo cell is 44 points just like a regular cell. But if there is an image, it's a lot higher: 280 points. That is 260 points for the image view plus 10 points margin on the top and bottom.

➤ Add the next line to `imagePickerController(didFinishPickingMediaWithInfo)`, just before you dismiss the view controller:

```
tableView.reloadData()
```

This refreshes the table view and sets the photo row to the proper height.

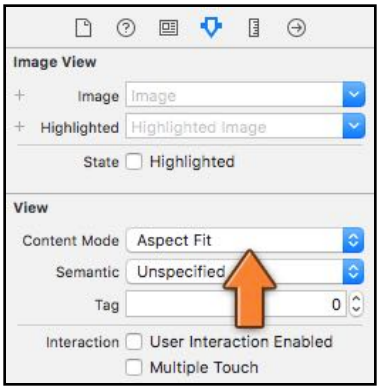
➤ Try it out. The cell now resizes and is big enough for the whole photo. The image does appear to be stretched out a little, though.



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 **Content Mode** to **Aspect Fit**.



Changing the image view's content mode

This will keep the image's aspect ratio intact as it is 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 get the aspect ratio 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! You can find solutions from other readers at forums.raywenderlich.com ■

By the way, notice how the if-statements in `tableView(heightForRowAt)` all look at the index-path's section and/or row?

```
if indexPath.section == 0 && indexPath.row == 0 {
  . . .
} else if indexPath.section == 1 {
  . . .
} else if indexPath.section == 2 && indexPath.row == 2 {
  . . .
} else {
  . . .
}
```

Whenever you see `if – else if – else if – else` where the conditions all check the same thing, it's a good opportunity to use a `switch` statement instead.

➤ Change the `tableView(heightForRowAt)` method to:

```
override func tableView(_ tableView: UITableView,
                        heightForRowAt indexPath: IndexPath) -> CGFloat {

  switch (indexPath.section, indexPath.row) {
    case (0, 0):
      return 88

    case (1, _):
      return imageView.isHidden ? 44 : 280

    case (2, 2):
```

```
      addressLabel.frame.size = CGSize(
                                    width: view.bounds.size.width - 115,
                                    height: 10000)
      addressLabel.sizeToFit()
      addressLabel.frame.origin.x = view.bounds.size.width -
                                      addressLabel.frame.size.width - 15
      return addressLabel.frame.size.height + 20

    default:
      return 44
  }
}
```

The logic inside each of the sections is the same as before, but now the different cases are easier to distinguish:

```
switch (indexPath.section, indexPath.row) {
  case (0, 0):
  case (1, _):
  case (2, 2):
  default:
}
```

This `switch` statement puts `indexPath.section` and `indexPath.row` into a **tuple**, and then uses *pattern matching* to look for the different cases:

- `case (0, 0)` corresponds to section 0, row 0.
- `case (1, _)` corresponds to section 1, any row. The _ is the wildcard again, which means any value for `indexPath.row` is accepted here.
- `case (2, 2)` corresponds to section 2, row 2.
- The `default` case is for any other rows in sections 0 and 2.

Using `switch` is very common in Swift because it makes large blocks of `if – else if` statements much easier to read.

> **Note:** A tuple is nothing more than a list of values inside `( )` parentheses. For example, `(10, 3.14, "Hello")` is a tuple with three elements.
>
> Tuples have various uses, such as allowing a method to return more than one value (simply put the different values into a tuple and return that). They are also very convenient in `switch` statements.

There was another change. The following lines have changed from this,

```
if imageView.isHidden {
  return 44
} else {
  return 280
}
```

into this:

```
return imageView.hidden ? 44 : 280
```

The `? :` construct is the **ternary conditional** operator. It works like an `if – else` statement compressed into a single line.

If the thing before the `?` is true (`imageView.isHidden`) it returns the first value, 44. If false, it returns the second value, 280.

Using `? :` is often simpler than writing it out as `if – else`.

Be careful: there must be a space between `imageView.isHidden` and `?` or else Swift thinks `isHidden` is an optional that you're trying to unwrap, which results in an error. This is a case where the same symbol, `?`, can mean more than one thing.

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 remove any alert or action sheet 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 will have forgotten what they were going to do. The presence of the alert or action sheet is confusing and the user 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 "going to the background" notifications through `NotificationCenter` 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 `UIApplicationDidEnterBackground` notification.

➤ In **LocationDetailsViewController.swift**, add a new method:

```
func listenForBackgroundNotification() {
  NotificationCenter.default.addObserver(
             forName: Notification.Name.UIApplicationDidEnterBackground,
             object: nil, queue: OperationQueue.main) { _ in
```

```
      if self.presentedViewController != nil {
        self.dismiss(animated: false, completion: nil)
      }

      self.descriptionTextView.resignFirstResponder()
    }
  }
```

This adds an observer for the `UIApplicationDidEnterBackground` notification. When this notification is received, `NotificationCenter` will call the closure.

(Notice that you're using the "trailing" closure syntax here; the closure is not a parameter to `addObserver(forName, …)` but immediately follows the method call.)

If there is an active image picker or action sheet, you dismiss it. You also hide the keyboard if the text view was active.

The image picker and action sheet are both presented as modal view controllers that lie on top of everything else. If such a modal view controller is active, `UIViewController`'s `presentedViewController` property has a reference to that modal view controller.

So if `presentedViewController` is not `nil` you call `dismiss()` to close the modal screen. (By the way, this has no effect on the category picker; that does not use a modal segue but a push segue.)

➤ Call the `listenForBackgroundNotification()` method from within `viewDidLoad()`.

➤ Try it out. Open the image picker (or the action sheet if you're on a device that has a camera) and exit to the home screen to put the app to sleep.

Then tap the app's icon to activate the app again. You should now be back on the Tag Location screen (or Edit Location screen if you opted to edit an existing one). The image picker (or action sheet) has automatically closed.

Cool, that seems to work.

There's one more thing to do. You should tell the `NotificationCenter` to stop sending these background notifications when the Tag/Edit Location screen closes. You don't want `NotificationCenter` to send notifications to an object that no longer exists, that's just asking for trouble! The `deinit` method is a good place to tear this down.

➤ First, add a new instance variable:

```
  var observer: Any!
```

This will hold a reference to the observer, which is necessary to unregister it later.

The type of this variable is `Any!`, meaning that you don't really care what sort of object this is. (The exclamation point is used for its regular purpose, to make this

an implicitly unwrapped optional.)

➤ In `listenForBackgroundNotification()`, change the first line so that it stores the return value of `addObserver(forName, …)` into this new instance variable:

```
func listenForBackgroundNotification() {
  observer = NotificationCenter.default.addObserver(forName: . . .
```

➤ Finally, add the `deinit` method:

```
deinit {
  print("*** deinit \(self)")
  NotificationCenter.default.removeObserver(observer)
}
```

You're also adding a `print()` here so you can see some proof that the view controller really does get destroyed when you close the Tag/Edit Location screen.

➤ Run the app, edit an existing location, and tap Done to close the screen.

I don't know about you but I don't see the `*** deinit` message anywhere in the Xcode debug pane.

Guess what? The `LocationDetailsViewController` doesn't get destroyed for some reason. That means the app is leaking memory… Of course, this was all a big setup on my part so I can tell you about closures and capturing.

Remember that in closures you always have to specify `self` when you want to access an instance variable or call a method? That is because **closures** capture any variables that are used inside the closure.

When it captures a variable, the closure simply stores a reference to that variable. This allows it to use the variable at some later point when the closure is actually performed.

Why is this important? If the code inside the closure uses a local variable, the method that created this variable may no longer be active by the time the closure is performed. After all, when a method ends all locals are destroyed. But when such a local is captured by a closure, it stays alive until the closure is also done with it.

Because the closure needs to keep the objects from those captured variables alive in the time between capturing and actually performing the closure, it stores a *strong* reference to those objects. In other words, capturing means the closure becomes a shared owner of the captured objects.

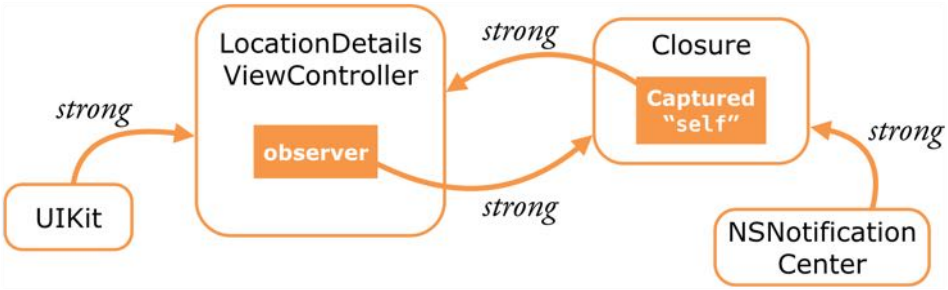
What may not be immediately obvious is that `self` is also one of those variables and therefore gets captured by the closure. Sneaky! That's why Swift requires you to explicitly write out `self` inside closures, so you won't forget this value is being captured.

Because in `self` is the `LocationDetailsViewController`, as the closure captures `self`

it creates a strong reference to the `LocationDetailsViewController` object, and the closure becomes a co-owner of this view controller. I bet you didn't expect that!

Remember, as long as an object has owners it is being kept alive. So this closure is keeping the view controller alive, even after you closed it!

This is known as an **ownership cycle**, because the view controller itself has a strong reference back to the closure through the `observer` variable.



The relationship between the view controller and the closure

(In case you're wondering, the view controller's other owner is UIKit. The observer is also being kept alive by `NotificationCenter`.)

This sounds like a classic Catch-22 problem! Fortunately, there is a way to break the ownership cycle. You can give the closure a so-called **capture list**.

➤ Change `listenForBackgroundNotification()` to the following:

```
func listenForBackgroundNotification() {
  observer = NotificationCenter.default.addObserver(
             forName: Notification.Name.UIApplicationDidEnterBackground,
             object: nil, queue: OperationQueue.main) {
    [weak self] _ in

    if let strongSelf = self {
      if strongSelf.presentedViewController != nil {
        strongSelf.dismiss(animated: false, completion: nil)
      }
      strongSelf.descriptionTextView.resignFirstResponder()
    }
  }
}
```

There are a couple of new things here. Let's look at the first part of the closure:

```
{ [weak self] _ in
  . . .
}
```

The `[weak self]` bit is the capture list for the closure. It tells the closure that the variable `self` will still be captured, but as a weak reference, not strong. As a result,

the closure no longer keeps the view controller alive.

Weak references are allowed to become `nil`, which means the captured `self` is now an optional inside the closure. You need to unwrap it with `if let` before you can send messages to the view controller.

Other than that, the closure still does the exact same things as before.

➤ Try it out. Open the Tag/Edit Location screen and close it again. You should now see the `print()` from `deinit` in the Xcode debug pane.

That means the view controller gets destroyed properly and the notification observer is removed from `NotificationCenter`. Good riddance!

Exercise. What happens if you remove the call to `removeObserver()` from `deinit`? Hint: add `print(self)` inside the closure. ■

Answer: Because the observer is not removed, it says alive and active. The next time you put the app in the background, even if you're not on the Tag/Edit Location screen, the closure from this "old" observer is called again but `self` is now `nil` (the object that it captured no longer exists).

This may seem innocuous but it's a serious bug. Every time the user opens and closes the Tag/Edit Location screen you end up with a new observer that stays in memory forever. The `if let` prevents the app from crashing on a `nil` dereference of `self` as you go to the background, but over time all these leftover observers will eat up the app's available memory.

That's why it's always a good idea to clean up after yourself. Use `print()`'s to make sure your objects really get deallocated! (Xcode also comes with Instruments, a handy tool that you can use to detect such issues.)

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.

> **Note:** Core Data has an "Allows external storage" feature that is designed to make this process completely transparent for the developer. In theory, you can put data of any size into your entities and Core Data automatically decides whether to put the data into the SQLite database or store it as an external file.
>
> Unfortunately, this feature doesn't work very well in practice. Last time I checked it just had too many bugs to be useful. So until this part of Core Data becomes rock solid, we'll be doing it by hand.

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. Besides, the user may have resized or cropped 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 need a way to associate that JPEG file with your `Location` object. 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 image 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**. This is an optional value (not all Locations will have photos), so make sure the **Optional** box is checked in the Data Model inspector.

➤ Add a property for this new attribute to **Location+CoreDataProperties.swift**:

```
@NSManaged var photoID: NSNumber?
```

Remember that for an object that is managed by Core Data, you have to declare the property as `@NSManaged`.

You may be wondering why you're declaring the type of `photoID` as `NSNumber` and not as `Int` (or more precisely `Int32`). Remember that Core Data is an Objective-C framework, so you're limited by the possibilities of that language. `NSNumber` is how number objects are handled in Objective-C.

For various reasons you can't represent an `Int` value as an optional in Objective-C, so instead you'll use the `NSNumber` class. Swift will automatically convert between `Int` values and this `NSNumber`, so it's no big deal.

You'll now add some other properties to the `Location` object to make working with the photo file a little easier.

➤ Add the `hasPhoto` computed property to **Location+CoreDataClass.swift**:

```
var hasPhoto: Bool {
  return photoID != nil
}
```

This determines whether the `Location` object has a photo associated with it or not. Swift's optionals make this easy.

➤ Also add the `photoURL` property:

```
var photoURL: URL {
```

```
    assert(photoID != nil, "No photo ID set")
    let filename = "Photo-\(photoID!.intValue).jpg"
    return applicationDocumentsDirectory.appendingPathComponent(filename)
  }
```

This property computes the full URL to the JPEG file for the photo. Recall that iOS uses URLs to refer to files, even those saved on the local device instead of on the internet.

You'll save these JPEG files inside the app's Documents directory. To get the URL to that directory, it uses the global variable `applicationDocumentsDirectory` that you added to Functions.swift earlier.

Notice the use of `assert()` to make sure the `photoID` is not `nil`. An **assertion** is a special debugging tool that is used to check that your code always does something valid. If not, the app will crash with a helpful error message. You'll see more of this later when we talk about finding bugs – and squashing them.

Assertions are a form of defensive programming. Most of the crashes you've seen so far were actually caused by assertions in UIKit. They allow the app to crash in a controlled manner. Without these assertions, programming mistakes could crash the app at random moments, making it very hard to find out what went wrong.

If the app were to ask a `Location` object for its `photoURL` without having given it a valid `photoID` earlier, the app will crash with the message "No photo ID set". If so, 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 `assert()` in strategic places to catch yourself making programming errors.

➤ Add the `photoImage` property:

```
  var photoImage: UIImage? {
    return UIImage(contentsOfFile: photoURL.path)
  }
```

This method returns a `UIImage` object by loading the image file. You'll need this later to show the photos for existing `Location` objects.

Note that this property has the optional type `UIImage?` – that's because loading the image may fail if the file is damaged or removed. Of course, that *shouldn't* happen, but no doubt you've heard of Murphy's Law… It's good to get into the habit of defensive programming.

There is one more thing to add, the `nextPhotoID()` method. This is a class method, meaning that you don't need to have a `Location` instance to call it. You can call this method anytime from anywhere.

➤ Add the `nextPhotoID()` method:

```
class func nextPhotoID() -> Int {
  let userDefaults = UserDefaults.standard
  let currentID = userDefaults.integer(forKey: "PhotoID")
  userDefaults.set(currentID + 1, forKey: "PhotoID")
  userDefaults.synchronize()
  return currentID
}
```

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 `UserDefaults` and update it every time the app asks for a new ID. (This is similar to what you did in the last tutorial to make `ChecklistItem` IDs for use with local notifications.)

It may seem a little silly to use `UserDefaults` for this when you're already using Core Data as the data store, but with `UserDefaults` 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/Edit Location screen and fill in the `Location` object's `photoID` field. This happens in the Location Details View Controller's `done()` action.

➤ In **LocationDetailsViewController.swift**, in the `done()` method, add the following in between where you set the properties of the `Location` object and where you save the managed object context:

```
if let image = image {
  // 1
  if !location.hasPhoto {
    location.photoID = Location.nextPhotoID() as NSNumber
  }
  // 2
  if let data = UIImageJPEGRepresentation(image, 0.5) {
    // 3
    do {
      try data.write(to: location.photoURL, options: .atomic)
    } catch {
      print("Error writing file: \(error)")
    }
  }
}
```

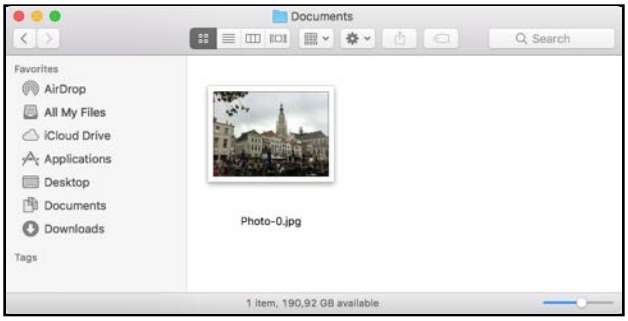
This code is only performed if `image` is not `nil`, in other words, when the user has picked a photo.

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 simply keep the same ID and overwrite the existing JPEG file.

2. The `UIImageJPEGRepresentation()` function converts the `UIImage` into the JPEG format and returns a `Data` object. `Data` is an object that represents a blob of binary data, usually the contents of a file.

3. Here you save the `Data` object to the path given by the `photoURL` property. (Also notice the use of the `do-try-catch` pattern again.)

➤ 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 JPEG file.



The photo is saved in the app's Documents folder

> **Note:** The first time you run the app after adding a new attribute to the data model (photoID), the `NSPersistentContainer` performs a migration of the data store behind the scenes to make sure the data store is in sync again with the data model. If this for some reason doesn't work for you, then remove the old DataModel.sqlite file from the Library/Application Support folder and try again (or simply reset the Simulator or remove the app from your test device).

➤ Tag another location and add a photo to it. Hmm… if you look into the app's Documents directory, this seems to have overwritten the previous photo.

Exercise. Try to debug this one on your own. What is going wrong here? This is a tough one! ■

Answer: When you create a new `Location` object, its `photoID` property gets a default value of 0. That means each `Location` initially has a `photoID` of 0. That should really be `nil`, which means "no photo".

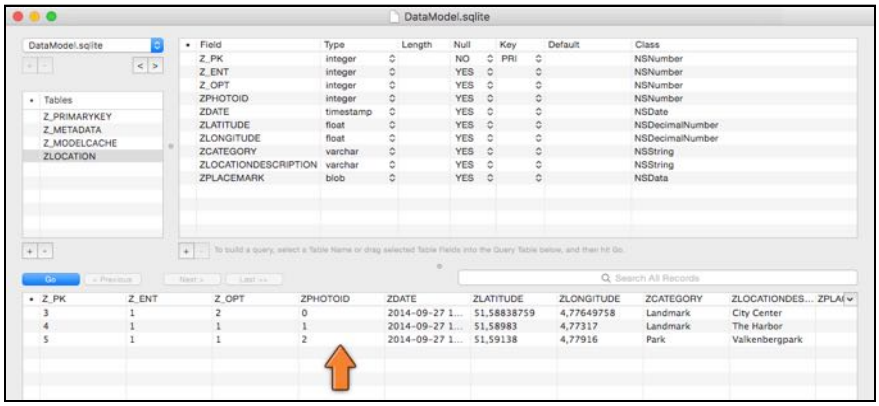
➤ In **LocationDetailsViewController.swift**, add the following line near the top of `done()`:

```
@IBAction func done() {
  . . .

  if let temp = locationToEdit {
    . . .
  } else {
    hudView.text = "Tagged"
    location = Location(context: managedObjectContext)
    location.photoID = nil                                    // add this
  }
  . . .
```

You now give new `Location` objects a `photoID` of `nil` so that the `hasPhoto` property 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.swift** to:

```
override func viewDidLoad() {
  super.viewDidLoad()

  if let location = locationToEdit {
    title = "Edit Location"
    if location.hasPhoto {
      if let theImage = location.photoImage {
        show(image: theImage)
      }
```

```
    }
  }
  . . .
```

If the `Location` that you're editing has a photo, this calls `show(image)` to display it in the photo cell.

Recall that the `photoImage` property returns an optional, `UIImage?`, so you use `if let` to unwrap it. This is another bit of defensive programming.

Sure, if `hasPhoto` is `true` there should always be a valid image file present. But it's possible to imagine a scenario where there isn't – the JPEG file could have been erased or corrupted – even though that "should" never happen. (I'm sure you've had your own share of computer gremlins eating important files.)

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 out 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 `done()` 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 photo file when the `Location` object is deleted.

➤ First add a new method to **Location+CoreDataClass.swift**:

```
func removePhotoFile() {
  if hasPhoto {
    do {
      try FileManager.default.removeItem(at: photoURL)
    } catch {
      print("Error removing file: \(error)")
    }
  }
}
```

This is a code snippet that you can use to remove any file or folder. The `FileManager` class has all kinds of useful methods for dealing with the file system.

➤ Deleting locations happens in **LocationsViewController.swift**. Add the following line to `tableView(commit, forRowAt)`:

```
override func tableView(_ tableView: UITableView,
                        commit editingStyle: UITableViewCellEditingStyle,
                        forRowAt indexPath: IndexPath) {
  if editingStyle == .delete {
    let location = fetchedResultsController.object(at: indexPath)

    location.removePhotoFile()              // add this line
    managedObjectContext.delete(location)
    . . .
```

The new line calls `removePhotoFile()` on the `Location` object just before it is deleted from the Core Data context.

➤ Try it out. Add a new location and give it a photo. You should see the JPEG file in the Documents directory.

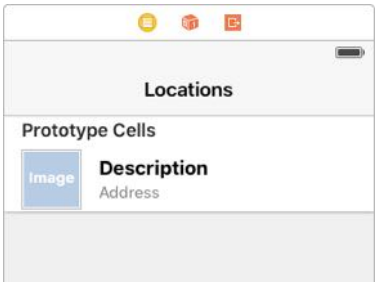
From the Locations screen, delete the location you just added and look in the Documents directory to make sure the JPEG file truly is a goner.

Thumbnails of the photos

Now that locations can have photos, it's a good idea to show thumbnails for these photos in the Locations tab. That will liven up this screen a little… a plain 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.



The table view cell has an image view

➤ Connect the image view to a new `UIImageView` outlet on `LocationCell`, named **photoImageView**.

Exercise. Make this connection with the Assistant editor. Tip: you should connect the image view to the cell, not to the view controller. ■

Now you can put any image into the table view cell simply by placing it inside the image view from `LocationCell`'s `photoImageView` property.

➤ Go to **LocationCell.swift** and add the following method:

```
func thumbnail(for location: Location) -> UIImage {
  if location.hasPhoto, let image = location.photoImage {
    return image
  }
  return UIImage()
}
```

This returns either the image from the `Location` or an empty placeholder image.

You should read this if-statement as, "if the location has a photo, and I can unwrap `location.photoImage`, then return the unwrapped image."

You can also write this as two nested if-statements:

```
if location.hasPhoto {
  if let image = location.photoImage {
    return image
  }
}
```

but because this is such a common thing to do, Swift lets you combine these two things inside a single `if`.

You have also seen that `&&` ("logical and") is used to combine two conditions, but you cannot write it like this:

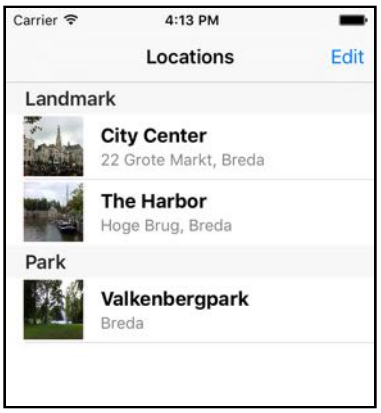
```
if location.hasPhoto && let image = location.photoImage
```

The `&&` only works if both conditions are booleans, but here you're unwrapping an optional as well. In that case you must combine the two conditions with a comma.

➤ Call this new method from the bottom of `configure(for)`:

```
photoImageView.image = thumbnail(for: location)
```

➤ Try it out. The Locations tab should now look something like this:



Images in the Locations table view

You've got thumbnails, all right!

But look closely and you'll see that the images are a little squashed again. That's because you didn't set the Aspect Fit content mode on the image view – but there's a bigger problem here. Literally.

These photos are potentially huge (2592 by 1936 pixels or more), even though the image view is only 52 pixels square. To make them fit, 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.

And what better way to do that than to use an extension?

Extensions

So far you've used extensions on your view controllers to group related functionality together, such as delegate methods. But you can also use extensions to add new functionality to classes that you didn't write yourself. That includes classes from the iOS frameworks, such as `UIImage`.

If you ever catch yourself thinking, "Gee, I wish object X had such-or-so method" then you can probably give it that method by making an extension.

Suppose you want `String` to have a method for adding random words to the string. You could add the `addRandomWord()` method to `String` as follows.

First you create a new source file, for example **String+RandomWord.swift**. It would look like this:

```
import Foundation

extension String {
  func addRandomWord() -> String {
    let value = arc4random_uniform(3)

    let word: String
    switch value {
    case 0:
      word = "rabbit"
    case 1:
      word = "banana"
    case 2:
      word = "boat"
    default:
      word = ""
    }

    return self + word
  }
}
```

Anywhere in your code you can now call `addRandomWord()` on any `String` value:

```
let someString = "Hello, "
let result = someString.addRandomWord()
print("The queen says: \(result)")
```

Extensions 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 extensions are often a cleaner solution.

Besides new methods you can also add new computed properties, but you can't add regular instance variables. You can also use extensions on types that don't even allow inheritance, such as `structs` and `enums`.

You are going to add an extension to `UIImage` that lets you resize the image. You'll use it as follows:

```
return image.resizedImage(withBounds: CGSize(width: 52, height: 52))
```

The `resizedImage(withBounds)` 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 extension.

➤ Add a new file to the project and choose the **Swift File** template. Name the file **UIImage+Resize.swift**.

➤ Replace the contents of this new file with:

```
import UIKit

extension UIImage {
  func resizedImage(withBounds bounds: CGSize) -> UIImage {
    let horizontalRatio = bounds.width / size.width
    let verticalRatio = bounds.height / size.height
    let ratio = min(horizontalRatio, verticalRatio)
    let newSize = CGSize(width: size.width * ratio,
                         height: size.height * ratio)

    UIGraphicsBeginImageContextWithOptions(newSize, true, 0)
    draw(in: CGRect(origin: CGPoint.zero, size: newSize))
    let 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.

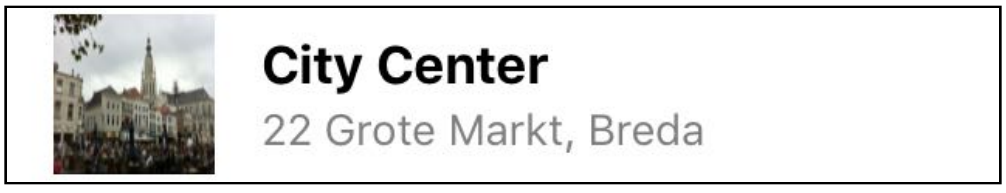
Then it creates a new image context and draws the image into that. We haven't really dealt with graphics contexts before, but they are an important concept in Core Graphics (it has nothing to do with the managed object context from Core Data, even though they're both called "context").

Lets put this extension in action.

➤ Switch to **LocationCell.swift**. Update the `thumbnail(for)` method:

```
func thumbnail(for location: Location) -> UIImage {
  if location.hasPhoto, let image = location.photoImage {
    return image.resizedImage(withBounds: CGSize(width: 52, height: 52))
  }
  return UIImage()
}
```

➤ Run the app. The thumbnails look like this:



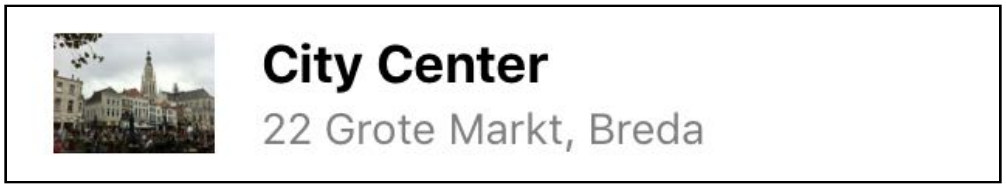
The photos are shrunk to the size of the thumbnails

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 **Content 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

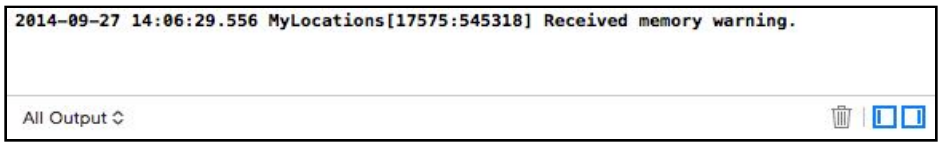
Exercise. Change the resizing function in the `UIImage` extension to resize using the "Aspect Fill" rules instead of the "Aspect Fit" rules. Both keep the aspect ratio intact but Aspect Fit keeps the entire image visible while Aspect Fill fills up the entire rectangle and may cut off parts of the sides. 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 the 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 respond to memory warnings by overriding the `didReceiveMemoryWarning()` method in your view controllers to free up any memory you no longer need. This is often done for things that can easily be recalculated or recreated 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.

For MyLocations there's not much that you 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.

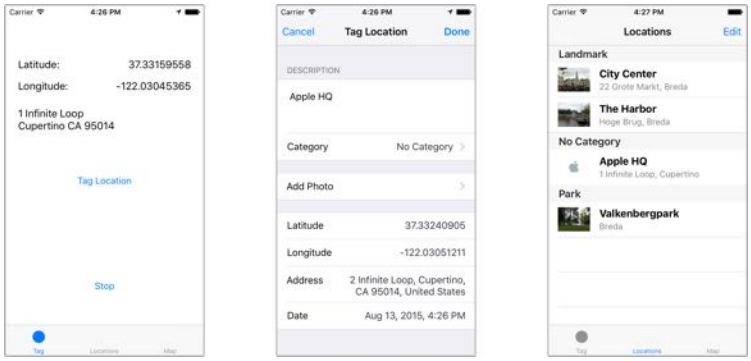
You can find the project files for the app up to this point under **06 – Photo Picker** in the tutorial's Source Code folder.

Making the app look good

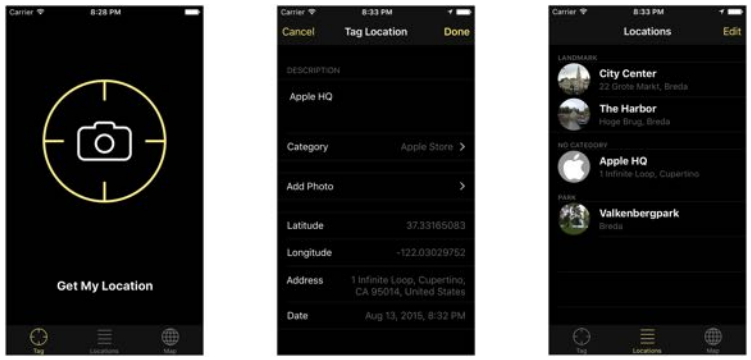
Apps with appealing visuals sell better than ugly ones. Usually I don't wait with the

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,



to this:



The main screen gets the biggest makeover, but you'll also tweak the others a little.

Converting placemarks to strings

Let's begin by improving the code. I'm not really happy with the way the reverse geocoded street address gets converted from a `CLPlacemark` object into a string. It works but the code is unwieldy and repetitive.

There are three places where this happens:

- `CurrentLocationViewController`, the main screen
- `LocationDetailsViewController`, the Tag/Edit Location screen
- `LocationsViewController`, the list of saved locations

Let's start with the main screen. **CurrentLocationViewController.swift** has a method named `string(from)` where this conversion happens. It currently does this:

```
func string(from placemark: CLPlacemark) -> String {
  var line1 = ""

  if let s = placemark.subThoroughfare {
    line1 += s + " "
  }
  if let s = placemark.thoroughfare {
    line1 += s
  }
  . . .
```

It's supposed to return 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 properties may be `nil`, so the code has to be smart enough to skip the empty ones – that's what all the `if lets` are for.

What I don't like is that 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:

```
func add(text: String?, toLine line: String,
         separatedBy separator: String) -> String {
  var result = line
  if let text = text {
    if !line.isEmpty {
      result += separator
    }
    result += text
  }
  return result
}
```

➤ Now you can rewrite `string(from)` as follows:

```
func string(from placemark: CLPlacemark) -> String {
  var line1 = ""
  line1 = add(text: placemark.subThoroughfare, toLine: line1,
              separatedBy: "")
  line1 = add(text: placemark.thoroughfare, toLine: line1,
              separatedBy: " ")

  var line2 = ""
```

```
    line2 = add(text: placemark.locality, toLine: line2, separatedBy: "")
    line2 = add(text: placemark.administrativeArea, toLine: line2,
                separatedBy: " ")
    line2 = add(text: placemark.postalCode, toLine: line2,
                separatedBy: " ")

    return add(text: line2, toLine: line1, separatedBy: "\n")
  }
```

That already looks a bit cleaner. The logic that decides whether or not to add a `CLPlacemark` property to the string now lives inside `add(text, toLine, separatedBy)` so you no longer need all those `if let` statements. You also use `add(text, …)` to add `line2` to `line1` with a newline character in between.

➤ Run the app to see if it works.

That settles `CurrentLocationViewController`. What about the other two controllers? Well, they probably need to do something very similar.

You could either copy the `add(text, toLine, separatedBy)` method into the other two view controllers, which causes unwanted code duplication, or… you can put it into an extension on `String`. I bet you can guess which one we're going to pick.

➤ Add a new file to the project using the **Swift File** template. Name it **String +AddText**.

➤ Replace the contents of **String+AddText.swift** with:

```
extension String {
  mutating func add(text: String?, separatedBy separator: String) {
    if let text = text {
      if !isEmpty {
        self += separator
      }
      self += text
    }
  }
}
```

You've simply moved the method into this new extension.

The only difference is that the `toLine:` parameter is no longer necessary because this method now always modifies the string object that it belongs to. It adds the text and the separator to `self`.

Mutating

Notice the `mutating` keyword. You haven't seen this before. Sorry, it doesn't have anything to do with X-men – programming is certainly fun but it isn't *that* exciting.

When a method changes the value of a `struct`, it must be marked as `mutating`. Recall that `String` is a `struct`, which is a value type, and therefore cannot be

modified when declared with `let`.

The `mutating` keyword tells Swift that the `add(text, separatedBy)` method can only be used on strings that are made with `var`, but not on strings made with `let`.

If you try to modify `self` in a method on a `struct` that is not marked as `mutating`, Swift considers this an error.

You don't need to use the `mutating` keyword on methods inside a `class` because classes are reference types and can always be mutated, even if they are declared with `let`.

➤ Back in **CurrentLocationViewController.swift**, change `string(from)` to the following:

```
func string(from placemark: CLPlacemark) -> String {
  var line1 = ""
  line1.add(text: placemark.subThoroughfare, separatedBy: "")
  line1.add(text: placemark.thoroughfare, separatedBy: " ")

  var line2 = ""
  line2.add(text: placemark.locality, separatedBy: "")
  line2.add(text: placemark.administrativeArea, separatedBy: " ")
  line2.add(text: placemark.postalCode, separatedBy: " ")

  line1.add(text: line2, separatedBy: "\n")
  return line1
}
```

➤ Remove the old `add(text, toLine, separatedBy)` method; it's no longer used for anything.

There's still a small thing you can do to improve the new `add(text, separatedBy)` method from the `String` extension.

➤ In **String+AddText.swift**, change the line that defines the method to:

```
mutating func add(text: String?, separatedBy separator: String = "") {
```

You've added the `= ""` bit behind the `separator` parameter. This is known as a **default parameter value**. If a parameter has a default value, it allows callers of this method to leave out that parameter.

When you now write,

```
line1.add(text: placemark.subThoroughfare)
```

it does the same thing as:

```
line1.add(text: placemark.subThoroughfare, separatedBy: "")
```

In this case the default value for `separator` is the empty string. If the `separatedBy` parameter is left out, `separator` will be "".

➤ Make these changes in **CurrentLocationViewController.swift**:

```
func string(from placemark: CLPlacemark) -> String {

  line1.add(text: placemark.subThoroughfare)
  . . .

  line2.add(text: placemark.locality)
  . . .
```

Where the separator was the empty string, you leave out the `separatedBy: ""` part of the method call. Now you have a pretty clean solution that you can re-use in the other two view controllers.

➤ In **LocationDetailsViewController.swift**, change the `string(from)` method to:

```
func string(from placemark: CLPlacemark) -> String {
  var line = ""
  line.add(text: placemark.subThoroughfare)
  line.add(text: placemark.thoroughfare, separatedBy: " ")
  line.add(text: placemark.locality, separatedBy: ", ")
  line.add(text: placemark.administrativeArea, separatedBy: ", ")
  line.add(text: placemark.postalCode, separatedBy: " ")
  line.add(text: placemark.country, separatedBy: ", ")
  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 where placemarks are shown is `LocationsViewController`. However, this class doesn't have a `string(from)` method. Instead, the logic for formatting the address lives in `LocationCell`.

➤ Go to **LocationCell.swift**. Change the relevant part of `configure(for)`:

```
func configure(for location: Location) {
  . . .
  if let placemark = location.placemark {
    var text = ""
    text.add(text: placemark.subThoroughfare)
    text.add(text: placemark.thoroughfare, separatedBy: " ")
    text.add(text: placemark.locality, separatedBy: ", ")
    addressLabel.text = text
  } else {
```

```
    . . .
```

You only show the street and the city so the conversion is simpler. The string will be `"subThoroughfare thoroughfare, locality"`.

And that's it for placemarks.

Back to black

Right now the app looks like a typical standard iOS 10 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:

1. the status bar text has become invisible (it is black text on a black background)
2. the grey tab bar sticks out like a sore thumb (also, the yellow tint color doesn't get applied to the tab bar icons)

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 customize 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.swift**:

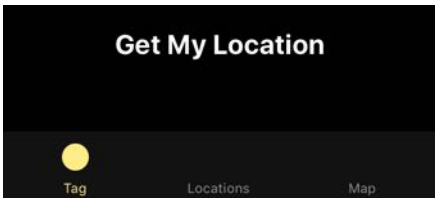
```
func customizeAppearance() {
  UINavigationBar.appearance().barTintColor = UIColor.black
  UINavigationBar.appearance().titleTextAttributes = [
                       NSForegroundColorAttributeName: UIColor.white ]

  UITabBar.appearance().barTintColor = UIColor.black

  let tintColor = UIColor(red: 255/255.0, green: 238/255.0,
                          blue: 136/255.0, alpha: 1.0)
  UITabBar.appearance().tintColor = tintColor
}
```

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 and applies the tint color to the tab bar.

➤ Call this method from the top of `application(didFinishLaunchingWithOptions)`:

```
func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions . . .) -> Bool {

  customizeAppearance()
  . . .
}
```

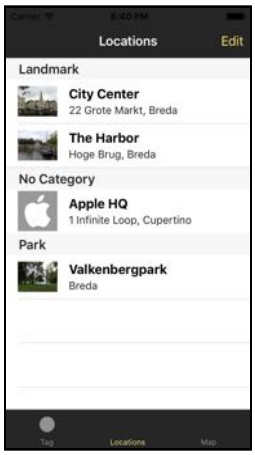
This looks better already.



The tab bar is now nearly black and has yellow icons

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.

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.

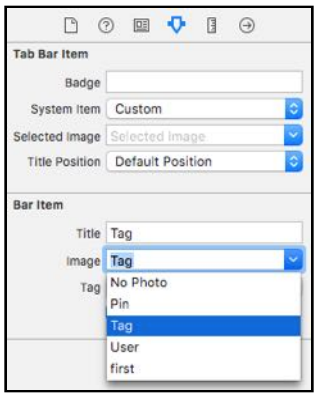
➤ Remove the **SecondViewController.swift** file from the project.

➤ Remove the **first** and **second** images from the asset catalog (Assets.xcassets).

Tab bar images should be basic grayscale images of up to 30 × 30 points (that is 60 × 60 pixels for Retina and 90 × 90 pixels for Retina HD). 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.

➤ Go to the storyboard. Select the **Tab Bar Item** of the Current Location screen. In the **Attributes inspector**, under **Image** choose **Tag**. This is the name of one of the images you've just added.

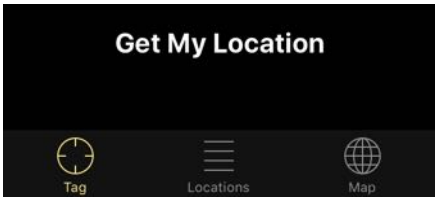


Choosing an image for a Tab Bar Item

➤ 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` property in your view controllers and make it return the value `.lightContent`.

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 source file to the project and name it **MyTabBarController**.

➤ Replace the contents of **MyTabBarController.swift** with:

```
import UIKit
```

```
class MyTabBarController: UITabBarController {
  override var preferredStatusBarStyle: UIStatusBarStyle {
    return .lightContent
  }

  override var childViewControllerForStatusBarStyle: UIViewController? {
    return nil
  }
}
```

By returning `nil` from `childViewControllerForStatusBarStyle`, the tab bar controller will look at its own `preferredStatusBarStyle` property instead of those from the other view controllers.

➤ 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.

That's right, you can replace standard UIKit components with your own subclass!

Subclassing lets you change what the built-in UIKit objects do – that's the power of object-oriented programming – although you probably shouldn't get carried away and alter their behavior *too* much. Before you know it, your app ends up with an identity crisis!

`MyTabBarController` still does everything that the standard `UITabBarController` does. You only overrode `preferredStatusBarStyle` to change the status bar color.

You can plug this `MyTabBarController` class into any app that uses a tab bar controller, and from then on all its view controllers have a white status bar.

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.

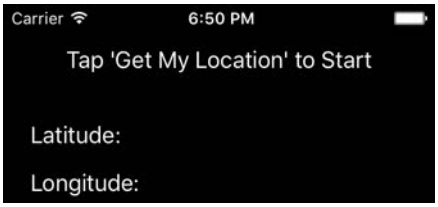
➤ Add another new file to the project and name it **MyNavigationController**.

➤ Replace the contents of **MyNavigationController.swift** with:

```
import UIKit

class MyNavigationController: UINavigationController {
  override var preferredStatusBarStyle: UIStatusBarStyle {
    return .lightContent
  }
}
```

➤ In the storyboard, select the Navigation Controllers and change their **Class** to **MyNavigationController**.

Now the status bar is white everywhere:



The status bar is visible again

Well, almost everywhere… When you open the photo picker, the status bar fades to black again. You can probably guess what the solution is…

➤ Add a new file to the project and name it **MyImagePickerController**. (Getting a sense of déjà vu?)

➤ Replace the contents of **MyImagePickerController.swift** with:

```
import UIKit

class MyImagePickerController: UIImagePickerController {
  override var preferredStatusBarStyle: UIStatusBarStyle {
    return .lightContent
  }
}
```

Now instead of instantiating the standard `UIImagePickerController` to pick a photo, you'll use this new subclass instead.

➤ Go to **LocationDetailsViewController.swift**. In `takePhotoWithCamera()` and `choosePhotoFromLibrary()`, change the line that creates the image picker to:

```
let imagePicker = MyImagePickerController()
```

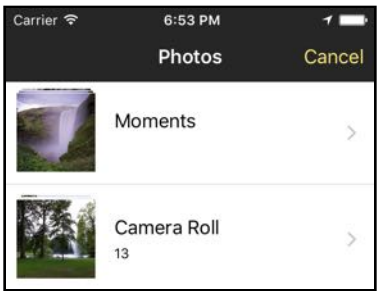
This is allowed because `MyImagePickerController` is a subclass of the standard `UIImagePickerController`, so it has the same properties and methods. As far as UIKit is concerned, the two are interchangeable. So you can use your subclass anywhere you'd use `UIImagePickerController`.

While you're at it, 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 Image Picker Controller just before you present it.

➤ Also add the following line to the two methods:

```
imagePicker.view.tintColor = view.tintColor
```

Now the Cancel button appears in yellow instead of blue.



The photo picker with the new colors

There is one more thing to change. When the app starts up, iOS looks in the Info.plist file to determine whether it should show a status bar while the app launches, and if so, what color that status bar should be.

Right now, it's set to Default, which is the black status bar.

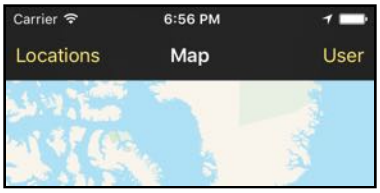
➤ Just to be thorough, go to the **Project Settings** screen. In the **General** tab, under **Deployment Info** is a **Status Bar Styl**e option. Change this to **Light**.

Changing the status bar style for app startup

And now the status bar really is white everywhere!

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 is to prefer text to icons 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:

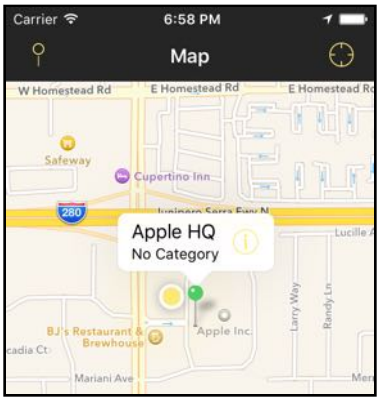
1. Remove the title. If the purpose of the screen is obvious, which it is in this case, then the title "Map" is superfluous. You might as well remove it.
2. Keep the title but replace the button labels with icons.

For this app you'll choose the second option.

➤ Go to the Map scene in the storyboard and select the **Locations** bar button item. 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:



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. There's no rule that says the tint color has to be the same everywhere!

➤ In **MapViewController.swift**, in the method `mapView(viewFor)`, add this below the line that sets `pinView.pinTintColor`:

```
  pinView.tintColor = UIColor(white: 0.0, alpha: 0.5)
```

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.swift**:

```
tableView.backgroundColor = UIColor.black
tableView.separatorColor = UIColor(white: 1.0, alpha: 0.2)
tableView.indicatorStyle = .white
```

This makes the table view itself black but does not alter the cells.

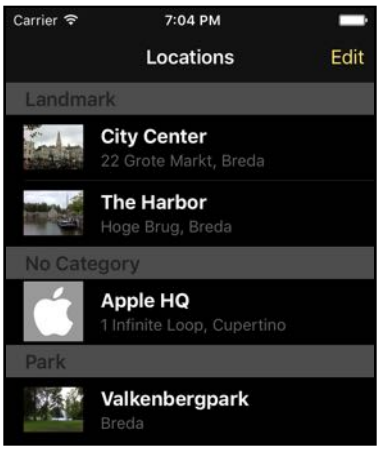
➤ In **LocationCell.swift**, implement the `awakeFromNib()` method to change the appearance of the actual cells:

```
override func awakeFromNib() {
  super.awakeFromNib()
  backgroundColor = UIColor.black
  descriptionLabel.textColor = UIColor.white
  descriptionLabel.highlightedTextColor = descriptionLabel.textColor
  addressLabel.textColor = UIColor(white: 1.0, alpha: 0.4)
  addressLabel.highlightedTextColor = addressLabel.textColor
}
```

Every object that comes from a storyboard has the `awakeFromNib()` method. This method is invoked when UIKit loads the object from the storyboard. It's the ideal place to customize its looks.

> **Note:** In case you're wondering why it's not called `awakeFromStoryboard()`, this is simply for historical reasons when nibs were all we had. A nib (or XIB) is like a storyboard but it can only hold the design of a single screen. Even though they are not as popular as they once were, you can still use nibs. In the next tutorial you'll see an example of that.

➤ Run the app. That's starting to look pretty good already:



The table view cells are now white-on-black

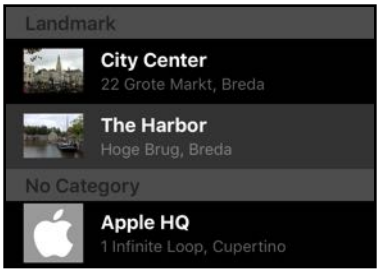
There is a small issue. When you tap a cell it still lights up in a bright color, which is a little extreme. It would look better if the selection color was more subdued.

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 `awakeFromNib()`:

```
let selectionView = UIView(frame: CGRect.zero)
selectionView.backgroundColor = UIColor(white: 1.0, alpha: 0.2)
selectedBackgroundView = selectionView
```

This creates a new `UIView` filled with 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

The section headers are also 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.

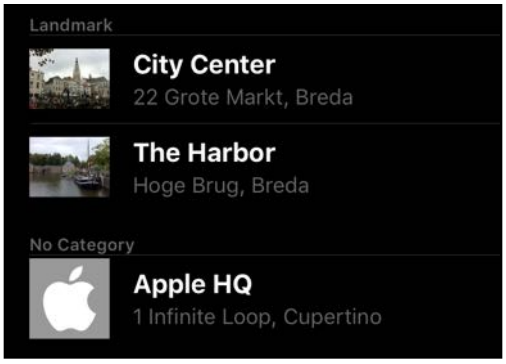
➤ Go to **LocationsViewController.swift** and add the following method:

```
// MARK: - UITableViewDelegate

override func tableView(_ tableView: UITableView,
                        viewForHeaderInSection section: Int) -> UIView? {
```

```
    let labelRect = CGRect(x: 15, y: tableView.sectionHeaderHeight - 14,
                           width: 300, height: 14)
    let label = UILabel(frame: labelRect)
    label.font = UIFont.boldSystemFont(ofSize: 11)

    label.text = tableView.dataSource!.tableView!(
                             tableView, titleForHeaderInSection: section)

    label.textColor = UIColor(white: 1.0, alpha: 0.4)
    label.backgroundColor = UIColor.clear

    let separatorRect = CGRect(x: 15,
                               y: tableView.sectionHeaderHeight - 0.5,
                               width: tableView.bounds.size.width - 15,
                               height: 0.5)
    let separator = UIView(frame: separatorRect)
    separator.backgroundColor = tableView.separatorColor

    let viewRect = CGRect(x: 0, y: 0, width: tableView.bounds.size.width,
                          height: tableView.sectionHeaderHeight)
    let view = UIView(frame: viewRect)
    view.backgroundColor = UIColor(white: 0, alpha: 0.85)
    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

If the section header has now completely disappeared on you, then add the following line to `viewDidLoad()`:

```
  tableView.sectionHeaderHeight = 28
```

This fixes a bug in Xcode that sets the section header height to -1 if you leave it at its default value of 28 in Interface Builder.

Note: Did you notice anything special about the following line?

```
label.text = tableView.dataSource!.tableView!(tableView,
titleForHeaderInSection: section)
```

This asks the table view's data source for the text to put in the header. The `dataSource` property is an optional so you're using `!` to unwrap it. But that's not the only `!` in this line…

You're calling the `tableView(titleForHeaderInSection)` method on the table view's data source, which is of course the `LocationsViewController` itself.

But this method is an optional method – not all data sources need to implement it. Because of that you have to *unwrap the method* with the exclamation mark in order to use it. Unwrapping methods… does it get any crazier than that?

By the way, you can also write this as:

```
label.text = self.tableView(tableView, titleForHeaderInSection: section)
```

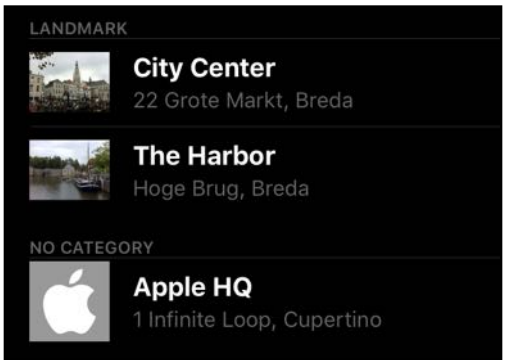
Here you use `self` to directly access that method on `LocationsViewController`. Both ways achieve exactly the same thing, since the view controller happens to be the table view's data source.

Another small improvement you can make is to always put the section headers in uppercase.

➤ Change the `tableView(titleForHeaderInSection)` data source method to:

```
override func tableView(_ tableView: UITableView,
                        titleForHeaderInSection section: Int) -> String? {
  let sectionInfo = fetchedResultsController.sections![section]
  return sectionInfo.name.uppercased()
}
```

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

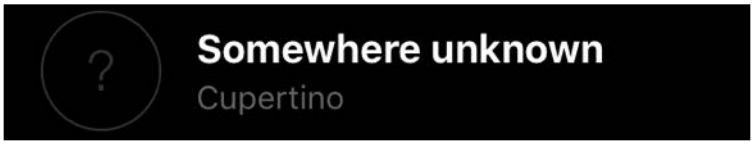
thumbnail is supposed to be. That doesn't look very professional. It's better to show a placeholder image. You already added one to the asset catalog when you imported the Images folder.

➤ In **LocationCell.swift**'s `thumbnail(for)`, replace the line that returns an empty `UIImage` with:

```
return UIImage(named: "No Photo")!
```

Recall that `UIImage(named)` is a failable initializer, so it returns an optional. Don't forget the exclamation point to unwrap the optional.

Now locations without photos appear like so:



A location using the placeholder image

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.

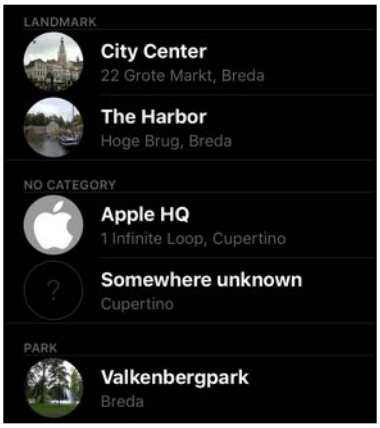
➤ Still in **LocationCell.swift**, add the following lines to the bottom of `awakeFromNib()`:

```
photoImageView.layer.cornerRadius = photoImageView.bounds.size.width / 2
photoImageView.clipsToBounds = true
separatorInset = UIEdgeInsets(top: 0, left: 82, bottom: 0, right: 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.



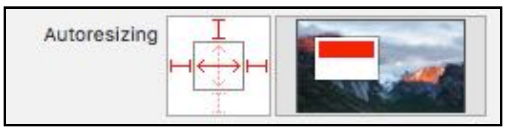
The thumbnails live inside little circles

> **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.

The labels in this screen have one final problem: they are not big enough on the iPhone 6s, 7, or Plus. Remember that the screens of these models are wider than the 320 points you've been designing for (375 and 414 points, respectively).

The obvious solution is to set autoresizing on the labels so they automatically resize.

➤ Change the autoresizing settings of the Description and Address labels to:



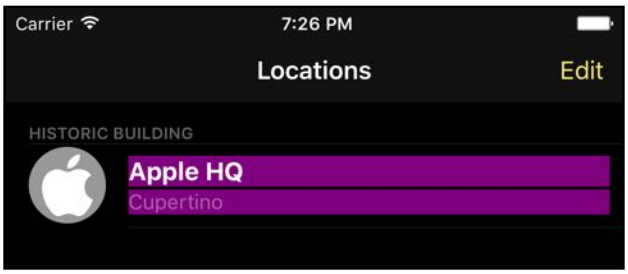
The autoresizing settings for the labels

Tip: To verify that the labels now take advantage of all the available screen space on the larger iPhones, give them a non-transparent background color. I like bright purple!

➤ Add these lines to `awakeFromNib()` and run the app:

```
descriptionLabel.backgroundColor = UIColor.purple
addressLabel.backgroundColor = UIColor.purple
```

This is what it looks like on the iPhone 7 Plus:



The labels resize to fit the iPhone 7

When you're done testing, don't forget to remove the lines that set the background color. It's useful as a debugging tool but not particularly pretty to look at.

There are two other table views in the app and they get a similar treatment.

➤ Add these lines to `viewDidLoad()` in **LocationDetailsViewController.swift**:

```
tableView.backgroundColor = UIColor.black
tableView.separatorColor = UIColor(white: 1.0, alpha: 0.2)
tableView.indicatorStyle = .white

descriptionTextView.textColor = UIColor.white
descriptionTextView.backgroundColor = UIColor.black

addPhotoLabel.textColor = UIColor.white
addPhotoLabel.highlightedTextColor = addPhotoLabel.textColor

addressLabel.textColor = UIColor(white: 1.0, alpha: 0.4)
addressLabel.highlightedTextColor = 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 "`cellForRowAt`" data source 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:

```
override func tableView(_ tableView: UITableView,
     willDisplay cell: UITableViewCell, forRowAt indexPath: IndexPath) {
  cell.backgroundColor = UIColor.black

  if let textLabel = cell.textLabel {
    textLabel.textColor = UIColor.white
    textLabel.highlightedTextColor = textLabel.textColor
  }

  if let detailLabel = cell.detailTextLabel {
    detailLabel.textColor = UIColor(white: 1.0, alpha: 0.4)
    detailLabel.highlightedTextColor = detailLabel.textColor
  }
```

```
    let selectionView = UIView(frame: CGRect.zero)
    selectionView.backgroundColor = UIColor(white: 1.0, alpha: 0.2)
    cell.selectedBackgroundView = selectionView
  }
```

The "`willDisplay`" delegate method is called just before a cell becomes visible. Here you can do some last-minute customizations on the cell and its contents.

You're using `if let` to unwrap `cell.textLabel` and `cell.detailTextLabel` because they are optionals. Not all cell types come with these built-in labels, in which case these properties are `nil`.

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 (try it out). That's because this cell does not use one of the built-in cell types so it does not have anything connected to the `textLabel` and `detailTextLabel` properties.

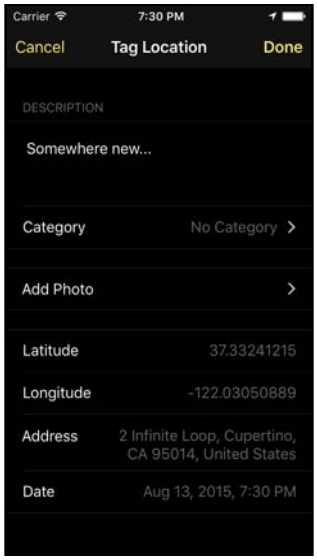
You could make a new outlet for this label but you can also do it as follows:

➤ Set the **Tag** of the "Address" label to 100 in the storyboard.

➤ Add these lines to `tableView(willDisplay, forRowAt)`:

```
  if indexPath.row == 2 {
    let addressLabel = cell.viewWithTag(100) as! UILabel
    addressLabel.textColor = UIColor.white
    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.

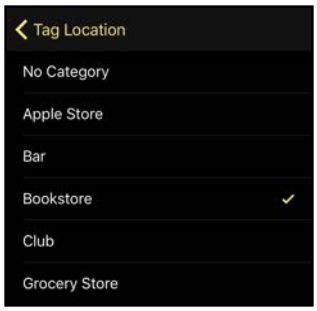
➤ Add this to `viewDidLoad()` in **CategoryPickerViewController.swift**:

```
tableView.backgroundColor = UIColor.black
tableView.separatorColor = UIColor(white: 1.0, alpha: 0.2)
tableView.indicatorStyle = .white
```

➤ And give this class a "`willDisplay`" method too:

```
override func tableView(_ tableView: UITableView,
     willDisplay cell: UITableViewCell, forRowAt indexPath: IndexPath) {
  cell.backgroundColor = UIColor.black

  if let textLabel = cell.textLabel {
    textLabel.textColor = UIColor.white
    textLabel.highlightedTextColor = textLabel.textColor
  }

  let selectionView = UIView(frame: CGRect.zero)
  selectionView.backgroundColor = UIColor(white: 1.0, alpha: 0.2)
  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.



The category picker is lookin' sharp

Polishing 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 I think we 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 found coordinates.

You will first hide the text 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.swift**:

```
@IBOutlet weak var latitudeTextLabel: UILabel!
@IBOutlet weak var longitudeTextLabel: UILabel!
```

You'll put the logic for updating these labels into a single place, the `updateLabels()` method, so that hiding and showing them is pretty straightforward.

➤ Change the `updateLabels()` method in **CurrentLocationViewController.swift**:

```
func updateLabels() {
  if let location = location {
    . . .

    latitudeTextLabel.isHidden = false
    longitudeTextLabel.isHidden = false
  } else {
    . . .

    latitudeTextLabel.isHidden = true
    longitudeTextLabel.isHidden = true
  }
}
```

➤ Connect the **Latitude:** and **Longitude:** labels in the storyboard editor to the `latitudeTextLabel` and `longitudeTextLabel` outlets.

➤ 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) while 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.

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 menu bar.

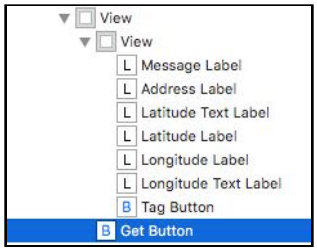
This creates a blank, white `UIView` and puts these labels and the button inside that new view.

➤ 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.

➤ To avoid problems on 3.5-inch screens, make sure that the Get My Location button sits higher up in the view hierarchy than the container view. If the button sits under another view you cannot tap it anymore.

Unintuitively, in the outline pane the button must sit below the container view. If it doesn't, drag to rearrange:



Get My Location must sit below the container view in the outline pane

Note: When you drag the Get My Location button, make sure you're not dropping it into the container view. The "View" you just added and "Get My Location" should sit at the same level in the view hierarchy.

➤ Add the following outlet to **CurrentLocationViewController.swift**:

```
@IBOutlet weak var containerView: UIView!
```

➤ In the storyboard, connect the `UIView` that contains all these labels to the `containerView` outlet.

Now on to the good stuff!

➤ Add the following instance variables to **CurrentLocationViewController.swift**:

```
var logoVisible = false

lazy var logoButton: UIButton = {
  let button = UIButton(type: .custom)
  button.setBackgroundImage(UIImage(named: "Logo"), for: .normal)
  button.sizeToFit()
  button.addTarget(self, action: #selector(getLocation),
                   for: .touchUpInside)
  button.center.x = self.view.bounds.midX
  button.center.y = 220
  return button
}()
```

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 (for example, after you press Stop and there are no coordinates and no error). To orchestrate this, you'll use the boolean `logoVisible`.

The button 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.

This is another one of those lazily loaded properties; I did that because it's nice to keep all the initialization logic inline with the declaration of the property.

➤ Add the following method:

```
// MARK: - Logo View

func showLogoView() {
  if !logoVisible {
    logoVisible = true
    containerView.isHidden = true
    view.addSubview(logoButton)
  }
}
```

This hides the container view so the labels disappear, and puts the `logoButton` object on the screen. This is the first time `logoButton` is accessed, so at this point the lazy loading kicks in.

➤ In `updateLabels()`, change the line that says,

```
statusMessage = "Tap 'Get My Location' to Start"
```

into:

```
statusMessage = ""
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 to check it out.

When you tap the logo (or Get My Location), the logo should disappear and the panel with the labels ought to show up. That doesn't happen yet, so let's write some more code.

➤ Add the following method:

```
func hideLogoView() {
  logoVisible = false
  containerView.isHidden = false
  logoButton.removeFromSuperview()
}
```

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 `getLocation()`, right after the authorization status checks:

```
  if logoVisible {
    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()`.

➤ First add an import for QuartzCore, the framework that provides Core Animation. This goes all the way at the top of the file, as usual:

```
  import QuartzCore
```

➤ Also give `CurrentLocationViewController` the ability to handle animation events by making it the `CAAnimationDelegate`:

```
  class CurrentLocationViewController: UIViewController,
                        CLLocationManagerDelegate, CAAnimationDelegate {
```

➤ Then replace `hideLogoView()` with:

```
  func hideLogoView() {
    if !logoVisible { return }

    logoVisible = false
    containerView.isHidden = false
    containerView.center.x = view.bounds.size.width * 2
    containerView.center.y = 40 + containerView.bounds.size.height / 2

    let centerX = view.bounds.midX

    let panelMover = CABasicAnimation(keyPath: "position")
    panelMover.isRemovedOnCompletion = false
    panelMover.fillMode = kCAFillModeForwards
    panelMover.duration = 0.6
    panelMover.fromValue = NSValue(cgPoint: containerView.center)
    panelMover.toValue = NSValue(cgPoint:
                          CGPoint(x: centerX, y: containerView.center.y))
    panelMover.timingFunction = CAMediaTimingFunction(
                                      name: kCAMediaTimingFunctionEaseOut)
    panelMover.delegate = self
    containerView.layer.add(panelMover, forKey: "panelMover")

    let logoMover = CABasicAnimation(keyPath: "position")
    logoMover.isRemovedOnCompletion = false
```

```
    logoMover.fillMode = kCAFillModeForwards
    logoMover.duration = 0.5
    logoMover.fromValue = NSValue(cgPoint: logoButton.center)
    logoMover.toValue = NSValue(cgPoint:
                            CGPoint(x: -centerX, y: logoButton.center.y))
    logoMover.timingFunction = CAMediaTimingFunction(
                                        name: kCAMediaTimingFunctionEaseIn)
    logoButton.layer.add(logoMover, forKey: "logoMover")

    let logoRotator = CABasicAnimation(keyPath: "transform.rotation.z")
    logoRotator.isRemovedOnCompletion = false
    logoRotator.fillMode = kCAFillModeForwards
    logoRotator.duration = 0.5
    logoRotator.fromValue = 0.0
    logoRotator.toValue = -2 * M_PI
    logoRotator.timingFunction = CAMediaTimingFunction(
                                        name: kCAMediaTimingFunctionEaseIn)
    logoButton.layer.add(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.

➤ Add the following method below `hideLogoView()`:

```
  func animationDidStop(_ anim: CAAnimation, finished flag: Bool) {
    containerView.layer.removeAllAnimations()
    containerView.center.x = view.bounds.size.width / 2
    containerView.center.y = 40 + containerView.bounds.size.height / 2

    logoButton.layer.removeAllAnimations()
    logoButton.removeFromSuperview()
  }
```

This cleans up after the animations and removes the logo button, as you no longer need it.

➤ Run the app. Tap on Get My Location to make the logo disappear. I think the animation looks pretty cool.

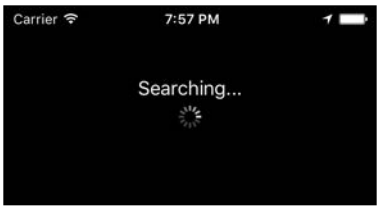
Tip: To get the logo back so you can try again, first choose **Location → None** from the Simulator's **Debug** menu. Then tap Get My Location followed by Stop to make the logo reappear.

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 currently change the button's text to say 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).

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 `configureGetButton()` to the following:

```
func configureGetButton() {
  let spinnerTag = 1000

  if updatingLocation {
    getButton.setTitle("Stop", for: .normal)

    if view.viewWithTag(spinnerTag) == nil {
      let spinner = UIActivityIndicatorView(
                                        activityIndicatorStyle: .white)
      spinner.center = messageLabel.center
      spinner.center.y += spinner.bounds.size.height/2 + 15
      spinner.startAnimating()
      spinner.tag = spinnerTag
      containerView.addSubview(spinner)
    }
  } else {
    getButton.setTitle("Get My Location", for: .normal)

    if let spinner = view.viewWithTag(spinnerTag) {
      spinner.removeFromSuperview()
    }
  }
```

```
  }
```

In addition to changing the button text to "Stop", you create a new instance of `UIActivityIndicatorView`. 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.

To keep track of this spinner view, you give it a tag of 1000. You could use an instance variable but this is just as easy and it keeps everything local to the `configureGetButton()` method. It's nice to have everything in one place.

When it's time to revert the button to its old state, you call `removeFromSuperview()` to remove the activity indicator view from the screen.

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 to 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.swift**:

```
  import AudioToolbox
```

➤ Add the `soundID` instance variable:

```
  var soundID: SystemSoundID = 0
```

Because writing just 0 would normally give you a variable of type `Int`, you explicitly mention the type that you want it to be: `SystemSoundID`. This is a numeric identifier – sometimes called a "handle" – that refers to a system sound object. 0 means no sound has been loaded yet.

➤ Add the following methods to the bottom of the class:

```
// MARK: - Sound Effect

func loadSoundEffect(_ name: String) {
  if let path = Bundle.main.path(forResource: name, ofType: nil) {
    let fileURL = URL(fileURLWithPath: path, isDirectory: false)
    let error = AudioServicesCreateSystemSoundID(fileURL as CFURL,
                                                 &soundID)
    if error != kAudioServicesNoError {
      print("Error code \(error) loading sound at path: \(path)")
    }
  }
}

func unloadSoundEffect() {
  AudioServicesDisposeSystemSoundID(soundID)
  soundID = 0
}

func playSoundEffect() {
  AudioServicesPlaySystemSound(soundID)
}
```

The `loadSoundEffect()` method loads the sound file and puts it into a new 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()` from `viewDidLoad()`:

```
loadSoundEffect("Sound.caf")
```

➤ In `locationManager(didUpdateLocations)`, in the geocoder's completion closure, change the following code:

```
if error == nil, let p = placemarks, !p.isEmpty {

  if self.placemark == nil {               // add this
    print("FIRST TIME!")
    self.playSoundEffect()
  }

  self.placemark = p.last!
} else {
  . . .
```

The added if-statement simply checks whether the `self.placemark` instance variable is still `nil`, in which case this is the first time you've reverse geocoded an address. It then plays a sound using the `playSoundEffect()` method.

Of course, you shouldn't forget to include the actual sound effect into the project!

➤ Add the **Sound** folder from this tutorial's Resources to the project. Make sure **Copy items if needed** is selected (click the Options button in the file open panel to

reveal this option).

➤ 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 – when you see the FIRST TIME! print – even if more precise locations keep coming in afterwards.

Note: If you don't hear the sound on the Simulator, try the app on a device. There have been reports that system sounds are not always playing in the simulators.

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.)

The icon and launch images

The Resources folder for this tutorial contains an **Icon** folder with the icons for this app.

➤ Import the icon images into the asset catalog. Simply drag them from Finder into the **AppIcon** group.

It's best to drag them one-by-one into their respective slots (if you drag the whole set of icons into the group at once, Xcode can get confused).

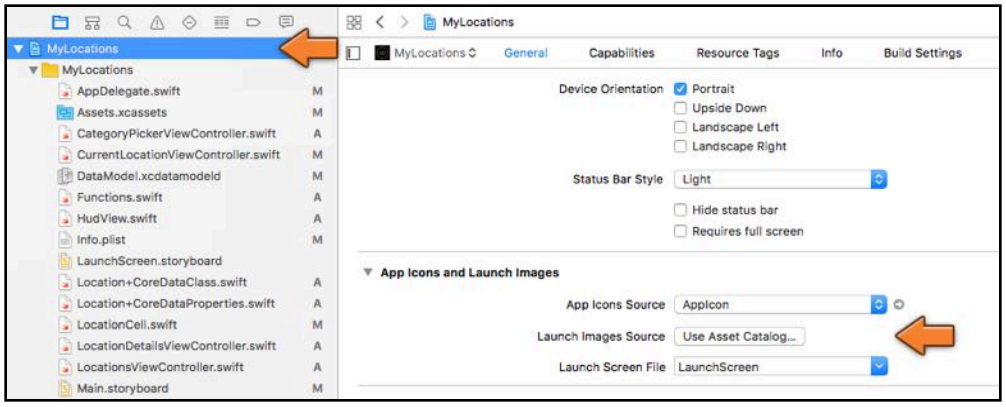
The icons in the asset catalog

The app currently also has a launch file, **LaunchScreen.storyboard**, that provides the "splash" image while the app loads.

Having a launch file is required to take advantage of the larger screens of the iPhone 6s, 7, and Plus. Without this launch file, the app will think it's running on a 4-inch screen (like the iPhone SE) but gets scaled up to fill the extra pixels on the iPhone 6 and 7 models. That doesn't look very good, except maybe for games. So you definitely want the app to use a launch file.

Instead of using a storyboard for the launch screen you can also supply a set of images. Let's do that for this app.

➤ In the **Project Settings** screen, in the **General** tab, find the **App Icons and Launch Images** section. Click the **Use Asset Catalog** button next to **Launch Images Source**:



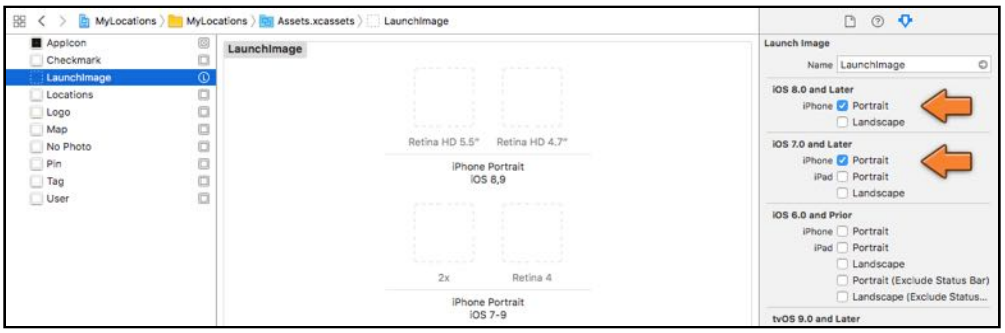
Using the asset catalog for launch images

Xcode now asks if you want to migrate the launch images. Click **Migrate**.

➤ Make the field for **Launch Screen File** empty.

➤ Also remove **LaunchScreen.storyboard** from the project. It's also a good idea to delete the app from the Simulator, or even reset it, so that there is no trace of the old launch screen.

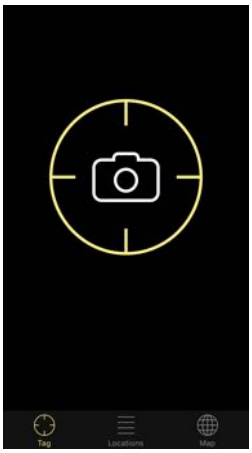
➤ Open **Assets.xcassets**. There is now a **LaunchImage** item in the list (this may also be called BrandAsset). Select it and go to the Attributes inspector. Under both **iOS 8.0 and Later** and **iOS 7.0 and Later**, put checkmark by **iPhone Portrait**:



Enabling the launch images for iPhone portrait

You now have four slots for dropping the launch images into. (If you have any slots that say "Unassigned", then select and remove them by pressing the delete key.)

The Resources folder for this tutorial contains a **Launch Images** folder. Let's take a look at one of those images, **Launch Image Retina 4.png**:



The launch image for this app

The launch image only has the tab bar and the logo button, but no status bar or any buttons. The reason it has no "Get My Location" button is that you don't want users to try and tap it while the app is still loading (it's not really a button!).

To make this launch image, I ran the app in the Simulator and chose **File → Save Screen Shot**. This puts a new PNG file on the Desktop. I then opened this image in Photoshop and blanked out any text and the status bar portion of the image. The iPhone will draw its own status bar on top anyway.

➤ Drag the files from the **Launch Images** folder into the asset catalog, one at a time. It should be pretty obvious into which slot each image goes.

Done. That was easy. :-)

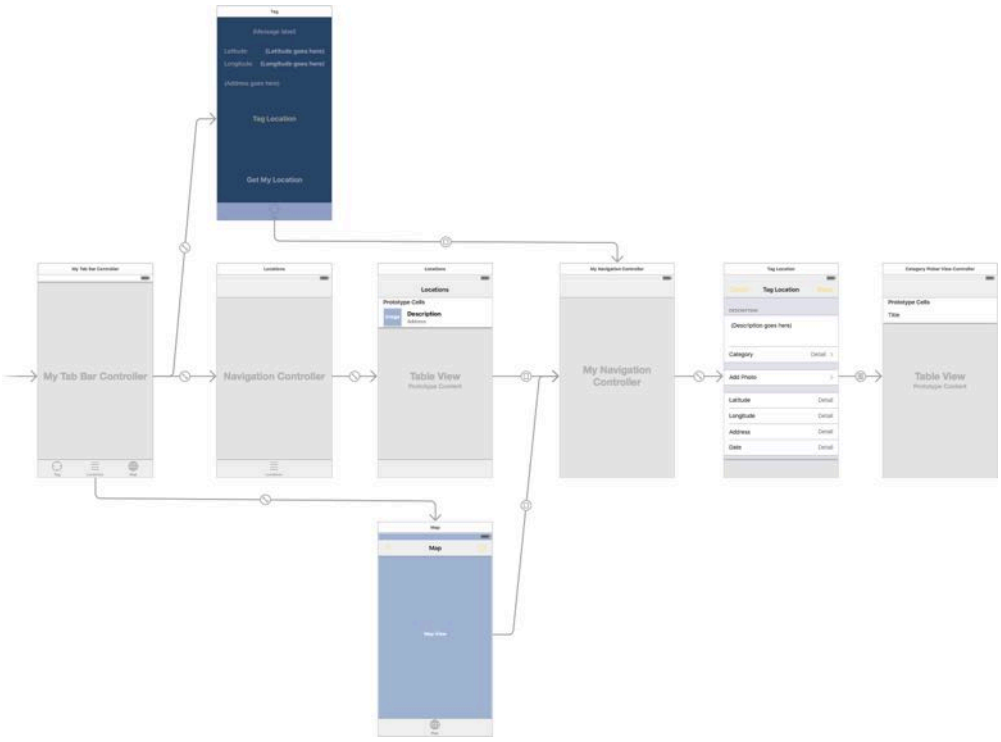
And with that, the MyLocations app is complete! Woohoo!

You can find the final project files for the app under **07 - 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:



In this lesson you took a more detailed look at Swift but there is still plenty to discover. To learn more about the Swift programming language, I recommend that you read the following books:

- **The Swift Programming Language** by Apple. This is a free download on the iBooks Store. If you don't want to read the whole thing, at least take the Swift tour. It's a great introduction to the language.
- **Swift Apprentice** by the raywenderlich.com Tutorial Team. This is a book that teaches you everything you need to know about Swift, from beginning to advanced topics. This is a sister book to the iOS Appentice; the iOS Apprentice focuses more on making apps, while the Swift Apprentice focuses more on the Swift language itself. www.raywenderlich.com/store

There are several good Core Data beginner books on the market. Here are two

recommendations:

- **Core Data by Tutorials** by the raywenderlich.com Tutorial Team. One of the few Core Data books that is completely up-to-date with the lastest iOS and Swift versions. This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn how to use Core Data to save data in their apps. www.raywenderlich.com/store
- **Core Data Programming Guide** by Apple. If you want to get into the nitty gritty, then Apple's official guide is a must-read. You can learn a ton from this guide. developer.apple.com/library/prerelease/content/documentation/Cocoa/Conceptual/CoreData/#//apple_ref/doc/uid/TP40001075-CH2-SW1

Credits for this tutorial:

- Sound effect based on a flute sample by elmomo, downloaded from The Freesound Project (freesound.org)
- Image resizing category is based on code by Trevor Harmon (vocaro.com/trevor/blog/2009/10/12/resize-a-uiimage-the-right-way/)
- HudView code is based on MBProgressHud by Matej Bukovinski (github.com/matej/MBProgressHUD)

Are you ready for the final lesson? Then continue on to tutorial 4, where you'll make an app that communicates over the network to a web service!