Notes From the Class

Enum

<https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html>

<https://codewithchris.com/swift-enum/>

OOP

I’d define it as an “approach” to programming, which is based on representing everything in your codebase with “entities” to make the code modular and easier to manipulate

Class is a casting mold and the object is the thing you create out of that mold

Structs

Structs are value types while classes are reference types. Structs can be instantiated without using a new operator -google

A struct is a value type so theoretically in the computer memory a struct uses that memory and stores it on top of a stack. That is why you must use a “mutating func” in order to change a variable

Class Initializing

can I write more than one init with different arguments (Attributes)?

Yes, You Can

### Struct Initialization

We can pass value to the initialized variable even if it has default value

**Creating Your Own Types in Swift**

There are three main ways for us to create our own types in Swift: **Classes, Structs, and Enums.** We'll be focusing on Classes and Structs since you will be using them heavily in your own code. If you complete all this material early, look into Enums! Let's start with classes.

**Classes**

Classes in Swift allow you to define the blueprint of a particular type and then create instances of that type (also known as Objects). This paradigm of creating your own types and then instantiating them is known as **Object Oriented Programming** and becomes very powerful when creating large scale applications with multiple components. The general OOP concepts that you have already learned apply here as well, this section is primarily about learning the swift syntax. Let's start by creating a class:

class Person {

var species = "H. Sapiens"

}

var myPerson: Person = Person()

print(myPerson.species)

Here we have created the class Person that has one **property** "species" and then we create an instance of the person called "myPerson". Note that we are declaring the type of myPerson to be the class that we defined above. This means we can create our own types!

**Stored Properties**

Above the Person class has **one stored property and it has a default value of "H. Sapiens"**. Since we declared the property as a variable it is readable and writable (meaning we can access it *and* change it).

var myPerson = Person() // Note use of type inference to infer that it is "Person" type

myPerson.species = "Homo Sapiens" // We can change the property

print(myPerson.species)      // And we can access it!

We can  **create read-only properties by simply declaring properties as a constant**. It can't be written after initialization which makes it a read-only property.

class Person {

let species = "H. Sapiens"

}

var myPerson = Person()

myPerson.species = "Homo Sapiens" // ! => Cannot assign to let constant 'species' in myPerson

**Methods**

**Methods are functions that are associated with a Type**. While properties define what an instance "knows", methods define what an instance "can do". Let's see methods in action with our Person example.

class Person {

var species = "H. Sapiens"

func speak() {

print("Hello! I am a \(self.species)") // Note how we refer to the properties of the instance through "self"

}

}

var myPerson: Person = Person()

myPerson.speak() // Note we call method with dot notation similar to how we

                                             // access properties

**Method Parameter Names**

So far we only implemented methods that take no arguments. Methods and functions are very similar.  **Methods are just functions associated with a Type**.

class User {

var intelligence = 0

func studyFor(topic: String, hours: Int) {

print("I am studying \(topic) for \(hours) hours")

}

}

var user = User()

user.studyFor(topic: "Math", hours: 12)

It is common practice to  **write the name of the function** ('studyFor') so that it sounds good when combined with the **first arguments external parameter name** (topic), so it reads 'study for topic'. By following this practice, it becomes very clear what each argument does.

**Initialization**

Initialization is the **operation of setting up an instance of a type**. It helps us create an instance with appropriate values. So far, we didn't have to worry about initializer because **we have been giving default stored values when defining properties**. However, if we want more control over how an instance of a Type is created, we will have to write our own initializer.

Let's add a name property to our Person class and then add the code to allow a user to specify the name when initializing the instance.

class Person {

var species = "H. Sapiens"

var name: String       // Note we are only defining the type, we will initialize the value in the init methi

init(name: String) { // Note this function doesn't get called explicitly. It is called

// when creating an instance using initialization syntax -- "Person()".

self.name = name // Note use of "self" here to refer to the name property.

}

func speak() {

print("Hello! I am a \(self.species) and my name is \(self.name)")

// Note how we refer to the properties using "self".

}

}

var myPerson: Person = Person(name: "Jay") // Now we have to pass a param to Person initialization.

myPerson.speak() // Note we call method similar to how we access properties.

# Structs in Swift

Structs in Swift are used to create types that hold a collection of values. **At first glance Structs may look very similar to classes but there are a couple key differences that we'll be going over in the coming chapters.**

Structs, like classes, are used to create types. Generally, Structs are used more for creating collections of values. For example representing shapes is a great use for structs! Let's see this in action:

struct Rectangle {

var width = 10

var height = 20

}

var myRectangle = Rectangle() // Taking advantage of type inference

print(myRectangle.width)

Here we declare the Rectangle struct that has two stored properties: width and height that each has default values of 10 and 20 respectively. We instantiate the rectangle using the same syntax as with Classes.

### Struct Methods

Methods in Structs are very similar to methods in classes. Let's revisit our rectangle example.

struct Rectangle {

var width = 10

var height = 20

func printDesc() {

print("I have a width of \(width) and a height of \(height)")

}

mutating func doubleWidth() { // Why do we need the mutating keyword here?

width \*= 2

}

}

We implemented two methods in Rectangle. The first method, printDesc does not have to be prefixed **with the mutating keyword because the method is not changing the value of the instance in any way**. However, our second method, doubleWidth, **changes the value of one of its properties. Since the value is changing we have to prefix the method with the mutating keyword.**

**Interestingly enough, we do NOT need the mutating keyword when writing a similar method in a Class. This has to do with value types vs reference types which we will go over soon.**

### Struct Initialization

Similar to Classes, with Structs, we need to initialize our properties when instantiating the class if they are not given default values. Fortunately, **Swift will give us a "memberwise" initializer so that we don't have to explicitly define the initialization method**. This **memberwise initializer will include external parameter names for all of the stored properties that need values**. Let's see this in action.

struct Rectangle {

var width = 200

var height = 400

}

let rectangle = Rectangle()

In the above implementation of the Rectangle Struct, we used the free "empty" initializer provided to us by the Swift compiler. We are given a  **free "empty" initializer for our Types if we give all of our stored properties default values** **and do not define our own custom initializer.**

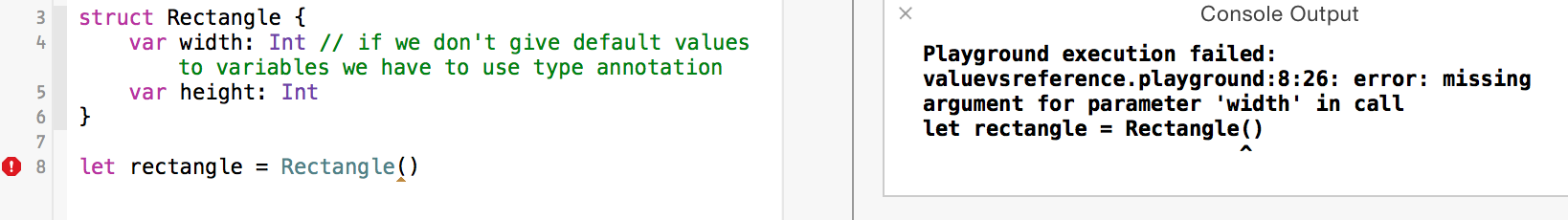
struct Rectangle {

var width: Int // If we don't give default vals to vars, we have to use type annotation.

var height: Int

}

If we don't set the default values for all of the stored properties, then we will have to rely on the **"memberwise" initializer mentioned above.** **In this case, we don't have access to the free "empty" initializers anymore because we no longer can ensure that every property is going to have a value by the end of the initialization.**



Wait a minute...the error is saying that we are missing an external parameter name...but we never defined any. Here, because we didn't provide default values for our stored properties, and we didn't implement our own custom initializer,  **Swift will give us a "memberwise" initializer this time**.  To reiterate, the **memberwise initializer will include external parameter names for all of the stored properties that need values**.

struct Rectangle {

var width: Int

var height: Int

}

let rectangle = Rectangle(width: 200, height: 400)

### Custom Init Method

If we want to have custom logic in our init method, we can define it just like we did with our classes.

struct Rectangle {

var width: Int

var height: Int

var size: String

init (width: Int, height: Int){

self.width = width

self.height = height

if width + height <= 10 {

size = "small"

} else {

size = "big"

}

}

}

let rectangle = Rectangle(width: 200, height: 400)

print(rectangle.size)

It is important to remember that the  **primary role of initialization is to make sure that the new instance has values for all of its stored properties**. If this is not possible, Swift will yell at us until we make the necessary changes to ensure that all stored properties will have a value at the end.

### But wait... What's the real difference between Classes and Structs? Why do we have both?

To really understand why we have both Classes and Structs we must understand the difference between value and reference types. Turn to the next chapter to explore this.

# Value vs Reference Types

Every type in Swift falls into one of **two categories: Value Types or Reference Types.** The easiest way to remember which is which is to remember that **Classes and Functions are the only Reference types** and everything else is a Value Type. But what does Value Type actually mean?

**Value types are copied**when they are **assigned to a variable or a constant** and **when they are passed in as an argument to a function**. If you change the value of the copy, you will not change the original.

On the other hand, **Reference types** are not copied. Instead, their **memory address location**is passed back and forth so changing the value of a Reference Type will change all of the "pointers" (variables or arguments) that refer to that object.

Let's take a look at an example:

class Person {

var fullName: String

init(name: String) {

self.fullName = name

}

func introduce() {

print("Hi my name is \(self.fullName)")

}

}

var j = Person(name: "Ketul Patel") // Initialize a Person object and assign it to the j variable

var k = j // Create a k variable and set its value to be the j variable

// (remember classes are reference types)

j.introduce() // Prints "Hi my name is Ketul Patel"

k.fullName = "Ketul J Patel" // We are changing the name through the k variable.

j.introduce() // Prints "Hi my name is Ketul J Patel" since both

// j and k refer to the same instance in memory

Whoa, that's a lot of confusing code! Let's walk through it step by step.

1. We create the Person class that has a "fullName" property and an introduce method
2. We create an instance of Person and assign it to the "j" variable. This instance has the value "Ketul Patel" stored in the fullName property
3. We create another variable "k" and assign it to the "j" variable. Since classes are reference types instead of copying the instance in the "j" variable we simply point both "k" and "j" to the same instance in memory
4. Since k and j refer to the same instance, when we change the value by referencing the instance through the k variable it changes the value for j as well (one instance in memory referred to by 2 variables or "pointers")

**In contrast, Structs are Value Types which means that when they are passed, their values are copied and then passed.**

Let's take a look at an example of this:

struct Rectangle {

var width: Int

var height: Int

}

var square1 = Rectangle(width: 10, height: 10)

var square2 = square1 // Here the val inside square 1 (an instance of Rectangle)

// is copied and passed rather than just pointed to.

print("square1's width: \(square1.width), square2's width \(square2.width)")

// They are the same because square2 is a copy of square1

square2.width = 20 // This only changes square2 because there are

// 2 separate instances of rectangle in memory

square2.height = 20

print("square1's width: \(square1.width), square2's width \(square2.width)")

// They are different now: changing square2 changed a

// completely separate instance from square1.

The major difference between Structs and Classes is the Value vs Reference Types that we see in the examples above. Remember that we **pass types in 2 main ways -- pass to a variable and pass to a function**.

**Every Type in Swift that is not an instance of a class or function is a Value Type. This includes Strings, Arrays, and Dictionaries which are implemented internally as Structs in Swift.**

The only way to pass a Value Type by memory location is by using the **inout** designation when passing the Value type as an argument to a function. If you have some spare time go read about **inout**, it's really cool!