orphan:

Objective-C Interoperability

This document tracks the differences between the Swift and Objective-C ABIs and class models, and what it would take to merge the two as much as possible. The format of each section lays out the differences between Swift and Objective-C, then describes what needs to happen for a user to mix the two seamlessly.

> **Warning**
>
> This document was used in planning Swift 1.0; it has not been kept up to date and does not describe the current or planned behavior of Swift.

Contents

Terminology used in this document:

- `id`-compatible: something that can be assigned to an `id` variable and sent messages using `objc_msgSend`. In practice, this probably means implementing the `NSObject` protocol, since most of Cocoa doesn't check whether something implements `NSObject` before sending a message like `-class`.
- Objective-C isa: something that identifies the class of an Objective-C object, used by `objc_msgSend`. To say a Swift object has an Objective-C isa does *not* mean that a fully-formed Objective-C runtime class structure is generated for the Swift class; it just means that (1) the header of the Swift object "looks like" an Objective-C object, and (2) the parts of an Objective-C class used by the `objc_msgSend` "fast path" are the same.

Design

All Swift objects [1] will be `id`-compatible and will have an Objective-C isa, on the assumption that you want to be able to put them in an array, set them as represented objects, etc. [2]

Swift classes that inherit from NSObject (directly or indirectly [3]) behave exactly like Objective-C classes from the perspective of Objective-C source. All methods marked as "API" in Swift will have dual entry points exposed by default. Methods not marked as "API" will not be exposed to Objective-C at all. Instances of these classes can be used like any other Objective-C objects.

Subclassing a "Swift NSObject class" in Objective-C requires a bit of extra work: generating Swift vtables. We haven't decided how to do this:

- Clang could be taught about Swift class layout.
- The Clang driver could call out to the Swift compiler to do this. Somehow.
- The runtime could fill in the vtable from the Objective-C isa list at class load time. (This could be necessary anyway to support dynamic subclassing... which we may or may not do.)

Swift classes that do not inherit from NSObject are not visible from Objective-C. Their instances can be manipulated as `id`, or via whatever protocols they may implement.

```
class AppController : NSApplicationDelegate {
  func applicationDidFinishLaunching(notification : NSNotification) {
    // do stuff
  }
}

// Use 'id <NSApplicationDelegate>' in Objective-C.
```

Like "Swift NSObject classes", though, "pure" Swift classes will still have an isa, and any methods declared in an Objective-C protocol will be emitted with dual entry points.

[1] Really, "All Swift objects on OS X and iOS". Presumably a Swift compiler on another system wouldn't bother to emit the Objective-C isa info.

[2] Dave is working out an object and class layout scheme that will minimize the performance cost of emitting both the Objective-C isa and a Swift vtable. It is entirely possible that from the Swift perspective, the Objective-C isa is just an opaque "vtable slice" that is fixed at offset 0.

[3] ...or any other Objective-C class, including alternate roots like NSProxy. Most likely this will be implemented with an inherited attribute `[objc]` on the class, which would even allow Swift to create Objective-C root classes.

Use Cases

Unfinished and undetailed.

Simple Application Writer

I want to write my new iOS application in Swift, using all the Objective-C frameworks that come with iOS.

Guidelines:

Everything should Just Workâ„¢. There should be no need to subclass NSObject anywhere in your program, unless you are specifically specializing a class in the Cocoa Touch frameworks.

Intermediate Application Writer

I want to write my new application in Objective-C, but there's a really nice Swift framework I want to use.

Guidelines:

- Not all Swift methods in the framework may be available in Objective-C. You can work around this by adding *extensions* to the Swift framework classes to expose a more Objective-C-friendly interface. You will need to mark these new methods as "API" in order to make them visible to Objective-C.
- "Pure" Swift classes will not be visible to Objective-C at all. You will have to write a wrapper class (or wrapper functions) in Swift if you want to use the features of these classes directly. However, you can still treat them like any other objects in your program (store them in `id` variables, Objective-C collections, etc).

Transitioning Application Writer

I have an existing Objective-C application, and I want to convert it piece-by-piece to Swift.

Guidelines:

- Swift is different from Objective-C in that methods in Swift classes are not automatically usable from everywhere. If your Swift class inherits from NSObject, marking your methods as "API" will allow them to be called from Objective-C code. A Swift class that does not inherit from NSObject will only respond to messages included in its adopted protocols. [4]
- Once you have finished transitioning to Swift, go through your classes and remove the "API" marker from any methods that do not need to be accessed from Objective-C. Remove NSObject as a superclass from any classes that do not need to be accessed from Objective-C. Both of these allow the compiler to be more aggressive in optimizing your program, potentially making it both smaller and faster.

[4] If you explicitly want to expose a Swift method to Objective-C, but it is not part of an existing protocol, you can mark the method as "API" and include the `[objc]` attribute:

```
// Note: This syntax is not final!
func [API, objc] accessibilityDescription {
```

```
        return "\(self.givenName) \(self.familyName)"
    }
```

New Framework Writer

I want to write a framework that can be used by anyone.

Requirements:

- Can call (at least some) Swift methods from Objective-C.

Intermediate Framework Writer

I have an existing Objective-C framework that I want to move to Swift.

Requirements:

- Can subclass Objective-C classes in Swift.
- Can call (at least some) Swift methods from Objective-C.

Decisions:

- Should I expose Swift entry points as API?
- If so, should they be essentially the same as the Objective-C entry points, or should I have a very different interface that's more suited for Swift (and easily could be "better")?

End User

- Things should be fast.
- Things should not take a ton of memory.

Nice to Have (uncategorized)

- Can write a Swift extension for an Objective-C class.
- Can write a Swift extension for an Objective-C class that adopts an Objective-C protocol.
- Can write a Swift extension for an Objective-C class that exposes arbitrary new methods in Objective-C.

Tradeoffs

This section discusses models for various runtime data structures, and the tradeoffs for making Swift's models different from Objective-C.

Messaging Model

Everything is `id`-compatible:

- Less to think about, maximum compatibility.
- Every Swift object must have an Objective-C isa.

Non-NSObjects are messageable but not `id`-compatible:

- Cannot assign Swift objects to `id` variables.
- Cannot put arbitrary Swift objects in NSArrays.
- Potentially confusing: "I can message it but I can't put it in an `id`??"
- Clang must be taught how to message Swift objects and manage their retain counts.
- On the plus side, then non-NSObjects can use Swift calling conventions.
- Requires framework authors to make an arbitrary decision that may not be ABI-future-proof.

Non-NSObjects are opaque:

- Can be passed around, but not manipulated.
- ...but Clang probably *still* has to be taught how to manage the retain count of an opaque Swift object, and doing so in the same way as dispatch_queue_t and friends may be dangerous (see <os/object.h> -- it's pretending they're NSObjects, which they are)
- Requires framework authors to make an arbitrary decision that may not be ABI-future-proof.

Method Model

This only affects methods marked as "API" in some way. Assume for now that all methods use types shared by both Objective-C and Swift, and that calls within the module can still be optimized away. Therefore, this discussion only applies to frameworks, and specifically the use of Swift methods from outside of the module in which they are defined.

Every method marked as API can *only* be accessed via Objective-C entry points:

- Less to think about, maximum compatibility.
- Penalizes future Swift clients (and potentially Objective-C clients?).

Every method marked as API can be accessed both from Objective-C and Swift:

- Maximum potential performance.
- Increases binary size and linking time.
- If this is a framework converted to Swift, clients that link against the Swift entry points are no longer backwards-compatible. And it's hard to know what you did wrong here.
- Overriding the method in Objective-C requires teaching Clang to emit a Swift vtable for the subclass.

Methods marked as "ObjC API" can only be accessed via Objective-C entry points; methods marked as "Swift API" can only be accessed via Swift entry points:

- Changing the API mode breaks binary compatibility.
- Obviously this attribute is inherited -- overriding an Objective-C method should produce a new Objective-C entry point. What is the default for new methods, though? Always Swift? Always Objective-C? Based on the class model (see below)? Specified manually?

Methods marked as "ObjC API" can be accessed both from Objective-C and Swift; methods marked as "Swift API" can only be accessed via Swift entry points:

- More potential performance for the shared API.
- Increases binary size and linking time.
- Overriding the method in Objective-C requires teaching Clang to emit a Swift vtable for the subclass.
- Same default behavior problem as above -- it becomes a decision.

Class Model

All Swift classes are layout-compatible with Objective-C classes:

- Necessary for `id`-compatibility.
- Increases binary size.

Only Swift classes marked as "ObjC" (or descending from an Objective-C class) are layout-compatible with Objective-C classes; other classes are not:

- Requires framework authors to make an arbitrary decision.
- Changing the API mode *may* break binary compatibility (consider a Swift subclass that is not generating Objective-C class information).

Subclassing Model

Requirement: can subclass Objective-C objects from Swift.

All Swift classes can be subclassed from Objective-C:

- Potentially increases binary size.
- Requires teaching Clang to emit Swift vtables.

Only Swift classes marked as "ObjC" (or descending from an Objective-C class) are subclassable in Objective-C:

- Probably *still* requires teaching Clang to emit Swift vtables.
- Requires framework authors to make an arbitrary decision that may not be ABI-future-proof.

Method Overriding Model

Requirement: Swift classes can override any Objective-C methods.

Methods marked as "overridable API" only have Objective-C entry points:

- Less to think about, maximum compatibility.
- Penalizes future Swift clients (and potentially Objective-C clients?).

Methods marked as "overridable API" have both Objective-C and Swift entry points:

- Requires teaching Clang to emit Swift vtables.
- Increases binary size and link time.

Methods marked as "overridable API" have only Swift entry points:

- Requires teaching Clang to emit Swift vtables.
- Later exposing this method to Objective-C in a subclass may be awkward?

Attributes for Objective-C Support

`@objc`

- When applied to classes, directs the compiler to emit Objective-C metadata for this class. Additionally, if no superclass is specified, the superclass is implicitly `NSObject` rather than the default `swift.Object`. Note that Objective-C class names must be unique across the entire program, not just within a single namespace or module. [5]
- When applied to methods, directs the compiler to emit an Objective-C entry point and entry in the Objective-C

method list for this method.
- When applied to properties, directs the compiler to emit Objective-C methods *-foo* and `-setFoo:`, which wrap the getter and setter for the property.
- When applied to protocols, directs the compiler to emit Objective-C metadata for this protocol. Objective-C protocols may contain optional methods. Method definitions for an Objective-C protocol conformance are themselves implicitly `@objc`.

This attribute is inherited (in all contexts).

`@nonobjc`

- When applied to methods, properties, subscripts or constructors, override the implicit inheritance of `@objc`.
- Only valid if the declaration was implicitly `@objc` as a result of the class or one of the class's superclasses being `@obj` -- not permitted on protocol conformances.
- It is permitted to override a `@nonobjc` method with a method marked as `@objc`; overriding an `@objc` (or implicitly `@objc`) method with a `@nonobjc` method is not allowed.
- It is an error to combine `@nonobjc` with `dynamic`, `@IBOutlet` or `@NSManaged`.

This attribute is inherited.

`@IBOutlet`

Can only be applied to properties. This marks the property as being exposed as an outlet in Interface Builder. **In most cases,** outlets should be weak properties.

The simplest implementation of this is to have `@IBOutlet` *cause an Objective-C getter and setter to be emitted, but this is* not *part of* `@IBOutlet`'s *contract.*

This attribute is inherited.

`@IBAction`

Can only be applied to methods, which must have a signature matching the requirements for target/action methods on the current platform. This marks the method as being a potential action in Interface Builder.

The simplest implementation of this is to have `@IBAction` *imply* `@objc`, *and this may be the* only *viable implementation given how the responder chain works. For example, a window's delegate is part of the responder chain, even though it does not subclass* `NSResponder` *and may not be an Objective-C class at all. Still, this is* not *part of* `@IBAction`'s *contract.*

This attribute is inherited.

[5] I'm not really sure what to do about uniquing Objective-C class names. Maybe eventually [objc] will take an optional argument specifying the Objective-C-equivalent name.

Level 1: Message-passing

Assuming an object is known to be a Swift object or an Objective-C object at compile-time, what does it take to send a message from one to the other?

ARC

By default, objects are passed to and returned from Objective-C methods as +0 (i.e. non-owned objects). The caller does not have to do anything to release returned objects, though if they wish to retain them they may be able to steal them out of the top autorelease pool. (In practice, the caller *does* retain the arguments for the duration of the method anyway, unless it can be proven that nothing interferes with the lifetime of the object between the load and the call.)

Objective-C methods from certain method families do return +1 objects, as do methods explicitly annotated with the `ns_returns_retained` attribute.

All Swift class objects (i.e. as opposed to structs) are returned as +1 (i.e. owned objects). The caller is responsible for releasing them.

Swift methods that are exposed as Objective-C methods will have a wrapper function (thunk) that is responsible for retaining all (object) arguments and autoreleasing the return value.

*Swift methods will **not** be exposed as* `ns_returns_retained` because they should behave like Objective-C methods when called through an* `id`.

Arguments

Objective-C currently requires that the first argument be `self` and the second be `_cmd`. The explicit arguments to a method come after `_cmd`.

Swift only requires that the first argument be `self`. The explicit arguments come after `self`.

The thunk mentioned above can shift all arguments over...which doesn't really cost anything extra since we already have to retain all the arguments.

Output Parameters

Because Objective-C does not have tuples, returning multiple values is accomplished through the use of pointer-to-object-pointer parameters, such as `NSError **`. Additionally, objects returned through these parameters are conventionally autoreleased, though ARC allows this to be specified explicitly.

Swift has tuples and does not have pointers, so the natural way to return multiple values is to return a tuple. The retain-count issue is different here: with ARC, the tuple owns the objects in it, and the caller owns the tuple.

Swift currently also has `[inout]` arguments. Whether or not these will be exposed to users and/or used for Objective-C out parameters is still undecided.

This issue has not been resolved, but it only affects certain API.

Messaging `nil`

In Objective-C, the result of messaging `nil` is defined to be a zero-filled value of the return type. For methods that return an object, the return value is also `nil`. Methods that return non-POD C++ objects attempt to default-construct the object if the receiver is `nil`.

In Swift, messaging `nil` is undefined, and hoped to be defined away by the type system through liberal use of some `Optional` type.

- I've seen other languages explicitly request the Objective-C behavior using `foo.?bar()`, though that's not the prettiest syntax in the world. -Jordan

As long as the implementation of `Optional` is layout-compatible with an object pointer, and an absent `Optional` is represented with a null pointer, this will Just Workâ„¢.

Overloading

In Objective-C, methods cannot be overloaded.

In Swift, methods can have the exact same name but take arguments of different types.

Note that in Swift, all parameters after the first are part of the method name, unless using the "selector syntax" for defining methods:

```
// 1. foo:baz:
func foo(Int bar, Int baz);

// 2. foo:qux:
func foo(Int bar, Int qux);

// 3. foo:qux: (same as above)
func foo(Int bar) qux(Int quux);

// 4. foo:baz: (but different type!)
func foo(Int bar, UnicodeScalar baz);

a.foo(1, 2)      // ambiguous in Swift (#1 or #2?)
a.foo(1, baz=2)  // calls #1
a.foo(1, qux=2)  // calls #2/3 (the same method)
a.foo(1, 'C')    // calls #4, not ambiguous in Swift!

[a foo:1 baz:2]; // ambiguous in Objective-C (#1 or #4?)
[a foo:1 qux:2]; // calls #2/3 (the same method)
```

The Swift compiler should not let both #1 and #4 be exported to Objective-C. It should already warn about the ambiguity between #1 and #2 without using named parameters.

Level 2: Messaging `id`

If a Swift object can be referenced with `id`*, how do you send messages to it?*

Note: the answer might be "Swift objects can't generally be referenced with `id`".

`isa` Pointers

The first word of every Objective-C object is a pointer to its class.

We might want to use a more compact representation for Swift objects...

...but we can't; see below.

Method Lookup

> Objective-C performs method lookup by searching a sequence of maps for a given key, called a *selector*. Selectors are pointer-sized and uniqued across an entire process, so dynamically-loaded methods with the same name as an existing method will have an identical selector. Each map in the sequence refers to the set of methods added by a category (or the original class). If the lookup fails, the search is repeated for the superclass.
>
> Swift performs method lookup by vtable. In order to make these vtables non-fragile, the offset into a vtable for a given message is stored as a global variable. Rather than chaining searches through different message lists to account for inheritance and categories, the container for each method is known at compile-time. So the final lookup for a given method looks something like this:
>
> ```
> vtable[SUBCLASS_OFFSET + METHOD_OFFSET]
> ```

Swift class objects will have `isa` pointers, and those `isa` pointers will have an Objective-C method list at the very least, and probably a method cache as well. The methods in this list will refer to the Objective-C-compatible wrappers around Swift methods described above.

The other words in the `isa` structure may not be used in the same way as they are in Objective-C; only `objc_msgSend` has to avoid special-casing Swift objects. Most of the other runtime functions can probably do a check to see if they are dealing with a Swift class, and if so fail nicely.

Level 3a: Adopting Objective-C Protocols in Swift

- Bare minimum for implementing an AppKit/UIKit app in Swift.
- Essentially the same as emitting any other Objective-C methods, plus making `-conformsToProtocol:` and `+conformsToProtocol:` work properly.

Level 3b: Adopting Swift Protocols in Objective-C

- Requires generating both Swift and Objective-C entry points from Clang.
- Requires generating Swift protocol vtables.

Note: including protocol implementations is essentially the same as implicitly adding an extension (section 5a).

Level 4a: Subclassing Objective-C Classes in Swift

To be written.

- Basically necessary for implementing an AppKit/UIKit app in Swift.
- Requires generating Objective-C-compatible method lists.
- When a new method is marked as API, does it automatically get the Objective-C calling conventions by default? (See "Tradeoffs" section.)

Level 4b: Subclassing Swift Classes in Objective-C

To be written.

- May require generating Swift vtables.

  Alternative: if a method is exposed for overriding, it only gets an Objective-C entry point. (Downsides: performance, other platforms will hate us.)

  Alternative: only Swift classes with an Objective-C class in their hierarchy can be subclassed in Objective-C. Any overridden methods must be exposed as Objective-C already. (Downsides: framework authors could forget to inherit from NSObject, Swift code is penalized ahead of time.)

  Alternative: only Swift classes with an Objective-C class in their hierarchy are *visible* in Objective-C. All other Swift objects are opaque. (Downsides: same as above.)

Level 5a: Adding Extensions to Objective-C Classes in Swift

To be written.

- May require generating Objective-C-compatible method lists.
- Less clear what the *default* calling convention should be for new methods.

Level 5b: Adding Categories to Swift Classes in Objective-C

To be written.

- Does not actually *require* generating Swift vtables. But we could if we wanted to expose Swift entry points for these methods as well.

- Does require an Objective-C-compatible `isa` to attach the new method list to.

Level 6: Dynamic Subclassing

To be written, but probably not an issue...it's mostly the same as statically subclassing, right?

Level 7: Method Swizzling

I'm okay with just saying "no" to this one.