The Guard Statement in Swift 2

June 17, 2015

Often when working with Swift Optionals, we will want to perform an action if the Optional contains a value, or do something else if it does not. This is done with Optional

Binding with the "if let" syntax, which lets us test for wether it contains a value, and if it does, binds it to a constant that can be used later in the if statement. This works great for many cases, but it does put the emphasis on the positive case. If there is a value, do this, otherwise do that. Now if the code to run when there is a value is short, you can easily see what the "else" clause is tied to. However, if the code to run when there is a value is long, then you might need to do some scrolling to see the associated else clause.

This is where the new guard statement comes in handy, newly introduced with Swift 2. The guard statement puts the emphasis on the error condition. In this case, if there is no value, you can clearly see what will be done to deal with it.

What happens if there is a value? The rest of the current scope continues running like normal. Let's go into an example of how the guard statement can be helpful.

The Simple Example

We'll start with a simple form just to show exactly what Swift's new guard statement is, in comparison to its "if" statement counterpart.

Let's say that we have a function that takes a custom object that contains information about an iOS development blog, like the one below:

```
1 struct iOSDevelopmentBlog {
2     let name: String?
3     let URL: String?
4     let Author: String?
5 }
```

All of these Swift String properties are marked as optional. For the simplest form of our example, let's just read the name Swift String and right a response based on that. With if-let, you could do it like this:

```
1 func originalStyleComplimentAboutBlog(blog: iOSDevelopmentBlog) {
2
3     if let blogName = blog.name {
4         print("The \(blogName) blog is a great iOS Development Blog!")
5     } else {
```

Recent Posts

watchOS 2 Hello World App in Swift

Swift is Open Source!

API Availability Checking in Swift 2

Protocol Extensions in Swift 2

Println Is Now Print, and Do-while Is Now Repeat-While in Swift 2

Sponsored Links

```
6         print("I don't know the name of this blog, but it's a good one!")
7     }
8
9 }
```

That seems simple enough. Check whether the "name" property has a value, and use it in the compliment string that is printed out, otherwise, print some general statement about this blog. To write the equivalent statement using Swift 2's new guard statement syntax, the code would look like this:

```
1  func guardStyleComplimentAboutBlog(blog: iOSDevelopmentBlog) {
2
3      guard let blogName = blog.name else {
4          print("I don't know the name of this blog, but it's a good one!")
5          return
6      }
7
8      print("The \(blogName) blog is a great iOS Development Blog!")
9
10 }
```

There are a few things to notice specifically about this. As mentioned before, we are binding the blog.name property to the "blogName" constant. However, if you are used to the if let syntax, and basically any use of Swift Optionals prior to Swift 2, you will notice something REALLY weird. We're using that optionally bound constant OUTSIDE of the scope of the conditional. In an if-let style, we could only use it in the brackets after the "if" statement, in the "true" case.

For the guard statement, the "true" case, is the entire rest of the function. The guard statement MUST be written with an else clause, because that's all it is, really. You try to optionally bind the value from a Swift Optional, and if it fails, run what is in the guard statement, otherwise, continue on. This lets the compiler know that it is okay to leave the optionally bound constant visible in the enclosing scope, because we have verified that it is indeed a valid value.

Also, note that after printing the general compliment in the guard statement, we returned from the function. Guard statements MUST have transfer control, in order to leave the scope it is written in. In this case, it must leave the function, via the "return" keyword.

Now, okay, looking at this example, why use the guard statement instead of the if-let statement? This one is even one line longer than the if-let style! Well, this was the simple example just to show how an if-let is converted to its guard statement counterpart. At this size, either is really fine. It is more of personal preference as to whether you want the emphasis on the "true" or "false" condition for such a small function as this. Now when things get a bit bigger, that's where Swift's new guard statement can really shine.

The Long Example

Now we have three properties in our iOSDevelopmentBlog struct, if we start using those with the if-let syntax, to make a similar compliment function, we get something like this:

```
1  func originalStyleLongComplimentAboutBlog(blog: iOSDevelopmentBlog) {
2
3      if let blogName = blog.name {
4          print("The \(blogName) blog is a great iOS Development Blog!")
5
6          if let blogAuthor = blog.Author {
7              print("It is written by \(blogAuthor).")
8
9              if let blogURL = blog.URL {
10                 print("Visit it at \(blogURL)")
11             } else {
12                 print("Search for it on your favorite on your favorite search engine.")
13             }
14         } else {
15             print("it is written by an unknown author.")
16         }
17     } else {
18         print("I don't know the name of this blog, but it's a good one!")
19     }
20
21 }
```

Now, if you look at it carefully, and go through it, what it's doing makes sense. It prints a line about each property, and if any of them are nil, the following properties are not read and the function returns early. The whole thing makes sense, and works…. but it is next to unreadable. The positive cases are….okay… you can see that if the name is there, it will print a line about the name, if the author is there it will print a line about the other. But which of the else clauses go with what if statement? You can follow the indentation to see which one…. but it is really unclear and confusing.

So with the following input:

```
let NSHipsterBlog = iOSDevelopmentBlog(name: "NSHipster", URL: "http://nshipster.com/", Aut
```

We would get the following Output:

> The NSHipster blog is a great iOS Development Blog!
> It is written by Nate Cook.
> Visit it at http://nshipster.com/

Now let's look at a functionally equivalent function using Swift's new guard statement:

```
func guardStyleLongComplimentAboutBlog(blog: iOSDevelopmentBlog) {

    guard let blogName = blog.name else {
        print("I don't know the name of this blog, but it's a good one!")
        return
    }

    print("The \(blogName) blog is a great iOS Development Blog!")

    guard let blogAuthor = blog.Author else {
        print("it is written by an unknown author.")
        return
    }

    print("It is written by \(blogAuthor).")

    guard let blogURL = blog.URL else {
        print("Search for it on your favorite on your favorite search engine.")
        return
    }

    print("Visit it at \(blogURL)")
}
```

Firstly, besides being inside the function itself, there are NO NESTED SCOPES! That makes things so much easier to read already. So now, you see that if the "name" property does not contain a Swift String, it prints the general statement about the name, and then returns. If it does find it, it prints the appropriate compliment. Then, if the author is nil, it prints something general about the other and returns, otherwise it will continue and

Categories

Class Reference

General

Interface Builder

My Apps

Objective-C

Swift

Syntax

Tutorial

Uncategorized

Archives

December 2015

July 2015

June 2015

April 2015

March 2015

mention the blog author.

It is about the same number of lines (not counting white space) as the if-let style, but it is really easy to tell which else clause goes with what conditional. It is almost as easy to tell which "true" case goes with which, since it comes right after the guard statement.

If you don't need individual else clauses for each conditional, both the if and guard statements can benefit from the compound optional bindings brought with Swift 1.2.

This would result in an if-let style like this:

```
func compoundOriginalStyleLongComplimentAboutBlog(blog: iOSDevelopmentBlog) {

    if let blogName = blog.name,
       let blogURL = blog.URL,
       let blogAuthor = blog.Author{
        print("The \(blogName) blog is a great iOS Development Blog!")
        print("It is written by \(blogAuthor).")
        print("")
        print("Visit it at \(blogURL)")
    } else {
        print("My information is incomplete, but I'm sure this iOS Development Blog is gre
    }
}
```

Or this using the new guard statement in Swift 2:

```
func compoundGuardStyleLongComplimentAboutBlog(blog: iOSDevelopmentBlog) {

    guard let blogName = blog.name,
          let blogURL = blog.URL,
          let blogAuthor = blog.Author else {
            print("My information is incomplete, but I'm sure this iOS Development Blog is
            return
    }

    print("The \(blogName) blog is a great iOS Development Blog!")
    print("It is written by \(blogAuthor).")
    print("")
    print("Visit it at \(blogURL)")
}
```

A Guard Statement MUST Transfer Control

Each guard statement has to end in something that will transfer control from the scope

that the guard statement is in. So for functions and methods, that would be "return", usually. If it is used in a loop like a for loop or while loop, you could use break, leaving it in the current function, but getting out of the loop. In loops you could also use "continue", which basically tells it to skip that iteration of the loop, and go to the next iteration, but remain in the loop.

So, if we have input and a loop like this:

```
let maybeNumbers: [Int?] = [3, 7, nil, 12, 40]

for maybeValue in maybeNumbers {

    guard let value = maybeValue else {
        print("No Value")
        break
    }

    print(value)
}
```

The output would be:

3
7
No Value

However, if the guard statement instead was:

```
guard let value = maybeValue else {
    print("No Value")
    continue
}
```

Where "continue" is used, we would get the output:

3
7
No Value
12

40

If things have REALLY gone wrong, you can also call functions that don't return, like Swift's preconditionFailure() function. Those functions must be marked with the @noreturn attribute, not just have no return value. In Swift, a function lacking a return value, but not marked as @noreturn actually returns (), the empty tuple.

The point is, the guard statement MUST get out of the block it currently is in.

Conclusion

All code in this post was tested against a playground in Xcode 7 Beta 2.

The guard statement is a great addition to Swift 2. It cleans up code that has a lot of else clauses to go along with their if statements. It keeps the code that deals with a broken conditional NEAR that conditional, as opposed to the end of the "true" clause of an "if" statement. It also has access to the compound optional bindings brought to us in Swift 1.2, making it even more compact.

The only thing I might ask to change about it, would be if we did not have to write "return" explicitly in each guard statement, if it could infer what control transfer command to use from context if omitted (similar to how you do not need to use "break" in a Swift Switch statement (unlike their C counterparts)). On the flip side though, having it written explicitly makes it REALLY obvious that you will not be continuing in that scope (or loop iteration) if that condition is met. They would also have to arbitrarily choose one for the loops, either "break" or "continue", for which there really isn't a good default. It would all depend on the point of that loop. I guess doing so would sacrifice some clarity and consistency for less lines of code to write for default cases, so that's probably why the control flow statement is written explicitly in the guard statement in Swift 2.

I hope you found this article helpful. If you did, please don't hesitate to share this post on

Twitter or your social media of choice, every share helps. Of course, if you have any questions, don't hesitate to contact me on the Contact Page, or on Twitter @CodingExplorer, and I'll see what I can do. Thanks!

Filed Under: Swift
Tagged With: optionals, properties, Swift