orphan:

Integrating Swift Constructors with Objective-C

> **Warning**
>
> This proposal was rejected, though it helped in the design of the final Swift 1 initialization model.

Objective-C's "designated initializers pattern seems at first to create a great deal of complication. However, designated initializers are simply the only sane response to Objective-C's initialization rules, which are the root cause of the complication.

This proposal suggests an approach to initialization that avoids the problems inherent in Objective-C while still *allowing* Objective-C programmers to pursue the designated initializer pattern on subclasses of Swift classes.

The Root of the Problem

The root problem with Objective-C's initialization rules is that the `init` methods of a superclass automatically become public members of its subclasses. This leads to a soundness problem:

```
@interface SuperClass
- initSuperClass
@end

@interface Subclass : Superclass
- (void)subclassMethod
@end

@implementation Subclass : Superclass
char* name;                       // never initialized

- (void)print { printf(name); } // oops
@end

mySubclassInstance = [[Subclass alloc] initSuperClass]
```

Because there is no way to hide a superclass' `init` method from clients, ensuring that subclass instances are properly initialized requires overriding *every* superclass initializer in *every* subclass:

```
@implementation Subclass : Superclass
char* name;
- initSuperClass {
  [super initSuperClass];       // Don't forget the superclass
  name = "Tino";
}
- (void)print { printf(name); } // OK
@end
```

Following this rule is obviously tedious and error-prone for users. Initialization is crucial to correctness, because it is where invariants are established. It therefore should be no more complex than everything else to reason about.

Also, it means adding an `init` method in a base class can be API-breaking.

Furthermore, as John McCall pointed out recently, it forces inappropriate interfaces on subclasses. For example, every subclass of `NSObject` has a parameter-less `init` function, whether or not there's an appropriate way to construct instances of that subclass without parameters. As a result, class designers may be forced to expose weaker invariants than the ones they could otherwise establish.

Exceptions to the Rule

I exaggerated a little in the previous section: because overriding *every* superclass initializer in *every* subclass is so tedious, the Objective-C community has identified some situations where you don't really need to override every `init` method:

1. When you know the default zero-initialization of a class' instance variables is good enough, you don't need to override any `init` methods from your superclass.
2. If a given superclass' `init` method always calls another `init` method, you don't need to override the first `init` method because your instance variables will be initialized by your override of the second `init` method. In this case, the first (outer) `init` method is called a **secondary initializer**. Any `init` method that's not secondary is called a **designated initializer**.

How to Think About This

At this point I'll make a few assertions that I hope will be self-evident, given the foregoing context:

1. If the programmer follows all the rules correctly, one initializer is as good as another: every `init` method, whether designated or secondary, fully initializes all the instance variables. This is true for all clients of the class, including subclassers.
2. Distinguishing designated from secondary initializers does nothing to provide soundness. It's *merely* a technique for limiting

the tedious `init` method overrides required of users.

3. Swift users would not be well-served by a construction model that exposes superclass `init` methods to clients of subclasses by default.

Proposal

I suggest we define Swift initialization to be as simple and easily-understood as possible, and avoid "interesting" interactions with the more complicated Objective-C initialization process. If we do this, we can treat Objective-C base classes as "sealed and safe" for the purpose of initialization, and help programmers reason effectively about initialization and their class invariants.

Here are the proposed rules:

- `init` methods of base classes defined in Objective-C are not, by default, part of the public interface of a subclass defined in Swift.
- `init` methods of base classes defined in Swift are not, by default, part of the public interface of a subclass defined in Objective-C.
- `self.init(...)` calls in Swift never dispatch virtually. We have a safe model for "virtual initialization:" `init` methods can call overridable methods after all instance variables and superclasses are initialized. Allowing *virtual* constructor delegation would undermine that safety.
- As a convenience, when a subclass' instance variables all have initializers, it should be possible to explicitly expose superclass init methods in a Swift subclass without writing out complete forwarding functions. For example:

  ```
  @inherit init(x:y:z) // one possible syntax
  ```

  > **Note**
  >
  > Allowing `@inherit init(*)` is a terrible idea
  >
  > It allows superclasses to break their subclasses by adding `init` methods.

Summary

By eliminating by-default `init`method inheritance and disabling virtual dispatch in constructor delegation, we give class designers full control over the state of their constructed instances. By preserving virtual dispatch for non-`self`, non-`super` calls to `init` methods, we allow Objective-C programmers to keep using the patterns that depend on virtual dispatch, including designated initializers and `initWithCoder` methods.