Self and self in Swift

Aasif Khan

By | Last Updated on August 22nd, 2023 9:00 am | 4-min read

In Swift, “self” typically refers to the current object within a class or struct. We’ve got more selves though: self, Self and even .self. What does it all mean? Get ready for a bit of introspection and metaprogramming – we’re going to discover the Self in Swift. Don’t worry, no New Age stuff involved!

Here’s what we’ll discuss:

- What’s self and Self in Swift
- How to use self when you’re working with classes or closures
- When you need to use self, and when it’s implicit
- What Self refers to when working with protocols and extensions
- How Self is often a placeholder for something else
- Why Self is needed when you don’t know the concrete type

Table of Contents

“self” as an Object in Swift

Let’s start with the most simple and most common occurrence of self in Swift. It’s the “self” that’s written with in lowercase, and it’s generally followed by a dot and a property or function name.

```
self.age = 42
self.walk()
```

This self is a reference to the current object (“instance”) of a class (or struct), within that class. You can work with self as if it’s a variable or object within the class (or struct), that is the class (or struct). It’s an essential concept of Object-Oriented Programming (OOP).

Check this out:

```
struct Teapot
{
var canTalk:Bool

init(canTalk: Bool) {
self.canTalk = canTalk
}
}

let potts = Teapot(canTalk: true)
```

What’s going on in the above code?

We’ve created a struct called Teapot, with one property canTalk of type Bool. The struct has an initializer that accepts one parameter named canTalk, which is assigned to the property canTalk of the struct.

Explicit “self”

Inside the initializer init(canTalk:), we’re making use of the self keyword. We’re taking the canTalk constant, which is provided as a parameter to the function, and assign its value to the canTalk property of the struct.

But there’s more... Within the Teapot struct, self refers to the current instance of a Teapot object. In the above code, self is identical to the potts

constant on the last line. In fact, self is identical to any object as seen from the perspective of within the struct or class.

In the above code, you can also see a great use case for using self in a function. Because both the initializer parameter and the property have the same canTalk name, we use self.canTalk to refer to the property and canTalk (without self.) to refer to the parameter. The use of self takes away any confusion about what you're referring to.

Let's make a quick analogy: When you and I are talking about my funny ears, you'll have to say "Reinder's funny ears" and I can say "My funny ears". I'm talking about myself, i.e. self.ears, whereas you're talking about reinder.ears. In this scenario, self === reinder, but within the context of a person, self.ears could be Bob's ears – or anyone's – from Bob's own inside perspective.

Implicit "self"

In the previous example we've used self explicitly to set a property from inside the struct. In Swift, the use of self is often implicit though. Check out the following example:

```
struct Teapot
{
var canTalk:Bool

func poke() {
if canTalk == true {
print("I'M A TEAPOT!!")
}
}
}

let potts = Teapot(canTalk: true)
```

potts.poke()

This is one talkative teapot! When you poke it, it'll shout I'M A TEAPOT!! if canTalk is true.

But wait... Why aren't we using self.canTalk == true in the conditional? That's because self, i.e. the reference to the current object of Teapot, is implicit. Swift knows we mean the canTalk property of the current object, and it saves us some coding time by making this assumption.

When Do You Use "self" in Swift?

When do you use self in Swift?

- If you're working with an outlet property in a view controller, like self.usernameLabel?.text = (You can often omit self., though!)
- If you're working with a property, and there's a naming conflict, so you use self.canTalk to explicitly tell Swift you're referring to the property
- If you're working with closures, and you want to capture a reference to a current class instance, and the closure escapes (Swift 5.3+), you'll also use self explicitly
- Since Swift 5.3, most notably in SwiftUI, you don't have to explicitly refer to self when capturing it in a closure if it's a value type, such as a struct

A rule of thumb is that you only need to use self when you must. For example, to resolve naming conflicts or to make capture semantics in closures clear. You can prepend all properties in your code with self., but that doesn't make it any clearer.

A counter-intuitive way to work with self is inside extensions. Check this out:

```
extension String {
func isPalindrome() -> Bool {
return self == String(self.reversed())
}
}
```

The above isPalindrome() function returns true when a String is a palindrome, i.e. it reads the same left-to-right as right-to-left. The function is part of an extension, which means the function is “tacked on” the already present String type. The self inside the function refers, of course, to a String object as seen from within the String struct.

What’s counter-intuitive about this approach is that, from the perspective of the coder, there aren’t any strings yet. You’re working within the struct; within the extension. You may be compelled to start writing a function like isPalindrome(string:), with a parameter and without making use of self, at first. You can use the string itself though, with self.

For now though, let’s work with the idea that self within a class or struct refers to the current instance of that class (or struct) from within the class (or struct). It literally refers to the “self”, seen from within. Awesome!

Note: Us humans, we often think that we are our thoughts. When a thought like “I’m not good at this” pops up in your mind, you may believe that thought, identify with it, and see it as “true”. Sounds familiar? Then give this exercise a try:

1. Sit down some place quiet and close your eyes
2. Keep your eyes closed for a while – then, a thought pops up in your mind
3. Watch the thought fly by, like a cloud in the sky, or a leaf on a river
4. Practice a bit with watching the thoughts go by, and see if you can let them go once they’ve flown or floated by
5. Open your eyes again when you’re ready

Now, ask yourself: Who's doing the watching? It's you, watching yourself and your thoughts. If that's true, then what is "me", "I" and "my thoughts"? Your thoughts may not be as solid as you think. This may lead you to believe that you are not your thoughts, which I think is true. You can watch your thoughts, even watch how you respond to them, and that could give you a sense of clarity, confidence and calm. This exercise is called mindfulness meditation.

"Self" as a Type in Swift

In the previous section, we've looked at the lowercase self that refers to an object. Next up, we're going to discuss the Self that refers to a type. This "Self" has an uppercase "S", which is how you can tell it apart from the lowercase self.

In Swift, Self refers to a type – usually the current type in the current context. Just as lowercase self can mean any current object, uppercase Self can mean just about any current type. It's intriguing!

"Self" in Classes and Structs

Let's start with a simple example:

struct Teapot
{

static func createTeapots(count: Int, canTalk: Bool) -> [Self] {
return (0..Meta Note: You can refer to Teapot.self, i.e. some class.self, which is the type itself. This is called a metatype; a type of a type. The type of Teapot.self is Teapot.Type. You could see Teapot.self as a reference or value of the type itself, that you can pass around in your code. An example is tableView.register(UITableViewCell.self, forCellReuseIdentifier:) for table views. You indicate that you want to register the type itself with the table view, which is UITableViewCell.self. (If

you want to create a reference to a property, check out keypaths.)

“Self” in Protocols and Extensions

But what if you’re in a context where the concrete type you’re using is not that clear? For example with generics, protocols and extensions. They can refer to a multitude of types.

Check out the following example:

```
extension Numeric {
func squared() -> Self {
return self * self
}
}

let number = 42
print(number.squared()) // 1764
```

The above code defines a protocol extension for Numeric, which is the protocol that all “numeric” types conform to, like Double and Int. The squared() function returns the power of 2 of a given number; 1764 for 42 in the above example.

This extension for Numeric is made for a protocol type, which means that the squared() function is added to any type that conforms to the Numeric protocol. As such, we don’t know what type we’ll be multiplying within the squared() function. That’s self-evident for the self * self code (pun intended), but what should be the return type of the squared() function?

It’s Self, with an uppercase “S”. In this context, Self refers to the type that conforms to the Numeric protocol. In the example, Self would be the concrete type Int, because 42 is an integer. When calling squared() on a Double, Self would be Double. We can’t hard-code a concrete type here,

because of the protocol extension. That's Self-as-a-type!

OK, let's look at another example. It covers the same principles as before, but it's a bit more complex. Check it out:

```
extension Sequence where Element: Numeric {
func squareAll() -> [Self.Element] {
return map { $0 * $0 }
}
}

print((0...10).squareAll())
// Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The above code defines an extension for the Sequence, which is a low-level protocol for collections, arrays, dictionaries, ranges, etcetera. The where keyword is used to limit this extension for types where Element conforms to Numeric. You can use squareAll() with any sequence or collection of numbers.

Within the Sequence protocol, Element is an associated type. You can see this as a generic placeholder for the protocol. When the protocol is used in a concrete type like Array, Element would be the type of the items in the array.

The squareAll() function will calculate the power of 2 for all numbers in the sequence, regardless of whether those numbers are integers, doubles or belong to a range. The return value of the function is an array of... an array of what, exactly?

Let's focus on the return type of the squareAll() function, which is [Self.Element]. We want to return something, but we don't know what. We've declared a protocol extension, which means that the "input" for squareAll() can be any sequence whose items conform to Numeric.

You cannot declare Self as the return type, because that's the type of the sequence. Also, the map(_:) function is going to return an array. If you map over a range, like we're doing, you're going to get an array back. So squareAll() -> Self or -> [Self] isn't going to work.

What we can do, however, is use Self.Element. Within Sequence, Element is the type of the items in the sequence. For the range 0...10, the Element corresponds to integers. In an array of doubles, Sequence is the protocol, Self is the array, and Self.Element is Double.

We're 100% sure that map(_:) returns an array, we only need to know the type of the items in the array. We can't be 100% what that is, because Sequence where Element is Numeric can be so many things. Fortunately, we can use a placeholder or "subtype" Self.Element. Neat!

The code we've discussed in this section makes use of generics and placeholders. Placeholders are just that: they get replaced by a concrete type when Swift compiles your code. When you're using squareAll() with an array of doubles, for example, [Self.Element] becomes the concrete type [Double]. Swift will compile additional variations of that function, depending on the types you use in your code. The whole reason we have those placeholders in the first place, is because it allows you to create a squareAll() function for dozens of types: arrays, dictionaries, ranges, etcetera. This syntax is what makes Swift powerful, albeit confusingly powerful at first.

Further Reading

In this app development tutorial, we've discussed what self and Self are in Swift and how you can use them. Here's a quick summary:

- self (lowercase "s") refers to the current object of a class or struct from within that class or struct — think self.funnyEars from within vs. bob.funnyEars from outside

- Self (uppercase “S”) refers to the current type with a context, and it’s often a placeholder for another type — think Self.Element in an array of integer numbers

Create Your App Now

Related Articles

- Why Do Hospitals Need a Mobile App?
- Understanding The «Unexpectedly found nil while unwrapping an Optional value» Error
- How To Make An App Like Cash App in 2022?
- Major Google Sheets Integrations to Automate Business Workflows
- 11 Best Trello Integrations to Use in 2021
- Appy Pie – Data: Women App Creation
- Workflow Automation Planning: Tips, Tricks & Strategies for Efficient Automated Workflows
- Best Mortgage Calculator Apps [Loan Calculator for iPhone & Android]
- How to create an app like Yelp?
- How To Launch Your App