Chapter 31: Associated Objects

Section 31.1: Property, in a protocol extension, achieved using associated object

In Swift, protocol extensions cannot have true properties.

However, in practice you can use the "associated object" technique. The result is almost exactly like a "real" property.

Here is the exact technique for adding an "associated object" to a protocol extension:

Fundamentally, you use the objective-c "objc_getAssociatedObject" and _set calls.

The basic calls are:

```
get {
   return objc_getAssociatedObject(self, & _Handle) as! YourType
   }
set {
   objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
    }
```

Here's a full example. The two critical points are:

1. In the protocol, you must use ": class" to avoid the mutation problem.
2. In the extension, you must use "where Self:UIViewController" (or whatever appropriate class) to give the confirming type.

So, for an example property "p":

```
import Foundation
import UIKit
import ObjectiveC          // don't forget this

var _Handle: UInt8 = 42    // it can be any value

protocol Able: class {
    var click:UIView? { get set }
    var x:CGFloat? { get set }
    // note that you >> do not << declare p here
}

extension Able where Self:UIViewController {

    var p:YourType { // YourType might be, say, an Enum
      get {
          return objc_getAssociatedObject(self, & _Handle) as! YourType
          // HOWEVER, SEE BELOW
          }
      set {
          objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
          // often, you'll want to run some sort of "setter" here...
          __setter()
          }
    }
```

```
    func __setter() { something = p.blah() }

    func someOtherExtensionFunction() { p.blah() }
    // it's ok to use "p" inside other extension functions,
    // and you can use p anywhere in the conforming class
}
```

In any conforming class, you have now "added" the property "p":

You can use "p" just as you would use any ordinary property in the conforming class. Example:

```
class Clock:UIViewController, Able {
    var u:Int = 0

    func blah() {
      u = ...
      ... = u
      // use "p" as you would any normal property
      p = ...
      ... = p
    }

    override func viewDidLoad() {
      super.viewDidLoad()
      pm = .none // "p" MUST be "initialized" somewhere in Clock
    }
}
```

Note. You MUST initialize the pseudo-property.

Xcode **will *not* enforce** you initializing "p" in the conforming class.

It is essential that you initialize "p", perhaps in viewDidLoad of the confirming class.

It is worth remembering that p is actually just a ***computed property***. p is actually just two functions, with syntactic sugar. There is no p "variable" anywhere: the compiler does not "assign some memory for p" in any sense. For this reason, it is meaningless to expect Xcode to enforce "initializing p".

Indeed, to speak more accurately, you must remember to "use p for the first time, as if you were initializing it". (Again, that would very likely be in your viewDidLoad code.)

Regarding the getter as such.

Note that it **will crash** if the getter is called before a value for "p" is set.

To avoid that, consider code such as:

```
    get {
        let g = objc_getAssociatedObject(self, &_Handle)
        if (g == nil) {
            objc_setAssociatedObject(self, &_Handle, _default initial value_, .OBJC_ASSOCIATION)
            return _default initial value_
        }
        return objc_getAssociatedObject(self, &_Handle) as! YourType
        }
```

To repeat. Xcode **will *not* enforce** you initializing p in the conforming class. It is essential that you initialize p, say in

viewDidLoad of the conforming class.

Making the code simpler...

You may wish to use these two global functions:

```
func _aoGet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ safeValue: Any!)->Any! {
    let g = objc_getAssociatedObject(ss, handlePointer)
    if (g == nil) {
        objc_setAssociatedObject(ss, handlePointer, safeValue, .OBJC_ASSOCIATION_RETAIN)
        return safeValue
    }
    return objc_getAssociatedObject(ss, handlePointer)
}

func _aoSet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ val: Any!) {
    objc_setAssociatedObject(ss, handlePointer, val, .OBJC_ASSOCIATION_RETAIN)
}
```

Note that they do nothing, whatsoever, other than save typing and make the code more readable. (They are essentially macros or inline functions.)

Your code then becomes:

```
protocol PMable: class {
    var click:UILabel? { get set } // ordinary properties here
}

var _pHandle: UInt8 = 321

extension PMable where Self:UIViewController {

    var p:P {
        get {
            return _aoGet(self, &_pHandle, P() ) as! P
        }
        set {
            _aoSet(self, &_pHandle, newValue)
            __pmSetter()
        }
    }

    func __pmSetter() {
        click!.text = String(p)
    }

    func someFunction() {
        p.blah()
    }
}
```

(In the example at _aoGet, P is initalizable: instead of P() you could use "", 0, or any default value.)