Understanding Optional Values in Swift

Craig A. Will
March, 2015

A major source of errors in programming occurs when a variable is expected to have a value but there is in fact no value. The problem is not that this is particularly difficult to deal with. The problem is that it is often simply *not* dealt with. The code might be prototype code that suddenly is deemed to be production code. Or it is code where the programmer defers dealing with the problem and then forgets, or disappears to work on some other project. The consequences vary from some function simply not working in some circumstances to an app crashing. In some cases the problem only manifests itself at a later time and is difficult to debug.

It is common for iOS and Mac APIs to return a `nil` (that is, the absence of a value) when a value, such as the contents of a file from a remote server or on the device's own file system, is not available. Some functions will return `nil` rather than a value: For example, the function toInt() takes a number coded as a string (e.g., "1234") and converts it to an integer. However, if the function is given a string like "abcd", it will return a `nil`. Similarly, an attempt to access a dictionary with a key that has no associated value will return a `nil`.

Swift has a capability for avoiding and quickly catching errors resulting from the absence of values. This capability is implemented in the form of what are known as *optional values,* or *optionals.*

If *type safety* refers to designing a language that tried to help programmers avoid making errors about the correct data type, then defining a language with capabilities (such as optionals) that help programmers avoid making errors when variables lack values might be called *value safety.*

Ordinary variables, known as non optionals, in this scheme, are *required* to have values. They are required to have an initial value. If an attempt is later made to set them to `nil`, a runtime error will result.

In order for a variable to be allowed to not contain a value, that is, be set to `nil`, that variable must have been declared as some form of "optional" data type. (Constants aren't involved in this discussion, because they can't change value and thus can't suddenly become nil unexpectedly.)

The fact that a runtime error will result from setting a non optional variable to `nil` and that `nil` can only be set to a variable declared with an optional data type might be considered a first line of defense (of two) in avoiding this kind of error.

The First Line of Defense--Only Optional Variables Can Fail to Have a Value

Every data type has an optional version of it that allows a variable of that type to be set to `nil`--that is, to not have a value.

Thus, an integer data type has an optional version formally defined as:

```
Optional<Int>
```

and known as an "optional integer".

You won't actually see a notation like `Optional<Int>` much. What you will commonly see is an alias (meaning the same thing) that consists of the type name followed by a question mark--In this case `Int?`.

Thus, the following code will declare a variable as an optional integer:

```
var n:Int? = 5
var n:Int? = nil
```

In the first line `n` must be explicitly declared as an optional with the `:Int?`. If n is simply declared with no type annotation and then set to 5 the type inferred will just be `Int`, a non optional.

In the case of the second line, an explicit declaration is also required, because although setting it to `nil` makes it clear that it is an optional, the compiler would otherwise not know that the type desired was an integer.

When a variable with an optional type is set to a value that value is *wrapped.* This means that it cannot be directly accessed in the normal way. To access it it must be *unwrapped.*

The Second Line of Defense--Values in Optionals Must Be Unwrapped

The second line of defense in using optionals is the necessity to unwrap them before use. If `n` is an optional and there is an attempt to execute the following code, the program will have a runtime error.

```
var n:Int? = 5
var p = n // Program will crash
```

The rule is that a reference to an optional data type that is made without unwrapping it will cause a runtime error as a way of reminding the programmer that some effort must be put into ensuring that the value is actually there.

The simplest (but dangerous) way to do this is by *forced unwrapping.* This is done by appending an exclamation point to the variable being unwrapped:

```
var n:Int? = 5
var p = n! // works but dangerous
```

This works but is not particularly recommended and should only be used if you are *absolutely sure* that the value is not `nil`. The problem is, if the value is `nil` a runtime error will result.

The best practice here is to use what is known as *optional binding:*

```
var n:Int? = 5
if let p = n {
    println("The unwrapped value is\(p)")
}
else {
    println("There is no value")
}
```

What this does is to set `p` to the value contained in the wrapped value `n`. The value of `p` is then tested in the if expression: if the value exists then the test succeeds and the first part of the `if` statement is executed; if there is no value the test fails and the second part of the `if` statement is executed.

Once the value of `p` has been set the value is available in an unwrapped form by referencing `p`. Thus, when it is referenced the value can be printed without any use of an exclamation point to unwrap it: it is already unwrapped.

This particular behavior is an idiom that only sets a value if the second variable is an optional type.

Note that this `if` statement and the ability to assign a value to a variable in it is against the general rule in Swift that assignment statements cannot be made in `if` clauses. This is allowed because the constant is declared within the `if` statement and thus its scope is local to it. It is thus relatively safe. The usual practice is to use a constant. It is also possible to use a variable, which can be useful in some circumstances. This behavior also works with `while` statements as well as `if` statements.

In optional binding it is common to use the same variable name in binding as was used

for the optional value.

```
var n:Int? = 5
if let n = n {
    println("The unwrapped value is\(n)")
}
else {
    println("There is no value")
}
```

Optional binding is a better practice than testing for `nil`, which I will describe just below, because it is a little safer. I'm just describing this alternative here for completeness so that you understand it. But you should use optional binding in preference to testing for `nil`.

This alternative simply tests whether the variable has a value or not:

```
var m:Int? = 5
if m!=nil {
    println("xxx\(m!)")
}
else {
    println("m has no value")
}
```

This is just a simple test of whether the variable has a value or not. Note that if there is a value, it is necessary to do forced unwrapping (with an !) on the variable to extract the value. The reason that this is considered less safe than optional binding is that the programmer may accidentally omit the exclamation point and thus cause a runtime error.

Implicitly Unwrapped Optionals

An alternative to the basic wrapped optional is an *implicitly unwrapped optional.* This is used only when you are absolutely sure that a variable has a value. It is usually used in situations such as a property defined in a class as a `nil`, but actually initialized with a value as part of an init(), perhaps when an instance is created with a parameter that sets the property. Care is then taken to not thereafter set the property to `nil`.

It is also common for the Cocoa Touch API to return optionals that are guaranteed to have a value, and these can be declared as implicitly unwrapped optionals also.

A variable is defined as an implicitly unwrapped optional by using type annotation and providing an exclamation point ("!") just after its name:

```
var n:Int! = 5
```

```
var p = n
```

Here in a very simplified example the variable n is declared as an integer implicitly unwrapped optional with the value of 5 assigned. When the value is then assigned to the variable p, it is not necessary (or allowed) to do anything to unwrap it.

If, however, an implicitly unwrapped optional is accessed that does not have a value, there will be a runtime error.

A more realistic example is shown below:

```
class bird {
    var species:String! = nil
    init(species:String) {
        self.species = species
    }

let aParticularBird = bird(species:"Mallard Duck")
```

In this case we have a class that stores information about particular birds, namely, the name of their species. We define a property, `species`, as a variable with a type of an implicitly unwrapped optional string, with a value of `nil`. The initializer accepts a parameter when a new instance is created and sets its value (a string) to the species parameter.

What actually happens in the last line is that a new instance is created, and the property species is first initialized as `nil`. When the initializer is executed, however, the property is then set to the value of "Mallard Duck".

If this property is accessed, unwrapping is not required:

```
println("The species is \(aParticularBird.species)")
```

If it is later discovered that the species name is not correct, it is not allowed to simply set the value back to `nil` (for unknown). If the correct species name is not available the instance must be deleted.

Nil Coalescing Operator

Another approach to optionals ensures that a variable has a value, either that contained in an optional variable or a predefined default value. It is quite safe because unwrapping is done automatically and there is no possibility of a runtime error.

This is done with a *nil coalescing operator.* This operator is a double question mark and is used with an optional variable and a default value.

```
    var errorMessageFromDisk:String? = "Bad File Sector"
    let defaultErrorMessage = "Something Bad Happened"

    let errorMessage = errorMessageFromDisk ??
defaultErrorMessage
```

Here the first line declares and sets an optional variable with an error message. This is what might reasonably come from a file system. However, the message is declared as an optional, because the file system might not always know the source of the error, in which case it would return a `nil`.

A default error message is defined, and then the use of the nil coalescing operator.

In the case shown, `errorMessage` would be set to "Bad File Sector" after it was unwrapped from the optional version. This is because the left hand side of the ?? nil coalescing operator has a value. However, suppose the variable `errorMessageFromDisk` was `nil`. In such a case, `errorMessage` would be set to the default, "Something Bad Happened", by the statement with the nil coalescing operator.

This is equivalent to the following, more verbose, and less safe statements:

```
    var errorMessageFromDisk:String? = "Bad File Sector"
    let defaultErrorMessage = "Something Bad Happened"

    if errorMessageFromDisk == nil {
        let errorMessage = "defaultErrorMessage" }
    else {
        let errorMessage = errorMessageFromDisk!}
```

The name of the operator is a little obscure. "Coalesce" means to "come together", "combine", or "form one mass or whole". Thus presumably the coalescing operator allows the original optional value and the value that gets set if the original value is nil to "come together" and form a new variable or constant that contains all of the information in one whole. Of course, lots of operations do this. The name isn't really very enlightening about what it does.

Optional Chaining

In situations in which there is more than one optional in a sequence of property

relationships to unwrap, it takes less code and is more readable to use *optional chaining* for the unwrapping of optionals.

This accomplishes the same thing as optional binding. For optional chaining to have any effect, all of the values that are marked as optionals in the chain must in fact have values. If any of them are `nil`, the optional chaining statement does nothing.

To refresh your memory about how optional binding works, we can see it here:

```
var breedOfDog:String? = "Border Collie"
```

We have an optional with a value; we now have to unwrap it:

```
let breed = breedOfDog {
    println(\("Breed of dog is \(breed)")
    }
```

This works fine if there is just one value to test for nil.

Now consider a more complex situation. Suppose we have a person who owns a dog, and `breedOfDog` is a property of the class `Dog`.

```
class Person {
    var dog:Dog?
}
class Dog {
    var breedOfDog?
}

var george:Person? = Person()
george.dog:Dog? = Dog()
george.dog.breedOfDog:String? = "Border Collie"
```

We've created an instance of a `Person`, and assigned it to a variable `george` that is an optional Person. We've also created an instance of a `Dog`, and assigned it to a property of the variable `george`. And we have created a value for a property of `Dog` that is itself a property of `george`.

All of these variables and properties have values, because we could otherwise not assign anything to them. However, they are all declared as optional versions of whatever type they are, and at any point after these values are assigned any or all of them could have `nil` assigned to them. Thus we need to take care in unwrapping them.

Before unwrapping the optional value of `breedOfDog`, then, we have to make sure that the others have values. This leads to relatively unwieldy optional binding nested if-let

sequences:

```
    if let person = george {
        if let dog = george.dog {
            if let breed = george.dog.breedOfDog {
                 println("If everything has a value,
breedOfDog is \(breed)")
            }
        }
    }
```

Optional chaining, in contrast, just requires the following:

```
    if let breed = george?dog?breedOfDog? {
        println("If everything has a value, breedOfDog is \
(breed)")
    }
```

The question mark at the end of each variable, known as the *chaining operator,* indicates that the property or variable in question is an optional, and the value should be unwrapped if it exists. If the value does not exist then the optional chaining should stop immediately.

If a particular variable or property in the chain is not an optional, it does not have a question mark at the end of it, and does not need to be unwrapped. Its value is assumed to exist because otherwise the system would have generated a runtime error.

Note that we could have simply used forced unwrapping, and an exclamation point, for each optional in the chain:

```
    var breed = george!dog!breedOfDog!
```

This is actually OK if you are sure that all of these variables/properties do indeed have values. But it is safer to use optional binding.

Some optional chains have even more optionals in the chain. We might have the class and relationship structure shown above, but also for each person have a House and StreetAddress and Street and City. Thus the chain might be:

```
   if let breed = city.street.streetAddress?house?person?
dog?breedOfDog? {
            println("If everything has a value, breedOfDog
is \(breed)")
    }
```

Note that city and street do not have question marks after them. They are not optionals. It is assumed in this structure that there is always a `city`, and always a certain `street`. But a given `streetAddress` may not exist, a given `house` may not exist, a given `person` may not exist, a given `dog` may not exist, and a given `breedOfDog` may not exist (or be known). These are all defined as optionals.

Any property in the property relationship chain may be referenced. It is only given a "?" at the end if it is an optional and must be unwrapped.

Optional chaining is common for relationship chains in apps themselves, and especially common when a relationship chain is retrieved from an API like Cocoa Touch.

Optional chaining can be used to write to optional values as well as to read from them, and they can be used for methods and subscripts as well as properties.

Optional Binding for Multiple Variables

In situations in which there are more than one optional values that need to be unwrapped at one time, this can be done, beginning in Swift 1.2, compactly with a simplified syntax.

(This is based in part on an example by Nate Cook at NSHipster.com, and is not believed to be covered by Apple's NDA for Swift 1.2.)

Suppose that a, b, and c are all declared as optionals and have all been assigned a true value. They can be unwrapped by the code below. First, the optional declarations:

```
    var a:Int? = 5
    var b:Int? = 6
    var c:Int? = 7
    var d:Int? = 8
```

Then the code:

```
    if let a=a, b=b, c=c, d=d {
        println("All of the optionals a, b, c, and d have
values")
    }
```

The syntax above does the same thing as the more wordy:

```
    if let a = a {
        if let b = b {
            if let c = c {
                if let d = d {
```

```
                println("All of the optionals a, b, c,
and d have values")
            }
        }
    }
}
```

It's also possible to add a where clause to do any additional test beyond making sure that ll of the optionals have values:

```
var a:Int? = 5
var b:Int? = 6
var c:Int? = 7
var d:Int? = 8
var m = 5
if let a=a, b=b, c=c, d=d where m > 4 {
    println("M is > 4 and all of the optionals a, b, c,
and d have values")
}
```