
STRANGELOVE
OR:
HOW I
LEARNED
TO STOP
WORRYING
AND
LOVE THE
U.S. AIR FORCE
U.S. AIR FORCE
COPYRIGHT (C) MCMLXIII HAWK FILMS LTD. ALL RIGHTS RESERVED

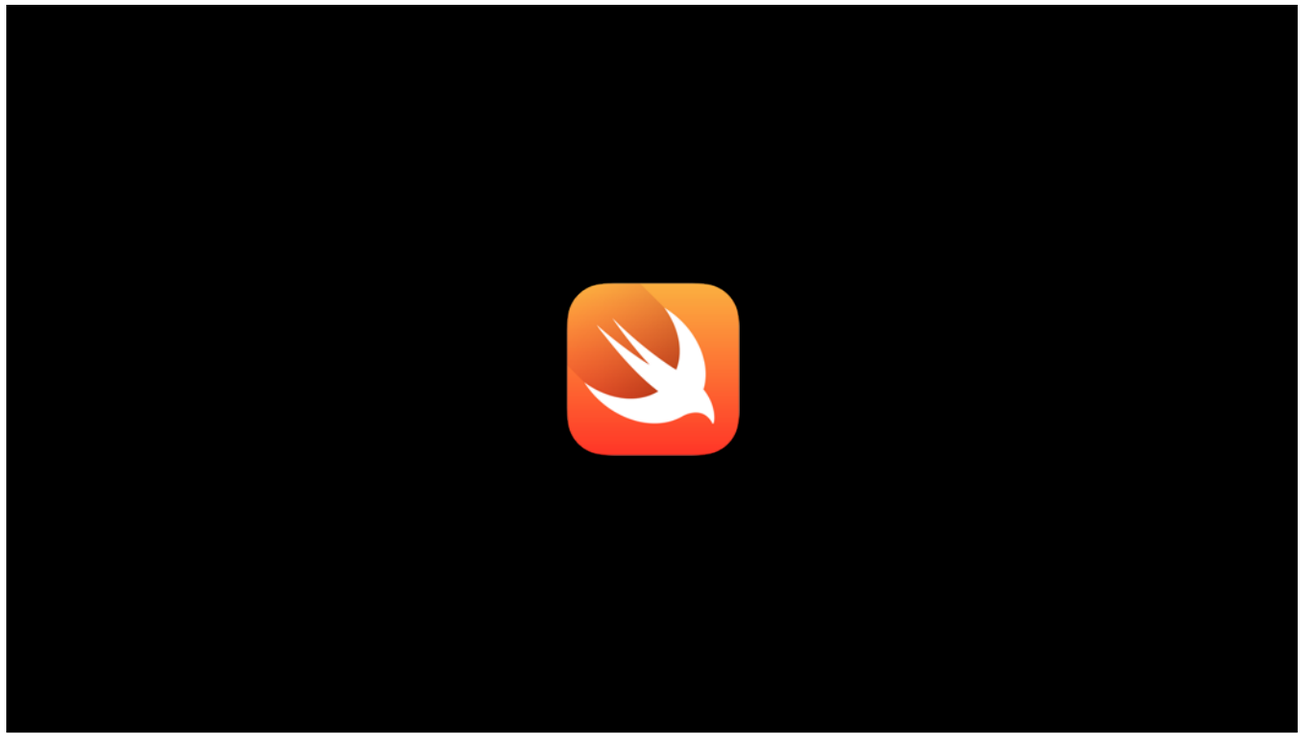
Hi.

Brent Royal-Gordon

TWITTER, GITHUB, ETC.

@brentdax

I’m Brent Royal-Gordon. I write Mac and iOS apps for, well, mostly myself.



So, today we’re gonna be talking about Swift. By show of hands: how many of you have written at least one line of Swift, or read part of the book, or seen some of the WWDC sessions?
[say something]
Okay, for those of you who haven’t: What is Swift?

Swift is what you'll write apps in for the next fifteen years

Swift is Apple's new language. It integrates almost seamlessly with Objective-C code and is meant to be easier to learn and easier to use. There's a good chance you'll start using it soon, and a very good chance you'll be using it for a long time if you stay on Apple's platforms.

```
class ViewController: UIViewController {
  override func viewDidAppear(animated: Bool) {
    super.viewDidAppear(animated)

    let alert = UIAlertView(
        title: "Hello world!",
        message: nil,
        delegate: nil,
        cancelButtonTitle: "Hi!")
    alert.show()
  }
}
```

I can’t teach you an entire language in half an hour, but if you haven’t seen Swift, here’s an example of it. I’m using iOS classes because they fit more interesting stuff on a single slide, but you could imagine very similar AppKit code.

As you can see, Swift is a little more terse, and in some ways more clean, than Objective-C, but this talk is mostly about its safety features. For example…

```
class
  override



        title:
        message:
        delegate:
        cancelButtonTitle:
    alert.
  }
}
```

…you might notice this “override” keyword. When you’re overriding a superclass method or property, you *must* include that keyword, and when you’re not, you *can’t* include it. That way, if you accidentally override a method you didn’t know was there—or try to override one but mistype the name—you’ll get an error.

```
class ViewController: UIViewController {
  override func viewDidAppear(animated: Bool) {
    super.viewDidAppear(animated)

    let alert = UIAlertView(
        title: "Hello world!",
        message: nil,
        delegate: nil,
        cancelButtonTitle: "Hi!")
    alert.show()
  }
}
```

Swift is full of design decisions like the “override” keyword—places where it forces you to say things explicitly in order to avoid mistakes. It’s like using Objective-C with all the warnings and errors turned on, but worse.

But not all of Swift’s safety features are like the “override” keyword. Sometimes, the mistakes it’s saving you from aren’t as obvious, and the solutions aren’t as simple as adding a keyword.

What’s forbidden

Why it’s forbidden

What you should do about it

That’s the subject of my talk today. I’m going to show you a few things you used to do in Objective-C that are now forbidden in Swift.

Then I’m going to show you an example of a bug it prevents,

and explain you what you should do instead.

No Implicit Conversions

Probably the first thing you'll notice when you start writing Swift is that you can't mix types like you used to.

```objc
- (NSInteger)numberOfRows {
    NSUInteger count = …;
    return count;
}
```

In Objective-C, you’re usually allowed to mix similar types freely.

```
- (   Integer
      UInteger count
   return count
}
```

Here, we’re using an NSUInteger as the return value of an NSInteger method. Objective-C converts that unsigned integer to a signed one and everything is just fine.

```swift
func numberOfRows() -> Int {
    let count: UInt = …
    return count
}
```


‘UInt’ is not convertible to ‘Int’

However, the equivalent in Swift is a compile-time error.

```
func numberOfRows() -> Int {
    let count: UInt = …
    return Int(count)
}
```

If you want to do that, you’ll have to construct an Int from the UInt.

Why would they do something like that?

```objc
- (BOOL)hasObjects {
    return self.objects.count;
}
```

Well, here’s something you’ve probably done in Objective-C: treat an integer like a boolean. After all, in C, 0 is false and all other values are true. So you return an integer as a boolean.

```
container.objects = arrayWith256Objects;

NSAssert(container.hasObjects,
    "This can’t fail…right? Right?");
```

Except that this doesn’t quite work the way you’d expect. BOOL is eight bits, so the 32- or 64-bit NSUInteger returned by “count” gets truncated. That means that any number divisible by two-to-the-eighth registers as false, not true!

There are lots of subtle issues like this when you’re automatically converting things, so Swift forbids it entirely. That means you have to convert things explicitly—which can be painful. But I know a few ways to make it easier.

```
// Before
let NUM_COLUMNS: Int = 4
gridSize.width *= CGFloat(NUM_COLUMNS)

// After
var NUM_COLUMNS: CGFloat = 4
gridSize.width *= NUM_COLUMNS
```

The first is to normalize types throughout your app. For instance, if you know a value is going to be used in layout, just make it a CGFloat to begin with. Swift already does this to the standard frameworks—NSUInteger values in the frameworks are treated as Ints by Swift. Your own code doesn't get that magic, but you can still change its types yourself.

```
…
var gridSize = cellSize

gridSize.width *= CGFloat(NUM_COLUMNS)
gridSize.height *=
    CGFloat(items.count / NUM_COLUMNS)
…
```

The second way you can work around this is to extract operations that encapsulate the conversions you need to do. Let’s say you need to lay out a grid of cells. The obvious way to do it will involve a couple of calculations with ugly conversions.

But if you think about it, this code isn’t very clear to begin with. It contains bare, unexplained math, presumably mixed with other code to handle those cells.

```
extension CGSize {
    func sizeOfCells(count: Int,
      columns: Int = 1) -> CGSize {
        return CGSize(
            width: width  * CGFloat(columns),
           height: height * CGFloat(count / columns)
        )
   }
}
```

We might be better off extracting a method to encapsulate that calculation. Here, we put it in an extension—basically a category—on CGSize (which, by the way, is a really handy thing you can do to a struct in Swift). Once we've done that…

```
…
let gridSize = cellSize.sizeOfCells(items.count,
        columns: NUM_COLUMNS)
…
```

The original line now expresses the intent of the code, not the way we chose to implement that intent. And the conversions are just an implementation detail.

```
func *= (inout lhs: CGFloat, rhs: Int) {
    lhs *= CGFloat(rhs)
}

gridSize.width *= NUM_COLUMNS
gridSize.height *= items.count / NUM_COLUMNS
```

The third solution is to hide the conversion behind an overloaded function or operator. Here, we write a custom multiply-and-assign operator that takes a CGFloat and an Int. The line with the actual math is nice and clean.

```
private func *= (inout lhs: CGFloat, rhs: Int) {
    lhs *= CGFloat(rhs)
}

gridSize.width *= NUM_COLUMNS
gridSize.height *= items.count / NUM_COLUMNS
```

Of course, overloading is considered dangerous for a reason. If you decide to do this, you should use access modifiers to reduce the overload's scope. Making the operator private means it's only available in this source file, so you're less likely to accidentally use it elsewhere.

With techniques like these, you can usually hide the noisiest conversions and write nice, simple code.

II

Swift Optionals

Another big change is Swift’s introduction of optional types.

```
NSView * myView = nil;
```

In Objective-C, any object reference can instead be set to “nil”, which roughly means “no object”.

```
var myView: NSView = nil
```

type 'NSView' does not conform to protocol 'NilLiteralConvertible'

But in Swift, you can’t. A plain old object reference can’t be nil.

What? Why not?

Well, I probably don’t have to elaborate on all the ways nils can go wrong. You’ve all experienced them dozens of times.

```
self.label.stringValue =
    @"This text doesn’t appear!";
```

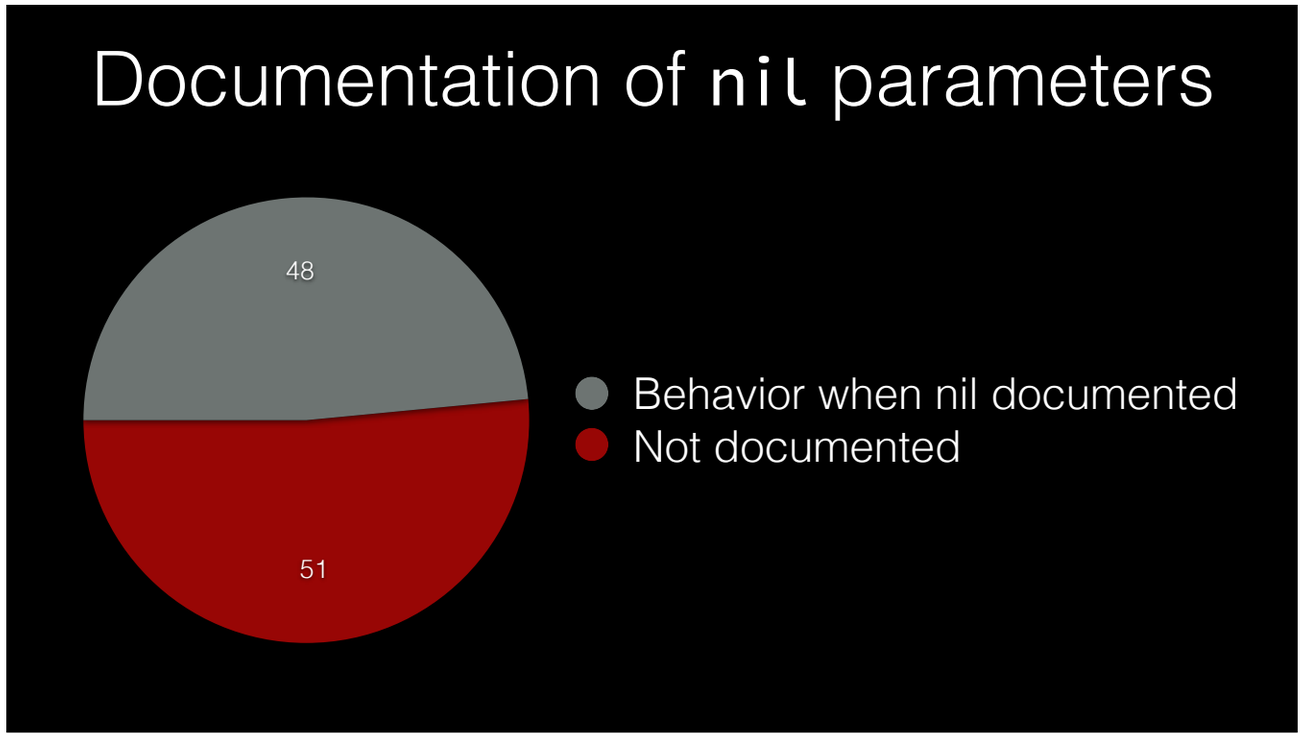
The outlet you didn’t connect that ignores everything you do to it.

```
NSData * data = [NSData dataWithContentsOfURL:url
    options:0 error:&error];
if(!data) {
    // TODO: handle error
}
[self doSomethingWithBuffer:data.bytes];
```

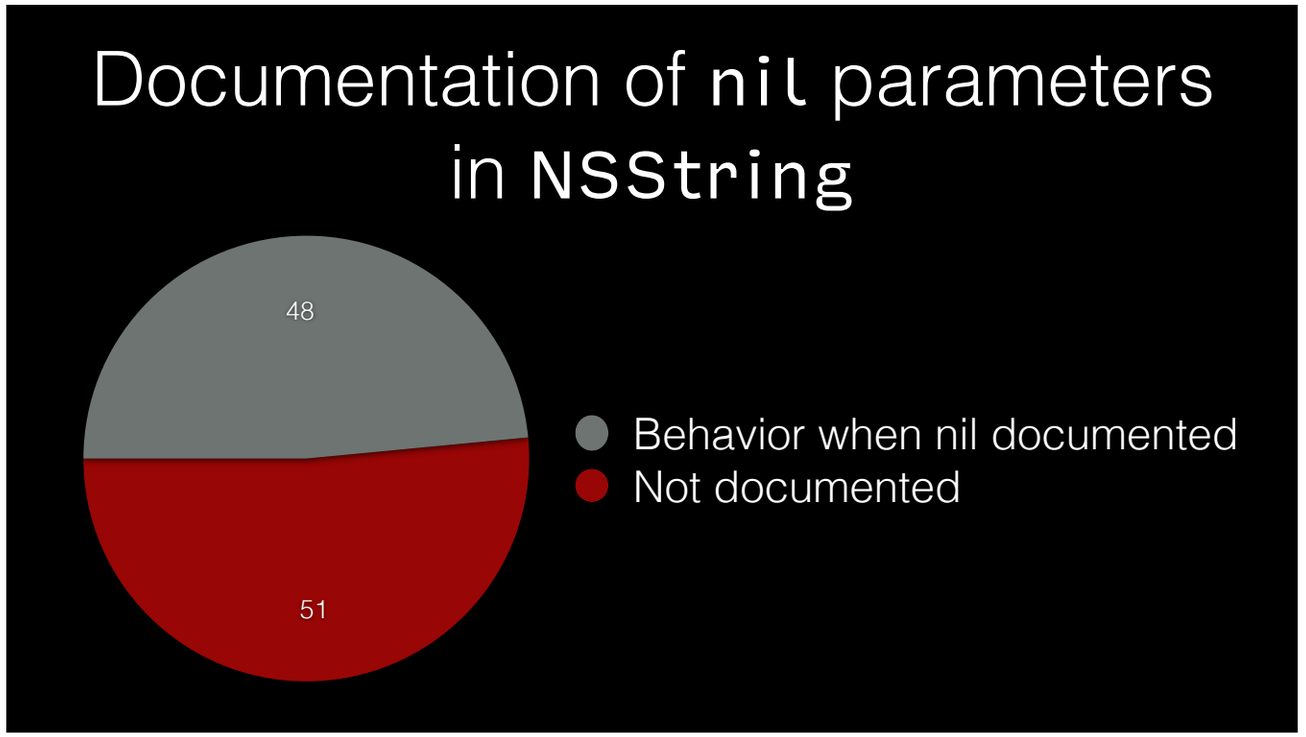
The error you forgot to do something about.

```
if([myString hasPrefix:possiblyNilString]) {
```

The method that you didn’t realize hates nils.



And it isn’t just you! I audited the documentation for every object or pointer parameter of every method in one framework class, and half of them didn’t describe what would happen if you passed nil or NULL. Half! And, well,



this isn’t some dusty corner of the frameworks, either. I’m not saying this to rag on Apple’s documentation—it’s some of the best I’ve worked with. But keeping track of things yourself just doesn’t work for this problem.

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965…. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”

– Tony Hoare, designer of the ALGOL type system
(and discoverer of Quicksort)

In fact, the man who designed the first strict type system has said he regrets including nil values. It’s just too dangerous to allow any alleged object to actually be the absence of an object.

```
var myView: NSView? = nil
```

So instead of allowing just any object reference to be nil, Swift uses what’s called an optional type. By adding a question mark to any type, you indicate that it’s just fine for that particular variable to be nil.

```
var myIndex: Int? = find(array, value)

if myIndex == nil {
    return
}
```

Unlike in Objective-C, optionals aren’t limited to just objects. You can mark any type as optional in Swift. That means that values like NSNotFound and -1 aren’t usually used as “none” markers in APIs designed for Swift—they’re just declared to return optional values, and return nil for “not found”. Objective-C APIs haven’t been changed to use optionals instead of NSNotFound, but maybe they will be in the future.

```swift
var myInt: Int? = nil

assert(myInt != 0)
assert(myInt != -1)
assert(myInt != NSNotFound)

assert(myInt == nil)
```

It’s worth keeping in mind that an optional Int that’s nil is not equal to zero, or -1, or NSNotFound, or any other Int, any more than a nil NSString is equal to an empty string. A nil Int is equal to nil and nothing else.

```
aView.addSubview( optionalView )
```

value of optional type 'NSView?' not unwrapped

Just as you can’t use a UInt in Swift where an Int is required, you also can’t use an optional type where a non-optional type is required. That’s actually a useful feature—it means you have to put at least a little thought into what your code should do when the value is missing. You have a few options for handling optionals in situations like this.

```
aView.addSubview( optionalView! )
```

The simplest is the unwrapping operator, indicated by exclamation point, which forces the value out of an optional. The catch is, if the optional really *is* nil, the unwrapping operator will crash your app. Obviously, an operator that crashes is often not the solution you’re looking for.

```swift
let image = NSImage(named: "PlayIcon")!
```

But if you have an optional statement that “can’t fail”, like if you’re loading a resource from your app bundle, the unwrapping operator is a quick way to convert that optional value into a non-optional.

```
aView.addSubview( optionalView ?? defaultView )
```

Sometimes you'll want to fall back to some default value if the optional doesn't exist. You can do that with the defaulting operator, which is spelled with two question marks. The value on its left is an optional; the value on its right is a non-optional that will be used if the optional is nil.

```
if let existingView = optionalView {
    aView.addSubview( existingView )
}
```

More often, you'll want to test if a value exists and then do something with it if it does. The special "if let" construct does this. If the optional contains a value, the "let" declares a constant which is set to that value. If the optional contains nil, the entire branch is skipped, just like an ordinary "if". You can use "else" blocks with it, too.

```
optionalView?.removeFromSuperview()
```

Finally, if you want the classic Objective-C behavior of ignoring method calls on nil objects, put a question mark before the dot in the method call. The method will then return nil if the optional it’s called on is nil.

```
// Before you write:
var myString: String? = nil

// Consider using:
var myString: String = ""
```

But before you decide to make a value optional, you should consider if that’s really what you mean. Are you sure you don’t want zero, or an empty string, or an empty array or dictionary or whatever? Using nil where you don’t really want it can make your code more complicated than it needs to be. That’s another benefit of optionals—they make you think a little more about whether nil is what you really mean.

```
- (instancetype)initWithURL:(NSURL*)URL {
    if((self = [super init])) {
        NSDictionary * plist = [NSDictionary
            dictionaryWithContentsOfURL:URL];
        if(!plist) {
            return nil;
        }
        …
    }
    return self;
}
```

One more thing, which was added in Swift 1.1: You may be familiar with the Objective-C pattern where, if you can’t initialize an object, you should return nil.

```
init?(URL: NSURL) {
    super.init()
    let plist = NSDictionary(contentsOfURL: URL)
    if plist == nil {
        return nil
    }
    …
}
```

In Swift, you normally can’t do that. But if you mark an initializer with a question mark, as shown here, you become able to return nil from it. An initializer that can return nil is called a “failable initializer”. I’ll be showing you a few examples of them during the rest of this talk.

Using these features, Swift ensures you use nils only where you actually intend to—avoiding often subtle bugs.

III

No Type Changes

Objective-C also let you play fast and loose with types in other ways that Swift no longer allows.

```
@interface SunsetView: NSView

@property CAGradientLayer * layer;

@end
```

For example, Objective-C lets you redeclare properties to give them narrower types. Here, for instance, we know this view’s layer is always a CAGradientLayer, and we’d like to access it more easily, so we override the property to give it a more specific type.

```swift
class SunsetView: NSView {
    override var layer: CAGradientLayer? {
        get {
            return super.layer as CAGradientLayer?
        }
        set (newValue) {
            super.layer = newValue
        }
    }
```

Cannot override mutable property ‘layer’ of type ‘CALayer?’ with covariant type ‘CAGradientLayer?’

In Swift, though, this is forbidden. If you override a mutable property, you can’t change its type.

But wait a minute, that’s just stupid. I know that my view always uses that layer type. Why can’t I express it to Swift?

```
class SunsetView: NSView {
    override var layer: CAGradientLayer? {
        get {
            return super.layer as CAGradientLayer?
        }
        set (newValue) {
            super.layer = newValue
        }
    }
```

Cannot override mutable property ‘layer’ of type ‘CALayer?’ with covariant type ‘CAGradientLayer?’

Here’s the reason. Imagine that you could write this class and override the property to change its type.

```
let sunsetView = SunsetView(frame: self.bounds)
let aView = sunsetView as NSView
aView.layer = CALayer()

assert(sunsetView.layer is CAGradientLayer,
    "This can’t fail, right? Right?")
```

You then go off and create a SunsetView.
Then you can cast it to an NSView. That’s fine too, it’s a superclass.
Then you set the layer property to a CALayer, which is just fine for an NSView…
and you’ve violated the type guarantee on SunsetView’s layer property. That’s not so good.

Liskov Substitution Principle

"Let *q(x)* be a property provable about objects *x* of type *T*.

"Then *q(y)* should be provable for objects *y* of type *S*, where *S* is a subtype of *T*."

This is captured by a concept from type theory called the Liskov substitution principle. It has a formal definition, but that doesn't really matter very much for the working programmer.

```
class MyClass {
    func doIt(param: NSResponder) -> NSResponder
}

class MySubclass: MyClass {
    override func doIt(param: NSObject) -> NSView
}
```

What it comes down to is that, when you override,

```
class
                    NSResponder  -> NSResponder
{}

class
                           NSObject  -> NSView
{}
```

you can specify broader types for your inputs or narrower types for your outputs,

```
class
                    NSResponder  -> NSResponder
}

class
                          NSView  -> NSObject
}
```

Method does not override any method from its superclass
(this is considered an overload instead, and doesn't take the "override" keyword)

but not the other way around. This is actually a lot like a common principle of Internet engineering: "Be conservative in what you send, be liberal in what you accept." Objective-C doesn't really enforce Liskov substitutability, because in its heart of hearts, it believes everything is an *id*. But Swift is not so lenient.

```
class SunsetView: NSView {
    override var layer: CAGradientLayer? {
        get {
            return super.layer as CAGradientLayer?
        }
        set (newValue) {
            super.layer = newValue
        }
    }
```

Cannot override mutable property ‘layer’ of type ‘CALayer?’ with covariant type ‘CAGradientLayer?’

Since a mutable property’s type is an input to the setter and an output from the getter, you can’t make it narrower *or* wider without violating Liskov substitutability. Fortunately, there’s a very simple workaround:

```
class SunsetView: NSView {
    var gradientLayer: CAGradientLayer? {
        get {
            return super.layer as CAGradientLayer?
        }
        set (newValue) {
            super.layer = newValue
        }
    }
```

Don’t override the existing property; just create a new property beside it. That’s a bit clearer anyway.

IV

Safe Initialization

Swift includes a lot of rules to ensure that objects are created and destroyed correctly—but these rules can cause big problems.

```
class Person {
    var name: String

    init() { /* do nothing */ }
}
```


Property 'self.name' not initialized

Swift requires you to initialize every instance variable.

```
class Person: NSObject {
    var name: String

    override init() {
        super.init()
        self.name = "Jack"
    }
}
```

Property 'self.name' not initialized at super.init call

Moreover, it requires you to initialize them before the call to the superclass’s init().

```
class Person: NSManagedObject {
    override init(entity: NSEntityDescription,
      insertIntoManagedObjectContext:
      NSManagedObjectContext?) {
        // some initialization
    }
}
```


super.init isn't called before returning from initializer

It also requires you to actually call a superclass init().

```
class PersonDocument: NSDocument {
    init(type: String, error: NSErrorPointer) {
        super.init(type: type, error: error)
    }
}
```

must call a designated initializer of the superclass 'NSDocument'

That had better be a designated initializer, by the way, not a convenience initializer.

```swift
class Person {
    var name: String
    init?(URL: NSURL) {
        if let plist = NSDictionary(contentsOfURL: URL) {
            name = plist["name"] as String
        }
        else {
            return nil
        }
```

all stored properties of a class instance must be initialized before returning nil from an initializer

And finally, if you’re going to make a failable initializer, you actually have to fully initialize the object before you return nil to indicate that initialization failed.

Are you serious? Is Swift really this pedantic? Yes—and with good reason.

bit.ly/DrSwiftlove

Write this address down; it’s where you can get your goodie bag.

One of my Objective-C projects includes a class I call NPKUndoableObject; I use it to make model objects support undo. I’m actually giving you all a copy of this class because I’m such a nice guy.

```objc
@interface NPKUndoableObject: NSObject

// Override these
@property (readonly) NSUndoManager * undoManager;
+ (NSSet*)keyPathsForUndoRegistration;

// You must call these or bad things will happen
- (instancetype)init;
- (void)dealloc;

@end
```

Anyway, the basic design of this is really simple: you inherit from it and override two methods. One returns the undo manager that should be used for this object. The other returns a set of key paths whose contents should be undoable.

```
- (             init {


          [[
            [self addObserver:self forKeyPath:keyPath
                options:NSKeyValueObservingOptionPrior
                context:KVO];
        }
    }

}
```

The way this works is that the init method observes each of the key paths with KVO,

```
- (     dealloc {
    [
      [[
        [self removeObserver:self forKeyPath:keyPath
            context:KVO];
    }
}
```

And then the dealloc method removes those same observations.

```objc
@implementation Person

- (instancetype)initWithURL:(NSURL*)URL {
    NSDictionary * plist = [NSDictionary
        dictionaryWithContentsOfURL:URL];
    if(!plist) {
        return nil;
    }
    if((self = [super init])) {
        _name = plist[@"name"];
```

The problem is, what happens if you write a subclass that returns before it calls NPKUndoableObject’s init? Well, init is never called, but dealloc is, so it tries to remove observers that were never installed,

with predictable results.

You can’t really beat safe initialization, and it’s actually fairly helpful. But I do know a few tips that can help you work with it.

```
class Person {
    var name: String
    var administrator: Bool = false
    var birthdate: NSDate?

    init(name: String) {
        self.name = name
    }
}
```

There are two situations in which you don’t need to explicitly initialize a property in an initializer. One is if you set a default value when you declare the property. The other is if the property is optional—then the default value is nil. Here, only one of the three properties needs to be explicitly initialized because the others have default values.

```
init(coder: NSCoder?) {
    // Lots of complicated initialization
    super.init(coder: coder)
}

init(frame: NSRect?) {
    // Identical complicated initialization
    super.init(frame: frame)
}
```

One annoying aspect of safe initialization is that you can’t call other methods before calling super.init, in case they access properties that haven’t been initialized. This can make initializers very repetitive.

```
init(coder: NSCoder?) {
    (name, age) = commonInit()
    super.init(coder: coder)
}

init(frame: NSRect?) {
    (name, age) = commonInit()
    super.init(frame: frame)
}
```

There are a few ways to work around this, like using functions

```
init(coder: NSCoder?) {
    (name, age) = Person.commonInit()
    super.init(coder: coder)
}

init(frame: NSRect?) {
    (name, age) = Person.commonInit()
    super.init(frame: frame)
}
```

or class methods that return the properties’ initial values

```
init(coder: NSCoder?, orFrame: CGRect?) {
    // Lots of complicated initialization
    if let coder = coder {
        super.init(coder: coder)
    }
    else {
        super.init(frame: frame!)
    }
}
```

but my favorite is to make one Franken-initializer that encompasses both.

A few tricks like this and you won’t have nearly as much trouble with initializers.

v

Required Initializers

Finally, I’d like to discuss a Swift feature so painful that it actually caused flamewars when it was introduced: required initializers.

```
protocol Typeable {
    init?(text: String)
}
class IDNumber: Typeable { … }
class SocialSecurityNumber: IDNumber {
    init() { … }
    required init?(text: String) {
       super.init(text: text)
    }
}
```

One of the more troublesome safe initialization rules is that initializers in protocols *must* be explicitly implemented in all conforming classes, including subclasses. That doesn’t sound too terrible…

```
class MyView: NSView {
    init() { … }
    …
}
```

‘required’ initializer ‘init(coder:)’ must be provided by subclass of ‘NSView’

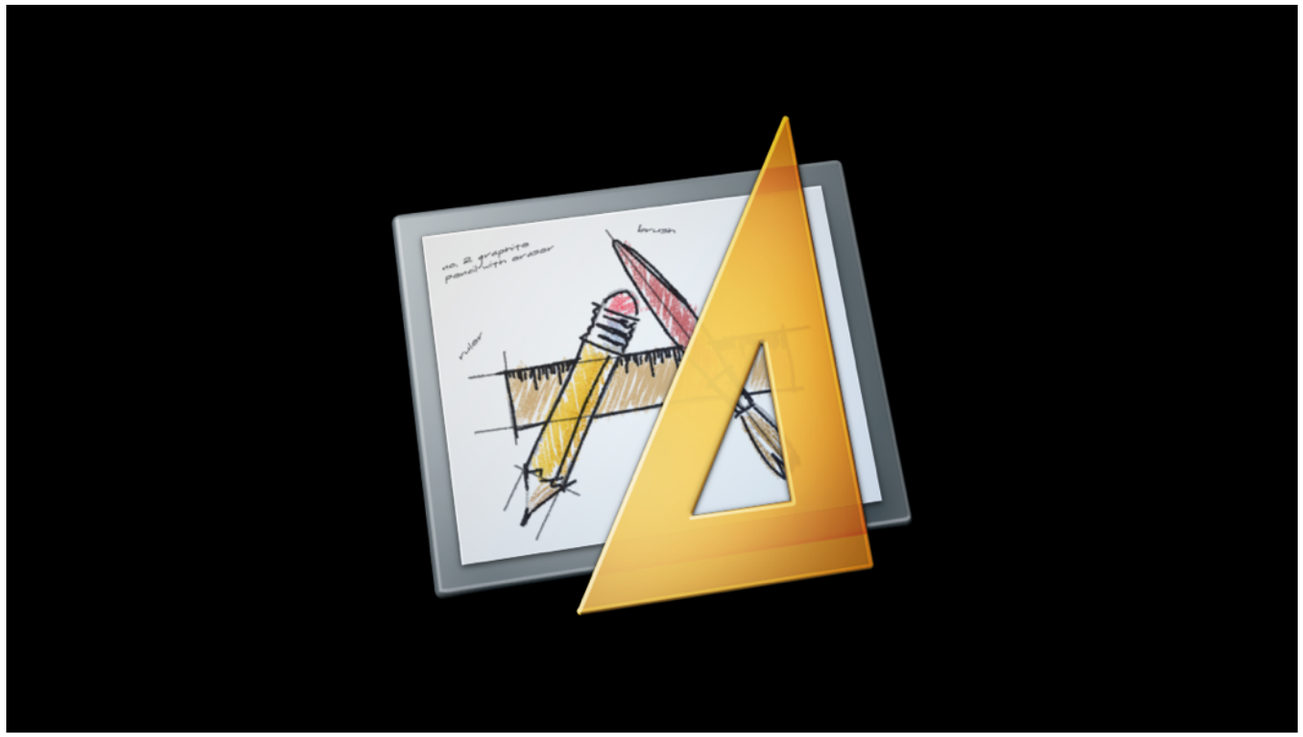
Until you remember that NSView and UIView both conform to NSCoding. That means that, if you implement any other initializers in your custom views, you must also implement init(coder:).

Well, maybe that wasn’t such a good idea after all.
(pause)
What could they be thinking? Well…

I have a shameful admission to make.

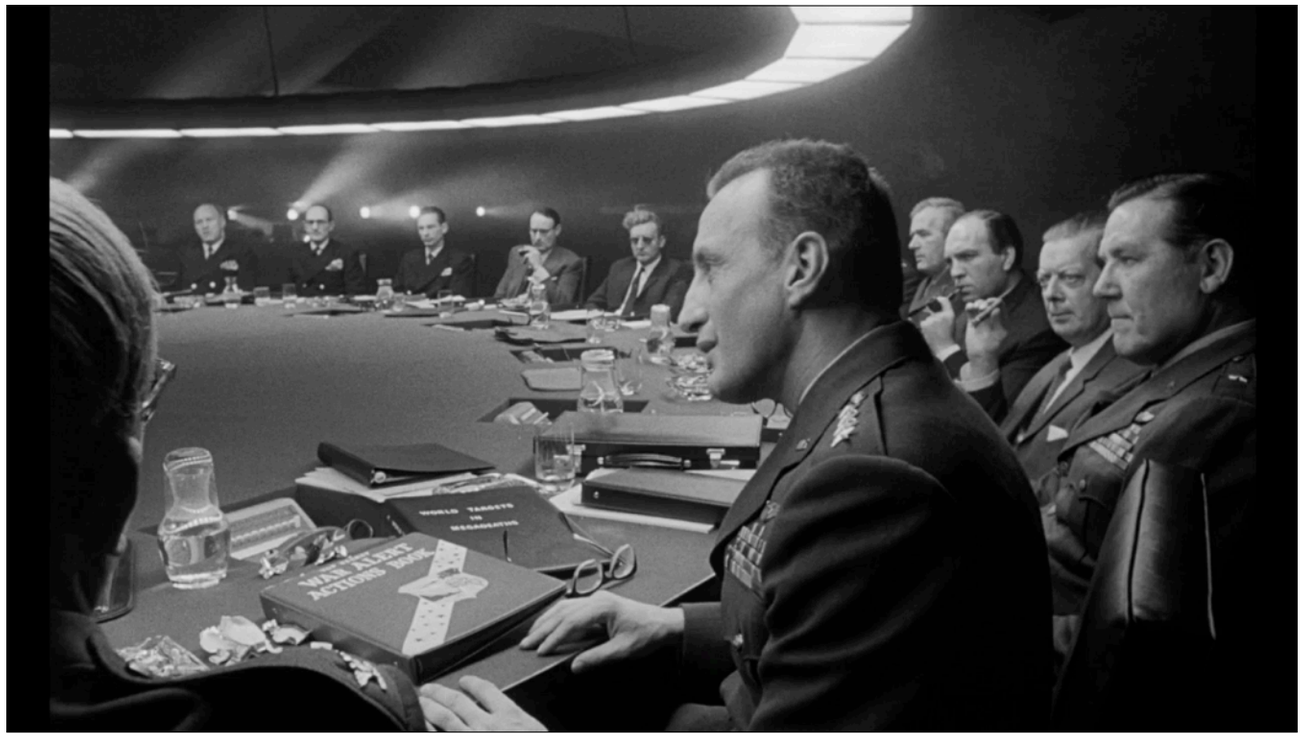
```
@implementation MyTextField

- (id)initWithFrame:(CGRect)frame {
    if((self = [super initWithFrame:frame])) {
       self.textColor = [NSColor blueColor];
    }
    return self;
}
```

You see, oftentimes, I sit down to write a view. And I override initWithFrame to do some initialization.



And then I go over into Interface Builder and drag out an instance of my new view…

…and my initialization didn’t happen, because nibs initialize views with initWithCoder.



This probably happens to me once every two weeks—I wish I was exaggerating—and I feel like a total moron every time. Anything Swift can do to keep me from making stupid mistakes like that is very welcome.

```
class PageStackView: NSView {
    // Instead of:
    var numberOfPages: Int
    override init(frame: NSRect) {
        numberOfPages = 1
        super.init(frame: frame)
    }

    // Just do:
    var numberOfPages: Int = 1
}
```

Now, one way to avoid this problem is to avoid writing initializers at all. Like I mentioned before, properties with default values don’t need to be initialized. If those are all the properties you have, then you may be able to avoid writing any other initializers, which means you can inherit your superclass’s init(coder:).

```
required init(coder: NSCoder) {
    …
}
```

If you can’t do that, the best solution to this problem is usually to just implement init(coder:) and stop making a fuss.

```
required init(coder: NSCoder) {
    fatalError(
        "init(coder:) has not been implemented")
}
```

But if you really don’t want to support nib loading for a particular view, just write a dummy implementation like this. Or better yet, let the Xcode Fix-It for this error write it for you.

Required initializers can be a bit of a pain, but they actually patch up a source of subtle bugs in Objective-C. Embrace the irritation.

No Implicit Conversions

Swift Optionals

No Type Changes

Safe Initialization

Required Initializers

In this talk, I’ve highlighted five safety features in Swift.

But there are dozens more. If you want to escape Swift's safety features, you pretty much have to go back to Objective-C. But I really suggest you learn to live within the rules, because they will save you from bugs.

Writing: a little slower

Debugging: way faster

Less boilerplate, more meaning

In my experience with Swift—which is still only part time—

It does take a little longer to write some new code.

But it takes *way* less time to debug it. The type system catches dozens of little things at compile time that you would've had to chase down in the debugger before.

And Swift lets you write at a higher level. A lot of things that are a half-dozen lines in Objective-C are just one or two in Swift. If you've been around long enough to remember the introduction of fast enumeration, it's like that, but better.

It turns out that a compiler error is much easier to fix than a runtime bug. So I hope you can, like me, learn to stop worrying and love the error.

So, uh, thank you.

Oh, and one more thing: I'll be leading a Swift lab tomorrow. No slides and no speech for this one—bring your own questions and experiences and code, and we'll all do show-and-tell for about 90 minutes. I think it'll be at 4:30, but double-check your schedules. I hope to see you there!