orphan:

> **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.

Mutable Namespace-Scope Variable Declarations

A namespace-scope variable (i.e. a variable not inside a function) is allowed to have an initializer, and that initializer is allowed to have side effects. Thus, we have to decide how and when the initializer runs.

WLOG, lets assume that all namespace-scope variables are mutable (and thus that immutable variables are just an optimization of the common case). Given that they can have mutable state, they cannot be "global" (in the C sense) because then they would be visible across multiple actors. Instead, the only logical semantic is for them to be actor-local data ("thread local" in the C sense) of some sort.

Given that there can be many of these variables in an address space, and very few of them may be dynamically used by any particular actor, it doesn't make sense to allocate space for all of the variables and run all of the initializers for the variables at actor-startup-time. Instead, swift should handle these as "actor associated data" (stored in a hashtable that the actor has a pointer to) and should be lazily initialized (in the absence of 'top level code', see below).

This means that if you write code like this (optionally we could require an attribute to make it clear that the value is actor local):

```
func foo(_ a : int) -> int { print(a) return 0 }

var x = foo(1)
var y = foo(2)
```

That the print statements will execute the first time that x or y is used by any particular actor.

Top Level Code

One goal of swift is to provide a very "progressive disclosure" model of writing code and learning how to write code. Therefore, it is desirable that someone be able to start out with:

```
print("hello world\n")
```

as their first application. This requires that we support "top level code", which is code outside any function or other declaration. The counter-example of this is java, which requires someone to look at "class foo / public static void main String[]...." all of which is non-essential to the problem of writing a simple app.

Top level code is useful for a number of other things: many scripts written by unix hackers (in perl, bourne shell, ruby, etc) are really just simple command line apps that may have a few helper functions and some code that runs. While not essential, it is a great secondary goal to make these sorts of simple apps easy to write in Swift as well.

Top-Level code and lazily evaluated variable initializers don't mix well, nor does top level code and multiple actors. As such, the logical semantics are:

1. Source files are partitioned into two cases: "has TLC" and "has no TLC".
2. All variables defined in "has no TLC" files are allocated and initialized lazily.
3. Source files that have TLC are each initialized in a deterministic order: The dependence graph of domains is respected (lower level domains are initialized before dependent ones), and the source files within a domain are initialized in some deterministic order (perhaps according to their filename or something, TBD).
4. Within a source file with TLC, the TLC is run top down in deterministic order whenever the file's initializer is run. This initializer executes in the context of the "first" actor, which is created on behalf of the program by the runtime library.
5. If/when some other actor refers to a variable in a file with TLC, it is allocated and initialized lazily just like globals in "has no TLC" files.

On "Not Having Headers"

One intentional design decision in swift is to not have header files, even for public API. This is a design point like Java, but unlike C or Objective-C. Having header files for public API is nice for a couple of reasons:

1. You *force* people to think about what they are making public, and the act of having to put it into a header makes them think about its fragility and duration over time.
2. Headers are a very convenient place to get an overview of what an API is and does. In Java you get all the implementation details of a class mixed in with its public API, which makes it very difficult to understand "at a glance" what a class does.
3. Headers are very useful documentation for Objective-C because we ship the headers but not the implementation of system classes. This allows "jump to definition" to go to the declaration of an API in the header, which is conveniently co-located with headerdoc.

On the other hand, headers have a number of disadvantages including:

1. It is plain code duplication, with all the negative effects of it. It slows down development, can get out of synch, makes changing API more difficult, etc.
2. If the prototype and implementation get out of synch, it is caught by the compiler, but this isn't true for API comments.
3. Swift natively won't "need" headers, so we'd have to specifically add this capability, making the language more complicated.
4. The implementation of a framework may not be in swift. If you're talking to a C or C++ framework, showing a C or C++ header when "jumping to definition" is not particularly helpful. We'd prefer to show you the synthesized API that swift code should be using.
5. In Swift, the implementation of some datatype can be split across different files. Forcing all their declarations to be next to each other lexically is an arbitrary restriction.

To address the disadvantages of not having headers, we think that we should:

1. Standardize on a syntax for doc comments, and bake it into the language. Mistakes using it should be diagnosed by the compiler. It should be a warning for public API to not have comments.
2. There needs to be an API that dumps out the public interface for a compiled module/domain in swift syntax, slicing on a declaration. When used on a type, for example, this would show the type definition and the declaration of all of the methods on it.
3. The API dumper should always dump in swift syntax, even when run on a Clang C/C++/ObjC module. It should make it very clear what the API maps to in swift syntax, so it is obvious how to use it.
4. Not having headers forces us to have really great tools support/integration.