Object Initialization

Warning

This document is incomplete and not up-to-date; it currently describes the initialization model from Swift 1.0.

Contents

Introduction

Object initialization is the process by which a new object is allocated, its stored properties initialized, and any additional setup tasks are performed, including allowing its superclass's to perform their own initialization. *Object teardown* is the reverse process, performing teardown tasks, destroying stored properties, and eventually deallocating the object.

Initializers

An initializer is responsible for the initialization of an object. Initializers are introduced with the `init` keyword. For example:

```
class A {
  var i: Int
  var s: String

  init int(i: Int) string(s: String) {
    self.i = i
    self.s = s
    completeInit()
  }

  func completeInit() { /* ... */ }
}
```

Here, the class `A` has an initializer that accepts an `Int` and a `String`, and uses them to initialize its two stored properties, then calls another method to perform other initialization tasks. The initializer can be invoked by constructing a new `A` object:

```
var a = A(int: 17, string: "Seventeen")
```

The allocation of the new `A` object is implicit in the construction syntax, and cannot be separated from the call to the initializer.

Within an initializer, all of the stored properties must be initialized (via an assignment) before `self` can be used in any way. For example, the following would produce a compiler error:

```
init int(i: Int) string(s: String) {
  completeInit() // error: variable 'self.i' used before being initialized
  self.i = i
  self.s = s
}
```

A stored property with an initial value declared within the class is considered to be initialized at the beginning of the initializer. For example, the following is a valid initializer:

```
class A2 {
  var i: Int = 17
  var s: String = "Seventeen"

  init int(i: Int) string(s: String) {
    // okay: i and s are both initialized in the class
    completeInit()
  }

  func completeInit() { /* ... */ }
}
```

After all stored properties have been initialized, one is free to use `self` in any manner.

Designated Initializers

There are two kinds of initializers in Swift: designated initializers and convenience initializers. A *designated initializer* is responsible for the primary initialization of an object, including the initialization of any stored properties, *chaining* to one of its superclass's designated initializers via a `super.init` call (if there is a superclass), and performing any other initialization tasks, in that order. For example, consider a subclass `B` of `A`:

```
class B : A {
  var d: Double

  init int(i: Int) string(s: String) {
    self.d = Double(i)            // initialize stored properties
    super.init(int: i, string: s) // chain to superclass
    completeInitForB()            // perform other tasks
  }

  func completeInitForB() { /* ... */ }
}
```

Consider the following construction of an object of type `B`:

```
var b = B(int: 17, string: "Seventeen")
```

Initialization proceeds in several steps:

1. An object of type `B` is allocated by the runtime.
2. `B`'s initializer initializes the stored property `d` to `17.0`.
3. `B`'s initializer chains to `A`'s initializer.
4. `A`'s initializer initializes the stored properties `i` and `s`'.
5. `A`'s initializer calls `completeInit()`, then returns.
6. `B`'s initializer calls `completeInitForB()`, then returns.

A class generally has a small number of designated initializers, which act as funnel points through which the object will be initialized. All of the designated initializers for a class must be written within the class definition itself, rather than in an extension, because the complete set of designated initializers is part of the interface contract with subclasses of a class.

Note

Swift differs from many other languages in that it requires one to initialize stored properties *before* chaining to the superclass initializer. This is part of Swift's memory safety guarantee, and is discussed further in the section on Three-Phase Initialization.

The other, non-designated initializers of a class are called convenience initializers, which tend to provide additional initialization capabilities that are often more convenient for common tasks.

Convenience Initializers

A *convenience initializer* is an initializer that provides an alternative interface to the designated initializers of a class. A convenience initializer is denoted by the return type `Self` in the definition. Unlike designated initializers, convenience initializers can be defined either in the class definition itself or within an extension of the class. For example:

```
extension A {
  init() -> Self {
    self.init(int: 17, string: "Seventeen")
  }
}
```

A convenience initializer cannot initialize the stored properties of the class directly, nor can it invoke a superclass initializer via `super.init`. Rather, it must *dispatch* to another initializer using `self.init`, which is then responsible for initializing the object. A convenience initializer is not permitted to access `self` (or anything that depends on `self`, such as one of its properties) prior to the `self.init` call, although it may freely access `self` after `self.init`.

Convenience initializers and designated initializers can both be used to construct objects, using the same syntax. For example, the `A` initializer above can be used to build a new `A` object without any arguments:

```
var a2 = A() // uses convenience initializer
```

Initializer Inheritance

One of the primary benefits of convenience initializers is that they can be inherited by subclasses. Initializer inheritance eliminates the need to repeat common initialization code---such as initial values of stored properties not easily written in the class itself, or common registration tasks that occur during initialization---while using the same initialization syntax. For example, this allows a `B` object to be constructed with no arguments by using the inherited convenience initializer defined in the previous section:

```
var b2 = B()
```

Initialization proceeds as follows:

1. A `B` object is allocated by the runtime.
2. `A`'s convenience initializer `init()` is invoked.
3. `A`'s convenience initializer dispatches to `init int:string:` via the `self.init` call. This call dynamically resolves to `B`'s designated initializer.
4. `B`'s designated initializer initializes the stored property `d` to `17.0`.
5. `B`'s designated initializer chains to `A`'s designated initializer.
6. `A`'s designated initializer initializes the stored properties `i` and `s`'.
7. `A`'s designated initializer calls `completeInit()`, then returns.
8. `B`'s designated initializer calls `completeInitForB()`, then returns.
9. `A`'s convenience initializer returns.

Convenience initializers are only inherited under certain circumstances. Specifically, for a given subclass to inherit the convenience initializers of its superclass, the subclass must override each of the designated initializers of its superclass. For example `B` provides the initializer `init int:string:`, which overrides `A`'s designated initializer `init int:string:` because the initializer name and parameters are the same. If we had some other subclass `OtherB` of `A` that did not provide such an override, it would not inherit `A`'s convenience initializers:

```
class OtherB : A {
  var d: Double

  init int(i: Int) string(s: String) double(d: Double) {
    self.d = d                    // initialize stored properties
    super.init(int: i, string: s) // chain to superclass
  }
}

var ob = OtherB()   // error: A's convenience initializer init() not inherited
```

Note that a subclass may have different designated initializers from its superclass. This can occur in a number of ways. For example, the subclass might override one of its superclass's designated initializers with a convenience initializer:

```
class YetAnotherB : A {
  var d: Double

  init int(i: Int) string(s: String) -> Self {
    self.init(int: i, string: s, double: Double(i)) // dispatch
  }

  init int(i: Int) string(s: String) double(d: Double) {
    self.d = d                    // initialize stored properties
    super.init(int: i, string: s) // chain to superclass
  }
}

var yab = YetAnotherB()   // okay: YetAnotherB overrides all of A's designated initializers
```

Note

The requirement that a subclass override all of the designated initializers of its superclass to enable initializer inheritance is crucial to Swift's memory safety model. See Initializer Inheritance Model for more information.

In other cases, it's possible that the convenience initializers of the superclass simply can't be made to work, because the subclass initializers require additional information provided via a parameter that isn't present in the convenience initializers of the superclass:

```
class PickyB : A {
  var notEasy: NoEasyDefault

  init int(i: Int) string(s: String) notEasy(NoEasyDefault) {
    self.notEasy = notEasy
    super.init(int: i, string: s) // chain to superclass
  }
}
```

Here, `PickyB` has a stored property of a type `NoEasyDefault` that can't easily be given a default value: it has to be provided as a parameter to one of `PickyB`'s initializers. Therefore, `PickyB` takes over responsibility for its own initialization, and none of `A`'s convenience initializers will be inherited into `PickyB`.

Synthesized Initializers

When a particular class does not specify any designated initializers, the implementation will synthesize initializers for the class when all of the class's stored properties have initial values in the class. The form of the synthesized initializers depends on the superclass (if present).

When a superclass is present, the compiler synthesizes a new designated initializer in the subclass for each designated initializer of the superclass. For example, consider the following class `C`:

```
class C : B {
  var title: String = "Default Title"
}
```

The superclass `B` has a single designated initializer,:

```
init int(i: Int) string(s: String)
```

Therefore, the compiler synthesizes the following designated initializer in `C`, which chains to the corresponding designated initializer in the superclass:

```
init int(i: Int) string(s: String) {
  // title is already initialized in the class C
  super.init(int: i, string: s)
}
```

The result of this synthesis is that all designated initializers of the superclass are (automatically) overridden in the subclass, becoming designated initializers of the subclass as well. Therefore, any convenience initializers in the superclass are also inherited, allowing the subclass (`C`) to be constructed with the same initializers as the superclass (`B`):

```
var c1 = C(int: 17, string: "Seventeen")
var c2 = C()
```

When the class has no superclass, a default initializer (with no parameters) is implicitly defined:

```
class D {
  var title = "Default Title"

  /* implicitly defined */
  init() { }
}

var d = D() // uses implicitly-defined default initializer
```

Required Initializers

Objects are generally constructed with the construction syntax `T(...)` used in all of the examples above, where `T` is the name of the type. However, it is occasionally useful to construct an object for which the actual type is not known until runtime. For example, one might have a `View` class that expects to be initialized with a specific set of coordinates:

```
struct Rect {
  var origin: (Int, Int)
  var dimensions: (Int, Int)
}

class View {
  init frame(Rect) { /* initialize view */ }
}
```

The actual initialization of a subclass of `View` would then be performed at runtime, with the actual subclass being determined via some external file that describes the user interface. The actual instantiation of the object would use a type value:

```
func createView(_ viewClass: View.Type, frame: Rect) -> View {
  return viewClass(frame: frame) // error: 'init frame:' is not 'required'
}
```

The code above is invalid because there is no guarantee that a given subclass of `View` will have an initializer `init frame:`, because the subclass might have taken over its own initialization (as with `PickyB`, above). To require that all subclasses provide a particular initializer, use the `required` attribute as follows:

```
class View {
  @required init frame(Rect) {
    /* initialize view */
  }
}

func createView(_ viewClass: View.Type, frame: Rect) -> View {
  return viewClass(frame: frame) // okay
}
```

The `required` attribute allows the initializer to be used to construct an object of a dynamically-determined subclass, as in the `createView` method. It places the (transitive) requirement on all subclasses of `View` to provide an initializer `init frame:`. For example, the following `Button` subclass would produce an error:

```
class Button : View {
  // error: 'Button' does not provide required initializer 'init frame:'.
}
```

The fix is to implement the required initializer in `Button`:

```
class Button : View {
  @required init frame(Rect) { // okay: satisfies requirement
    super.init(frame: frame)
  }
}
```

Initializers in Protocols

Initializers may be declared within a protocol. For example:

```
protocol DefaultInitializable {
  init()
}
```

A class can satisfy this requirement by providing a required initializer. For example, only the first of the two following classes conforms to its protocol:

Note

Initializers in protocols have not yet been implemented. Stay tuned.

```
class DefInit : DefaultInitializable {
  @required init() { }
}

class AlmostDefInit : DefaultInitializable {
  init() { } // error: initializer used for protocol conformance must be 'required'
}
```

The `required` requirement ensures that all subclasses of the class declaring conformance to the protocol will also have the initializer, so they too will conform to the protocol. This allows one to construct objects given type values of protocol type:

```
func createAnyDefInit(_ typeVal: DefaultInitializable.Type) -> DefaultInitializable {
  return typeVal()
}
```

De-initializers

While initializers are responsible for setting up an object's state, *de-initializers* are responsible for tearing down that state. Most classes don't require a de-initializer, because Swift automatically releases all stored properties and calls to the superclass's de-initializer. However, if your class has allocated a resource that is not an object (say, a Unix file descriptor) or has registered itself during initialization, one can write a de-initializer using `deinit`:

```
class FileHandle {
  var fd: Int32

  init withFileDescriptor(fd: Int32) {
    self.fd = fd
  }

  deinit {
    close(fd)
  }
}
```

The statements within a de-initializer (here, the call to `close`) execute first, then the superclass's de-initializer is called. Finally, stored properties are released and the object is deallocated.

Methods Returning `Self`

A class method can have the special return type `Self`, which refers to the dynamic type of `self`. Such a method guarantees that it will return an object with the same dynamic type as `self`. One of the primary uses of the `Self` return type is for factory methods:

```
extension View {
  class func createView(_ frame: Rect) -> Self {
    return self(frame: frame)
  }
}
```

Within the body of this class method, the implicit parameter `self` is a value with type `View.Type`, i.e., it's a type value for the class `View` or one of its subclasses. Therefore, the restrictions are the same as for any value of type `View.Type`: one can call other class methods and construct new objects using required initializers of the class, among other things. The result returned from such a method must be derived from the type of `Self`. For example, it cannot return a value of type `View`, because `self` might refer to some subclass of `View`.

Note

The return type `Self` fulfills the same role as Objective-C's `instancetype`, although Swift provides stronger type checking for these methods.

Instance methods can also return `Self`. This is typically used to allow chaining of method calls by returning `Self` from each method, as in the builder pattern:

```
class DialogBuilder {
  func setTitle(_ title: String) -> Self {
    // set the title
    return self;
  }

  func setBounds(_ frame: Rect) -> Self {
    // set the bounds
    return self;
  }
}

var builder = DialogBuilder()
                .setTitle("Hello, World!")
                .setBounds(Rect(0, 0, 640, 480))
```

Memory Safety

Swift aims to provide memory safety by default, and much of the design of Swift's object initialization scheme is in service of that goal. This section describes the rationale for the design based on the memory-safety goals of the language.

Three-Phase Initialization

The three-phase initialization model used by Swift's initializers ensures that all stored properties get initialized before any code can make use of `self`. This is important uses of `self`---say, calling a method on `self`---could end up referring to stored properties before they are initialized. Consider the following Objective-C code, where instance variables are initialized *after* the call to the superclass initializer:

```
@interface A : NSObject
- (instancetype)init;
- (void)finishInit;
@end

@implementation A
- (instancetype)init {
  self = [super init];
  if (self) {
    [self finishInit];
  }
  return self;
}
@end

@interface B : A
@end

@implementation B {
  NSString *ivar;
}

- (instancetype)init {
  self = [super init];
  if (self) {
    self->ivar = @"Default name";
  }
  return self;
}

- (void) finishInit {
  NSLog(@"ivar has the value %@\n", self->ivar);
}
@end
```

When initializing a `B` object, the `NSLog` statement will print:

```
ivar has the value (null)
```

because `-[B finishInit]` executes before `B` has had a chance to initialize its instance variables. Swift initializers avoid this issue by splitting each initializer into three phases:

1. Initialize stored properties. In this phase, the compiler verifies that `self` is not used except when writing to the stored properties of the current class (not its superclasses!). Additionally, this initialization directly writes to the storage of the stored properties, and does not call any setter or `willSet`/`didSet` method. In this phase, it is not possible to read any of the stored properties.

Notes

In Objective-C, `+alloc` zero-initializes all of the instance variables, which gives them predictable behavior before the init method gets to initialize them. Given that Objective-C is fairly resilient to `nil` objects, this default behavior eliminates (or hides) many such

2. Call to superclass initializer, if any. As with the first step, `self` cannot be accessed at all.

3. Perform any additional initialization tasks, which may call methods on `self`, access properties, and so on.

Note that, with this scheme, `self` cannot be used until the original class and all of its superclasses have initialized their stored properties, closing the memory safety hole.

initialization bugs. In Swift, however, the zero-initialized state is less likely to be valid, and the memory safety goals are stronger, so zero-initialization does not suffice.

Initializer Inheritance Model

FIXME: To be written

Objective-C Interoperability

Initializers and Init Methods

Designated and Convenience Initializers

Allocation and Deallocation

Dynamic Subclassing