Background: We expect many changes in Objective-C APIs between Swift 3 and Swift 4, but our primary goal is to allow an app to upgrade to Swift 4 without having to update its dependencies. This necessarily means that a Swift 4 app needs to be able to use Swift 3 APIs, where "use" includes:

- call (a method, property accessor, etc)
- subclass (an open class)
- override (a method, property, etc)
- conform to (a protocol)
- satisfy (protocol requirements)
- extend (a class or protocol)

The situation gets even more complicated when

- the Swift 3 API overrides an imported ObjC member or satisfies an ObjC protocol requirement
- the Swift 3 library is updated to Swift 4
- we start considering Swift 3 uses of Swift 4 APIs as well

So let's start with the simplest possible changes, and work up to some of the more complicated ones. These are changes we expect Swift (and Objective-C) library authors to want to make in Swift 4 without breaking source compatibility. (Authors of Swift libraries also have another alternative: treat their Swift 4 update as a major and breaking release, in the Semantic Versioning sense, just as Swift 4 itself will be a major and breaking release.)

This document is written in the form of a guide for app authors and framework authors migrating to Swift 4, but its primary purpose is really to discover any compiler work we need to do to support this.

Note that this only applies to public and open APIs. Anything internal or private should not affect source compatibility.

Renames

A Swift 3 API wants to change its name in Swift 4

This isn't usually recommended at all, but there is one good reason: you're matching an Objective-C or standard library API whose name changed.

- If the API is a class, struct, enum, or protocol: rename it, then add a typealias for the old name, deprecated in Swift 4.
- If the API is a typealias: rename it, then add a typealias for the old name, deprecated in Swift 4.
- If the API is a top-level property or function: rename it, then add a forwarding function or computed property with the old name, deprecated in Swift 4.
- If the API is a generic parameter: rename it, then add a member typealias for the old name, deprecated in Swift 4.
- If the API is an enum case with no payload: rename it, then add a static computed property that returns the new name, deprecated in Swift 4.
- If the API is a struct or enum member (method, property, initializer, or subscript): rename it, then add a forwarding member with the old name, deprecated in Swift 4.
- If the API is a convenience initializer for a class: rename it, then add a forwarding initializer with the old name, deprecated in Swift 4.
- If the API is a non-convenience initializer for a non-open class: rename it, then add a forwarding convenience initializer with the old name, deprecated in Swift 4.
- If the API is a non-open class member: rename it, then add a forwarding member with the old name, deprecated in Swift 4.
- If the API is an open class member: **make a new member with the new name that invokes the old one,** and deprecate the old one. This is *backwards* from the usual process but ensures that existing overrides

outside your module continue to work.

All of these changes should result in existing Swift 3 and Swift 4 clients continuing to work even when they fetch a new version of your library...barring a few awkward edge cases like a reference to a function by its base name becoming ambiguous when only argument names are changed.

It is recommended to mark any post-rename declarations as "introduced in Swift 4", to minimize the chances of affecting overload resolution in Swift 3 code. It's considered acceptable for migrating from Swift 3 to Swift 4 to require minor fixes.

Unfortunately, there are a few declarations that cannot be renamed without breaking clients.

- If the API is an associated type: renaming these will break any adopters. If you only care about people using the protocol, not making new types that conform to it, you can add a typealias for the old name in a protocol extension and deprecate it in Swift 4. FIXME: Will this work in constraints?
- If the API is an enum case with a payload: renaming these will break pattern-matching. If you don't care about clients pattern-matching this case, you can add a static function with the same name and signature as the old case, and deprecate it in Swift 4.
- If the API is a non-convenience initializer for an open class: you're pretty much out of luck. Since overriding all non-convenience initializers triggers special behavior, adding a new one would cause issues.

An Objective-C API has changed its name in Swift 4

- If the API is a top-level function or variable, it should be available under its old name as well, though with a deprecation warning---even if it was "renamed" into being a member. There should be no problems here.
- If the API is a type (typedef, struct, enum, union, class, or protocol), it should be available under its old name as well, as a deprecated typealias. There should be no problems here either.
- If the API is a generic parameter, the old name isn't available. FIXME: Uh oh?
- If the API is an enumerator, the old name should be available as a deprecated alias (technically a static var, but we have special logic for pattern-matching that).
- If the API is a struct or union field, the old name should be available as a deprecated alias.
- If the API is a convenience initializer, the old name should be available as a deprecated convenience initializer as well. FIXME: inheritance?
- If the API is a designated initializer...FIXME
- If the API is a method or property, the old name should be available as well, though deprecated. Overrides of either name should work (more information below).

Overrides

If a Swift 3 library has a class 'Sub' that overrides an Objective-C method using the method's original name, what should Swift 4 client code do when given an instance of 'Sub'? There are two options:

- Refer to the method by its Swift 3 name.
- Refer to the method by its Swift 4 name.

Obviously the second one is preferable, as it's more forward-looking. For plain renames, it'd be possible to make either choice work with some amount of compiler hacking. However, when we get to changes in types (below), the choice becomes more clear.

Optionality Changes

A Swift 3 API wants to make something optional in Swift 4

Like renames, this usually isn't recommended, but once again the change may be to match a change in an Objective-C API.

- If it's a non-inout parameter for a top-level function, go ahead.
- If it's a non-inout parameter for a struct or enum method, go ahead.
- If it's a non-inout parameter for a non-open class method, go ahead.
- If it's an inout parameter for a top-level function, a struct or enum method, or a non-open class method, you can't change the function without breaking source. However, you can probably add an overload.
- If it's an index for a non-open subscript, you can treat it like a function parameter. See above.

There are a few cases where IUO will let you preserve source compatibility. Note, however, that this is **very dangerous** since existing clients won't expect a nil value in these locations:

- If it's a non-open property or subscript element, using IUO will not break source compatibility (but may break clients at runtime if you actually produce nil).
- If it's a result type for a top-level function, struct or enum method, or non-open class method, using IUO will not break source compatibility (but may break clients at runtime if you actually produce nil).

And there are a few cases where there's just no good answer.

- If it's an open property on a class, an argument or result type for an open method on a class, or an element or index type for an open subscript on a class, you're going to break any overrides. You're pretty much stuck here.
- Similarly, if it's within the type for a protocol requirement, you're going to break conforming structs/enums/classes. You're stuck here too.
- If it's an enum case payload, this will break existing pattern-matching. Note that *constructing* the enum should still work even with the payload being made optional, but there are very few enums constructed by clients but never used by them.

A Swift 3 API wants to make something non-optional in Swift 4

This change is pretty much not possible to do without breaking clients. Nearly every use of an optional value could end up in an `if let` or optional chain, which is not valid to perform on non-optional values. There's really only one safe place to do this: parameters where `nil` was never a valid value anyway:

- If it's a parameter for a top-level function, a struct or enum method, or a non-open class method, introduce an overload that does not take nil and mark the original function/method obsoleted in Swift 4.
- If it's an index for a non-open subscript, you can treat it like a function parameter. See above.

Any other changes are likely to break clients, even more so if overrides are a possibility.

An Objective-C API made something optional in Swift 4

- If it's a parameter for a top-level function, a struct or enum method, or a class method you call but don't override, there should be no problem.
- If it's an index for a subscript you call but don't override, there should be no problem.

Unfortunately, the remaining changes are clearly unsafe, but may still happen because of APIs that were mis-annotated.

- If it's a property or a subscript element, this change **breaks source compatibility.**
- If it's a result type for a top-level function, struct or enum method, or class method, this change **breaks source compatibility.**
- If it's an argument for a method or subscript that's overridden, these overrides will be broken, as will their clients, so this definitely **breaks source compatibility.**
- Similarly, if it's within the type for a protocol requirement, this affects conforming classes, which means it **breaks source compatibility.**

If you own an Objective-C framework that needs to make these changes and you want to preserve Swift 3 compatibility, you can use the "API notes" mechanism to lie and mark the API as non-nullable when compiling in Swift 3 mode. (FIXME: link to a document about API notes.) If you're a client of such a framework, you may be able to minimize the impact of some of these changes by following the steps for something being made *non-*optional.

Overrides

(FIXME: to be written)

An Objective-C API made something non-optional in Swift 4

Like with the Swift 3 API case, this is pretty much **guaranteed to break clients** in non-parameter cases. The workaround for a client is to add `as Optional` to their use of the value.

```
// FIXME: example
```

When a parameter has been made non-optional, there is no way to continue passing `nil` to it save by using a helper method or function written in Objective-C. Again, this is only a client-side workaround, and the client should probably just not pass `nil` to begin with.

Overrides

It is generally allowed to override a method that takes a non-optional value with one that takes an optional value, so that particular change is no problem. In the case where it is a property type or a method result type being made non-optional, the change is inherently source-breaking, and the client must be updated to change its return type and to make sure it does not return nil.

Nested optionality

FIXME: Most specifically, Objective-C pointers to pointers.

Value Types, `NS_STRING_ENUM`

In Objective-C

FIXME: Hm.

Adopting new types in Swift

FIXME: Ugh.

Throwing/Non-throwing

In Objective-C

FIXME: Can we expose both versions? Maybe?

In Swift

FIXME: No.

Changes to designated initializers

Changing a designated initializer to a convenience initializer is almost guaranteed to break subclasses, since it's entirely possible that a subclass is overriding this initializer, or using `super.init` to chain to it.

In Objective-C, such code may have always been incorrect if the initializer really was a convenience initializer---that is, if it delegated to another initializer on `self`---which may result in double-initialization of properties of the subclass (possibly resulting in leaks). However, it is also possible that some subclasses *would* have worked correctly, and it is important for those to keep building. The best option is to update your headers, but reverse the change in Swift 3 mode using the "API Notes" mechanism.

Changing a convenience initializer to a designated initializer can also break subclasses that were relying on convenience initializers being inherited. In Swift, convenience initializers are only inherited if all designated initializers are overridden, or if *no* designated initializers are overridden and all new instance properties have initial values.

Once again, Objective-C clients should update their headers to match the implementations, but reverse the change in Swift 3 mode using the "API Notes" mechanism.

Class properties

FIXME: We could make sure overloading properties and methods actually works, and then we can present both.

Escaping closures

A Swift API wants to mark something as `@escaping`

This is pretty much always a **breaking change**. In addition to the trivial case of forcing clients to write `self.` on property accesses within the closure, it may also ruin their assumptions about their object graph.

If you're doing this just to deal with an API that takes an escaping closure, use the `withoutActuallyEscaping` helper instead, and make sure that the API you're using won't hold on to the closure by the time your function exits.

An API wants to remove `@escaping`

(or, for Objective-C, "an API adds the `noescape` attribute")

For an API that is not overridden, this is a source-compatible change---any existing calls will still work. Keep in mind that the reverse is not allowed, though, and plan for the future.

Overrides

FIXME: Seems problematic. Do we allow overriding an escaping parameter with a non-escaping one? That would help a lot. Otherwise we may need to make these insert an implicit "withoutActuallyEscaping".

Error code enums

An Objective-C enum may be newly flagged as an error enum (e.g. by the "NSErrorDomain" annotation in API notes). This change is *nearly* source-compatible anyway; although the name now refers to a struct wrapping an NSError instance instead of a raw error code, all of the enum constants are still available as members, and the domain string will still be exposed. This means that nearly all code working directly with NSError and error codes in Swift 3 will continue to work in Swift 4; it's only if someone is explicitly using the type itself that there may be problems.

(Which can happen, but is usually rare. Nevertheless, this Objective-C-side change *is* source-breaking, and should be reversed when compiling under Swift 3 compatibility mode using API notes.)

Other Things?

FIXME: Are there any other major categories of changes that don't manifest as type changes? At the very least there are a few things supported by API notes (or at least considered) that don't fall into those buckets.