LECTURE #2

Welcome

This week we're pulling back from the UI a bit and focusing more on the Swift language. This lesson will talk about how to use some of the basics of Swift such as numbers, strings, arrays, and dictionaries. The project will add back in the UI components and will push you to integrate your code with what you see on screen.

If you have a good understanding of object-oriented programming, you might be just as well served checking out the *Swift Programming Guide* on the iBooks Store.

Think Swiftly

In this tutorial, you'll be developing a simple version of recipe book that contains a bunch of recipes. Again, we won't be showing how to make this with buttons and labels, just how to write the code to maintain these objects. The project is where you'll use both.

Go ahead and create a new project, but instead of selecting a *Single View Application*, select *Command Line Tool* under the *OS X>Application* heading. Be sure to select the language to be *Swift*. Once you've created the project. You'll be presented with main.swift as the only file to work with. You can delete everything in this file, except for the *println* function that prints out "Hello, World!". Run this application (play button in the top left) and check out the output in the bottom. This area of output is known as the *console*, so whenever we say, "print to the console", this is what we're referring to.

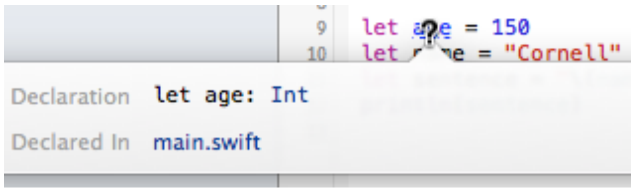
Basics

That's great and all, but let's play with some code. Go ahead and remove that *println* line and copy down the following code:

```
let age = 150
let name = "Cornell"
let sentence = "\(name) will be \(age) years old this year!"
println(sentence)
```

Listing 2-1

You'll notice that we start every declaration of a new constant with the **`let`** keyword which in the example of the *age* says that the constant called age will now has the value 150. Likewise, **`name`** now has the value "Cornell". The **`age`** and **`name`** have different types though.



```
let age = 150
let name = "Cornell"
```

Declaration `let age: Int`

Declared In main.swift

Figure 2-1

The **age** is of type *Int* (which holds integer values) and **name** is of type *String* (holds a bunch of characters). When we create the **sentence**, we are using string interpolation to insert the values of our name and age to create a new string called **sentence**. You can then see the output in the console. To view the types of any constant in Swift, you can option-click on the name of the constant and it will show you the type (Figure 2-1).

You'll notice a few problems about this code though, mainly if you tried either of the following:

```
let newThing = name + age // Error: 'String' is not convertible to 'Int'
age = 151                 // Error: Cannot assign to 'let' value 'age'
```

Listing 2-2

The first line generates an error because in Swift we cannot merge different types. **name** was declared a *String* and **age** was declared an *Int*. We would have to convert the age to a string before we could concatenate them together, but this can be accomplished with the string interpolation we saw with **sentence** in Listing 2-1.

The second line is an issue because anything we define using the **let** keyword is an immutable constant. This means that its value cannot be altered once defined. To fix this, we can declare our age as a *variable* or **var**. The following is perfectly valid code:

```
var mutableAge = 20
mutableAge = 21
```

Listing 2-3

Arrays and Dictionaries

Often times we want to hold a large collection of objects in our applications. Maybe we want to keep a list of groceries, your best friends, or all the bad grades you're bound to get at Cornell. Whatever it might be, we can hold this group of items in an a*rray*.

```
let bestFriends = ["John", "Jacob", "Paul"]
var grades = [98, 88, 72, 45]  // Typical Cornell class over time
grades[0] = 92 // You didn't get a 98!

println("My best friend \(bestFriends[0]) got \(grades[2]) on his 3rd test")
// Console: My best friend John got 72 on his 3rd test
```

Listing 2-4

Important things to notice are that arrays are created using an opening and closing square bracket, each item in the array is separated by a comma. We access the items in the order they're entered, the first element in the array is located at the 0th index and so on. The call **bestFriends[0]** is fetching the first element (0th index) of the array which is John, and the **grades[2]** gets back 72. You'll notice that the arrays were declared using different keywords. The **let** keyword means that the array cannot be changed in anyway, ever. The **var** keyword means we can add, remove, insert and even change elements that are in the array. That's arrays!

If you wanted to lookup a word in a dictionary, you would first find the word (or search for it on your computer) and then the word would have an associated definition. *Dictionaries* in Swift are the same idea, you can "lookup" a *key*, and get back a *value*. In our dictionary example, the *key* would be the word you want to lookup and the *value* would be the definition. Let's see an example:

```
var ages = ["John": 20, "Jacob": 40, "Paul": 21]
ages["John"] = 10

let johnAge = ages[“John"]
println("John is \(johnAge) years old") // Console: John is 10 years old
```

Listing 2-5

Dictionaries are declared in a similar fashion to arrays, however the difference is in the colon. We have 3 string keys and 3 int values. John is matched with the value 20, Jacob to 40, etc. The syntax for accessing dictionaries looks like arrays as well, except instead of an index we use a key.

Optionals

Sometimes when we are programming, we are not guaranteed to have valid values. For example, if the user of our app enters their year of birth into a text field, the text field stores this as a **`String`**. However, when we want to deal with years, we might prefer to hold this value as an **`Int`**. Let's look at the following conversions of strings:

```
let possibleNumber = "123"
let validNum = possibleNumber.toInt()

let alsoPossibleNumber = "12.2.2"
let invalidNum = alsoPossibleNumber.toInt()
```

Listing 2-6

This first case would work out well because it is clear that the value of "123" would be converted to 123. The second case is not clear though, it cannot be represented as an **`Int`**. The toInt() method will return an *optional Int*, otherwise known as **`Int?`**. This means that we will either get back the value **`Some(Int)`** or **`nil`**. The first example would be Some(123) and the latter would return nil because it failed to be converted. We can check if we get valid values in return by using an **`if let`** statement.

```
if let num = possibleNumber.toInt() {
    print(num+2) // Console: 125
} else {
    print("Not a number")
}
```

Listing 2-7

The **`if let`** statement unwraps the Some(123) and assigns the Int value of 123 to num. If we tried the same example with **`alsoPossibleNumber`**, we would print out "Not a number" because "12.2.2" cannot be converted into an Int.

If we absolutely know that our value will be valid, then we can force unwrap that value with the ! operator. Like this:

```
validNum! + 5 // Unwraps Some(num) to num and then adds 5
```

Listing 2-8

Classes and Protocols

Let's start tackling the original problem outlined in this lesson, how can we represent a recipe? A recipe might have a title, creator, cook time, list of ingredients, and more. How can we represent all of these different things as one object? Classes of course!

Our recipe class described above would look like this:

```
class Recipe: NSObject {
    var title: String
    var creator: String
    var prepTime = 0
    var cookTime: Int?
    var ingredients: [String]
    var directions: String

    init(title: String, creator: String, ingredients: [String], directions: String) {
        self.title = title
        self.creator = creator
        self.ingredients = ingredients
        self.directions = directions
    }

    override var description: String {
        return "\(title) by \(creator)"
    }
}

let recipe = Recipe(
    title: "Banana Split",
    creator: "Lucas",
    ingredients: ["Ice Cream", "Bananas"],
    directions: "Add banana to ice cream, BAM"
)

println(recipe)
// Console: Banana Split by Lucas
```

Listing 2-9

There's quite a bit going on in this code snippet, so let's digest it.

First off, we're declaring a new class called *Recipe*. The class conforms to the **`Printable`** protocol which means that we have to implement the *computed property*, **`description`**. By returning a string in that computed property, we have met the requirements for the **`Printable`** protocol and can now use our object in a **`print`** or **`println`** and have it print out the title and the creator of the recipe.

After we declare the class and any protocols to which it conforms, we declare all the *properties* that the class might contain. A *stored property* is used to store any information we'd like to keep in our recipe such as the **`title`**, **`creator`**, **`prepTime`**, etc. Looking at some of these we can see that a lot of them are of type *String*. **`prepTime`** we defined as 0 instead of requiring that every recipe defines a preparation time. Some recipes might not have any **`cookTime`** because they might not need to be cooked, so we declared this as an *Int? (optional Int)*. Notice that optionals don't require initialization, that's because they're optional, but will be initialized to *nil* if they aren't set to anything. Lastly, **`ingredients`** has type *[String]* which is an array of strings.

Then we come to the **`init`**, which is our initializer for the class. Our init requires that when we create a new recipe object we know the title, creator, ingredients, and directions. We use self.propertyName to distinguish the propertyName from the names of the parameters passed in. A very important thing to know about classes in Swift is that every stored property must have a value when the class is created, whether that be setting a value directly like we did with prepTime, or initializing the value in our init.

Once you've defined a printable **`Recipe`** class, now you have to create an object. This is done in the above code with the line **`let recipe = Recipe(…)`**. Here we are creating a new Recipe object and using our init call to setup the object. Once we have created a recipe, we can print it out and behold the magic of Swift.

That's it for this lesson. This should be enough to get you up and running for this week's project. Don't forget to check out the *Swift Programming Guide* on the iBooks Store for more information on everything we discussed.