Notes From the Class

Chaining methods in swift — (Don’t confuse with optional chaining)

<https://abhimuralidharan.medium.com/chaining-methods-in-swift-not-optional-chaining-3007d1714985>

Destroy The Object

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

**Inheritance**

There are two main differences between Structs and Classes. We discussed how **Structs are passed by value** and **Classes are passed by reference**. The second main difference is that **Classes support inheritance** while **Structs do not**. In our Classes example, we created a Person class. Now let's create a Developer class that **inherits** from the Person class.

class Person {

var species = "H. Sapiens"

var name: String

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

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

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

// instead of the name parameter

}

func speak() {

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

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

}

}

class Developer: Person { // Note how we are specifying that Developer will inherit from Person

var favoriteLanguage: String

init(name: String, favoriteLanguage: String) { // Note there is no override keyword on init because it has

// different parameter names then the Person init method

self.favoriteLanguage = favoriteLanguage

super.init(name: name)

}

override func speak() { // Note there is an override keyword on speak because it is

// identical to the Person speak method

print("Hello! I am a Developer! My name is \(self.name)")

}

}

var myDeveloper: Developer = Developer(name: "Jay", favoriteLanguage: "Swift")

myDeveloper.speak()

Wow, that's a lot of new code! Let's go over it in sections.

**Subclassing**

Subclassing refers to creating a new class that is based on another class (called the superclass, or parent class). This allows our new class to inherit all of the characteristics of the superclass. In our example above we specify that Developer is a subclass of Person in the class definition itself.

class Developer: Person { // Here we specify that Person is the "Superclass"

// Developer definition here

}

**Initialization**

When we create a subclass, it automatically gains access to the superclass's init method. But what if we want to add logic to the initialization of just the subclass? In our Developer example, we want to also specify a favoriteLanguage property in the subclass' initialization. We do not want the Person class to have a favoriteLanguage so we declare the property in the Developer class and create another init method.

// Within the Developer class

  var favoriteLanguage: String // Declaring the favoriteLanguage property in Developer means that

// it is only available in Developer and Developer's subclasses.

init(name: String, favoriteLanguage: String) { // Developer's init method takes in

// a name and a favoriteLanguage.

self.favoriteLanguage = favoriteLanguage // We set the favoriteLanguage property.

// We then call the superclass' init method and pass it the name

super.init(name: name) // property since the logic is already written in Person's init.

                                     // We reference the superclass through the 'super' variable.

}

The main thing to remember when creating an init method in a subclass is the use of

super.init

which allows us to call upon the superclass' initialization method. Note that super.init should be called after the initialization of the child class properties, but before we override any inherited properties.

**Overriding Superclass Methods**

By default, a subclass has access to all of the superclass' methods and properties. However, what if we wanted to override a particular method's functionality. Fortunately Swift allows us to prefix a method name with the override keyword to specify that we will be overriding the default superclass functionality. Let's see this in action:

override func speak() { // We specify that we are overriding the speak function.

print("Hello! I am a Developer! My name is \(self.name)")

}

Now when we called speak on an instance of Developer it will run the Developer's speak function rather than the superclass speak function.

**Note:**

The override keyword is needed for the init method if it has the same identity as the super init method.

**Chaining**

It is common practice to return the instance when writing object methods. This enables us the chain method calls on a single line. A general rule of thumb is, if the method doesn't have a required return value (like a count of something) you should return the object to enable chaining. In loosely typed languages this is simple, you just return this or self (whatever the keyword is). In swift we have to do a little more.

class Building {

    var health = 100

    func takeDamage(\_ damage: Int) -> Building { // Note the return type

        health -= damage

        return self                              // Note the returned value is self

    }

}

var building = Building()

building.takeDamage(10).takeDamage(20)

Above is a simple example of what is takes to chain in swift. This work fine for individual classes but what about classes we plan to subclass? If we subclass Building and make the House class, the takeDamage method will still return a Building type.

class House: Building {

    var door = "closed"

    func openDoor() -> House {

        door = "open"

        return self

    }

}

var house = House()

house.takeDamage(10).openDoor() // this will throw an error because Buildings don't have an openDoor method

This is very annoying because we would have to cast it back to a House to chain methods. Swift has a solution to this problem though, instead of defining the return type to be the class name, we define is as Self with a capital S. This means, the return type is the same as the type of the object that called the method.

// in the Building class

func takeDamage(\_ damage: Int) -> Self { // we changed the return type to Self

// code

}

// This line will work now because the object calling the takeDamage method is a House, so the return type is House

house.takeDamage(10).openDoor()

# Initialization

**Ready to construct an object?** You already know how to do this, but it's time to take a deep dive.

Initialization in Swift is about what happens when you create a new instance of a named type.

They’re used to “construct” an instance of the given type such as the following:

let name = String()

Initializers (also called constructors) are called to create a new instance of a particular type.

The most simple form of this could be:

* The initializer is like an instance method with no parameters, written using the init keyword:

*Remember that an initializer is like any other function, except that the function name is the same as the name of the type.*

Things to remember:

*You can't use an instance until it is fully initialized. That includes accessing properties, setting properties and calling methods.*

During initialization, we take care of the initial values of stored properties for named types: classes, structures, and enumerations.

An initializer must assign a value to every single stored property that does not have a default value, or else you’ll get a compiler error. Remember that optional variables automatically have a default value of nil.

## Default, Memberwise, Failable & Required Initializers

Swift provides us a lot of power and convenience with initializers, but with this comes complexity that we need to understand! Let's take a look at Default, Memberwise, Failable & Required Initializers in Swift.

#### Default

Swift provides a default initializer for any structure or class that provides default values for all of its properties and doesn’t provide at least one initializer itself. The default initializer simply creates a new instance with all of its properties set to their default values.

In the default initializer, the name of the type is followed by empty parentheses

You can use default initializers when your types either don’t have any stored properties, or all of the type’s stored properties have default values.

You can see this if we create a Player struct that has one property:

struct Player {

var name: String

}

When we create one of those structs, we must provide a username:

var player = Player(name: "michael\_jordan")

We can provide our own initializer to replace the default one. For example, we might want to create all new users as “John Doe” and print a message, like this:

struct Player {

var name: String

init() {

name = "John Doe"

print("Adding a new player to the team")

}

}

#### Memberwise

Swift structures (and only structures) automatically generate a memberwise initializer. This means you get a ready-made initializer for all the stored properties that don’t have default values.

The memberwise initializer is a nice shorthand way to initialize the member properties of new structure instances.

*Initial values for the properties of the new instance can be passed to the memberwise initializer by name.*

You only get a memberwise initializer if a structure does not define any initializers. As soon as you define an initializer, you lose the automatic memberwise initializer.

Basically, Swift will help you out in the beginning! But as soon as you add your own initializer, it assumes you want it to get out of the way.

Using an **extension** for this is really clever so we can create instances in two ways!

struct Student {

var firstName: String

var lastName: String

}

extension Student {

init(name: String) {

let split = name.components(separatedBy: " ")

firstName = split.first ?? ""

lastName = split.last ?? ""

}

}

let dojoStudent1 = Student(firstName: "Taylor", lastName: "Swift")

let dojoStudent2 = Student(name: "Taylor Swift")

#### Failable

It’s sometimes a great idea to define a class, structure, or enumeration that allows for initialization that can fail. You could even think of this as "optional initializers"

A failable initializer creates an optional value of the type it initializes.

You can declare your own failable initializer with init?()

struct Student {

var id: String

init?(id: String) {

if id.count == 10 {

self.id = id

} else {

return nil

}

}

}

* We are creating a new student with a 10 digit ID. If anything other than a 10 digit string is used, we will return nil.

## A quick note on convenience initializers

What's the difference between designated initializers and convenience initializers?

**Designated initializers** perform the actual initialization for a class’s properties. We know all about these already!

**Convenience initializers** let you program defaults into simpler initializers with less input parameters, and they hand off the actual initialization back to the designated initializers which we are used to dealing with.

*You don’t have to have any convenience initializers if you wish, but they are quite useful in certain situations.*

Here is an example from the Apple Developer Documentation:

The designated initializer is the init(name: String), it ensures that all stored properties are initialized.

The init() convenience initializer, taking no argument, automatically sets the value of name stored property to not sure yet by using the designated initializer.

class Food {

let name: String

// designated initializer

init(name: String) {

self.name = name

}

// convenience initializer

convenience init() {

self.init(name: "not sure yet")

}

}

let food = Food(name: "Cheese") // name will be "Cheese"

let food = Food() // name will be "not sure yet"