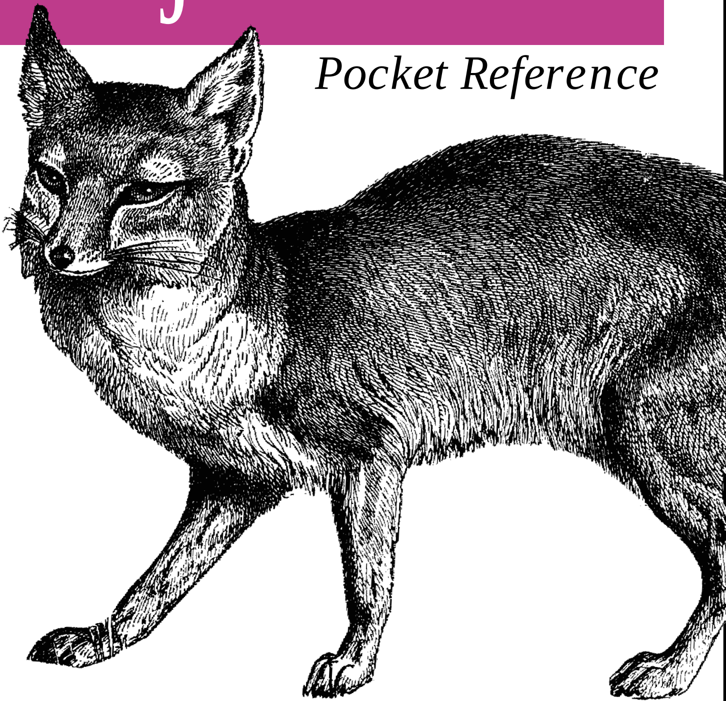
A Guide to Language Fundamentals

Objective-C

Pocket Reference



O'REILLY®

Andrew M. Duncan

Objective-C
Pocket Reference

Andrew M. Duncan

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Object Lifecycle

Your classes will have methods that distinguish them from other classes and make them useful, but all classes must implement methods that manage their lifecycle—allocation, initialization, copying, and deletion. In addition, you will use library classes that come supplied with these methods, which you need to use in a consistent way. This section describes the design patterns that Objective-C programmers use and that library classes support.

The root classes `Object` and `NSObject` provide the following methods for managing the lifecycles of objects:

```
+initialize
+alloc
+new
-init
-copy
-dealloc
```

In addition, `Object` also provides these methods:

```
-shallowCopy
-deepCopy
-deepen
-free
```

In addition to these methods, many classes will provide more methods for initializing newly allocated objects.

The "Root Classes" section describes how these methods behave for the root classes; this section gives you guidelines on how to actually use the methods in your programs.

In managing the lifecycle of an object, you are faced with two issues: how to call these methods and how to write them for your own classes. Each of the following sections will first discuss how to call the methods, and then how to write them.

Creating an Object

Objective-C separates object creation into two steps: allocating memory and initializing fields. Allocation returns a pointer to cleared memory where the object will be stored. Initializing an object means setting its fields to some values, either default or specified. These operations serve distinct purposes, are performed by different objects (allocation by a class, and initialization by an instance), and you write and call each of them explicitly.

Calling creation methods

To create an object, you first ask its class to allocate memory, and then you initialize the instance:

```
1 Circle* c = [Circle alloc];
2 c = [c init];
```

Line 1. In Objective-C you call a class method to return a pointer to memory for a new object. It is conventional to call this method `+alloc`. Both `Object` and `NSObject` supply an `+alloc` method. The object you get from `+alloc` will have all its fields set to zero.

Line 2. An *initializer* is an instance method that sets the fields of a newly allocated object. Every object responds to the `-init` message, which it inherits from the root class. The root class version does nothing, but leaves the object in its pristine just-allocated state. Descendants may override `-init` to provide a default initialization.

An initializer returns the initialized object, but that object may be different from the receiver of the initialization message. For example, an initializer may fail and return **`nil`**, or substitute a proxy or other special object for the one passed in. For this reason it is not safe to call an initializer as a void method:

```
[c init];  // Discards return value.
```

In this example, any return value is discarded. If `-init` returns an object different from the receiver, this code will lose that object and the receiver will not be correctly initialized. To avoid this problem, chain the calls to allocate and initialize your objects, or use the `new` method, which does this for you (but only for the bare-bones `-init` method). Both of the following lines allocate and initialize an instance of *`Circle`*:

```
Circle* c1 = [[Circle alloc] init];
Circle* c2 = [Circle new];  // Same effect.
```

Many classes will supply more initialization methods; it is conventional for their names to start with `init`. These methods may take additional parameters to guide the setting up of the receiver's state. For example:

```
Circle* c = [[Circle alloc] initWithRadius:3];
```

Writing creation methods

Your code should maintain the separation of allocation and initialization. You must also implement the chaining of initializers that ensures that objects are initialized first as instances of their root class, and successively as instances of more derived classes.

To coordinate these steps you are obliged to manage details of object creation that other languages automate. The advantage is that you can more easily understand the behavior of your program by direct inspection.

Your class should always inherit or provide a class method for returning a pointer to cleared memory where the object will be stored. You shouldn't override this method in subclasses. In addition to reserving memory, `+alloc` needs to set up the internal structure of an object before it is initialized. This makes writing a root class difficult, and your class should usually inherit its allocator from a root class provided by your development environment. Both `Object` and `NSObject` supply an `+alloc` method. If you need to write your own root class, look at *Object.m* to see how this is done.

When an object is initialized, it should become a valid, consistent instance of its class. For this to happen, all Objective-C classes need to cooperate to ensure several things:

- All the ancestors of a class must initialize an object before the class itself does.
- Ancestor initialization must proceed in order from the root class to the most-derived class.
- All ancestor initialization calls should be usable on a class's instance. (The difficulty here is guaranteeing that the class's own initialization will not be skipped.)

Objective-C programmers have adopted the following design patterns to ensure these conditions are always met:

- Your class may have several initialization methods; it is conventional to name those methods starting with `init`.
- The most specialized initializer (usually the one with the most parameters) is called the *designated initializer* and has a special role: all your class's other initializers should call it.
- Your class should override the parent class's designated initializer.
- Your designated initializer should call the parent class's designated initializer.

The rationale for these guidelines is presented in the next section in the context of a concrete example.

Sample code for initialization

The following code illustrates the design pattern you should follow to ensure correct initialization of your objects. In the example, you are writing the subclass *MyClass*. We assume the parent class follows the same rules we illustrate in the subclass.

```
@interface Parent : Object {
  int i;
}
```

```
 4   -(id)init;
 5   -(id)initWithI:(int)val;
 6 @end
 7
 8 @interface MyClass : Parent {
 9   int j;
10 }
11   -(id)initWithI:(int)iVal;
12   -(id)initWithI:(int)iVal andJ:(int)jVal;
13 @end
14
15 @implementation MyClass
16   -(id)initWithI:(int)iVal {
17     return [self initWithI:iVal andJ:42];
18   }
19   -(id)initWithI:(int)iVal andJ:(int)jVal {
20     if (self = [super initWithI:iVal]) {
21       j = jVal;
22     }
23     return self;
24   }
25 @end
```

Line 4. All initializers return **`id`**. This class has a simple `-init` method, taking no parameters. It calls (funnels to) `-initWithI:` passing in a default value.

Line 5. *`Parent`* provides another initializer, `initWithI:`, which lets you specify the value of the field *`i`*. This is the designated initializer.

Line 11. *`MyClass`* overrides (covers) its parent's designated initializer. Always do this in your classes.

NOTE

If you don't cover a parent's designated initializer, code such as:

```
MyClass* obj =
   [[MyClass alloc] initWithI:42];
```

will go straight to the parent class's initializer and leave *`j`* undefined.

You must cover all the parent class's initializers; if the parent class funnels all its initializers through the designated initializer (as we are assuming here), overriding it will cover them all. Full coverage ensures that your subclass instances will be substitutable for parent class instances.

Line 12. Provide specialized initializers for your class's specific new features. This method is the designated initializer for *`MyClass`*.

Line 17. All your class's other initializers should call (funnel to) your class's designated initializer. Funneling lets future subclasses cover all your initializers by just overriding the designated initializer. Pass in to your designated initializer some desired default value for parameters not specified in the simpler initializer.

Line 20. Your designated initializer should first call (chain to) the parent's designated initializer. Calling the parent initializer ensures that all the parent classes will get their chance to initialize the object.

NOTE

Calling the parent designated initializer avoids a circular call path: if instead you called `-init` here, its (presumed) call to `-initWithI:` would be dispatched to your new version, and from there back to this method.

You first assign the result of the chaining call to **`self`**, in case the call returns an object different from the receiver. Then test for **`nil`** (the **`if`** statement does this) in case the parent class failed to initialize the object.

Line 21. Your designated initializer performs class-specific work.

Line 23. If your initializer fails, it should return **`nil`**. The way this example is written, that happens automatically. If your initializer performs more complicated steps, you may have to ensure this explicitly.

Initializing classes

The runtime system will call a class's +initialize class method some time before that class or any of its descendant classes is used. Each class will receive the initialize message before any of its subclasses. If you have some set-up code for the class as a whole (apart from its instances) you should implement this method.

If your class implements +initialize but it has a subclass that does not, the call to initialize the subclass will be handled by your class—that is, your class will receive the message more than once. For this reason, you should guard against multiple calls in your method:

```
+(id)initialize {
  static BOOL done = NO;
  if (!done) {
    // Your initialization here.
    done = YES;
  }
  return self;
}
```

This example is for descendants of Object; the NSObject version of +initialize returns **void** instead of returning an **id**.

Copying an Object

When an object has only value types for fields (apart from the isa pointer), making a copy is a simple matter of duplicating all the fields in a new memory location. When some of the fields are pointers, there are two types of copy that you can make:

- A *shallow copy* duplicates the pointers by value, so the new object refers to the same objects as did the original one.

- A *deep copy* duplicates the objects pointed to, and continues this process, traversing all the pointers of duplicated objects.

The root classes provide a basic framework of copy methods, but your classes will have to override them to get proper deep copying.

Calling copy methods

The copy methods of `Object` all return a shallow copy of the receiver. For `Object` itself, the distinction between deep and shallow is meaningless, since the only pointer it has is the `isa` pointer, which is supposed to be shared between all instances. In descendant classes that properly override the methods, their behavior will be as follows:

`-(`**`id`**`)copy`
: Returns a deep copy of the receiver.

`-(`**`id`**`)shallowCopy`
: Returns a shallow copy of the receiver.

`-(`**`id`**`)deepen`
: Modifies the receiver, replacing all of its non-value fields with deep copies.

`-(`**`id`**`)deepCopy`
: Returns a deep copy of the receiver.

The `NSObject` class provides only the `-copy` method, and it simply calls the unimplemented method `-copyWithZone:`. You need to consult a class's documentation to know if it supports copying. The next section describes how to implement this method yourself to support copying in your classes.

When you get a copy of an object in Cocoa, it has already been retained for you and you will have to call `release` on it when you are done with it. The "Memory Management" section explains more about retaining and releasing objects.

Writing copy methods

To implement copying for subclasses of `Object`, you only need to override the `-deepen` method to recursively traverse the receiver's pointers and replace them with pointers to newly allocated objects identical to the originals.

Cocoa doesn't implement any copying methods for you; it just declares two copying protocols. These protocols don't distinguish between shallow and deep copies: it is up to your classes to decide (and document) what kind of copies they will return. The protocols do distinguish between mutable and immutable copies. A mutable object is one whose values can change; an immutable one must stay constant after it is created. The protocols are:

`NSCopying`
: Declares the method `-copyWithZone:`. Adopt this protocol and implement the method to return a copy of the receiver. You must at least adopt this protocol to support copying. If you also adopt `NSMutableCopying`, `-copyWithZone:` should return an immutable copy.

`NSMutableCopying`
: Declares the method `-mutableCopyWithZone:`. If your class distinguishes between mutable and immutable values, adopt this protocol and implement the method to return a mutable copy of the receiver.

Each method takes as a parameter a pointer to an `NSZone`, which isn't a class but an opaque type. Normally you will not look any deeper into the zone parameter, but pass it along to an inherited copy method like `-allocWithZone:` or a Cocoa runtime function like `NSZoneAlloc( )`. See the Cocoa documentation for information on how to use the runtime functions for allocating memory.

Since `NSObject` does not implement either protocol, you must adopt and implement the protocols if you want your objects to support copying. Your class will inherit from `NSObject`, whose methods `-copy` and `-mutableCopy` call the respective protocol methods (with **nil** parameters) so you can use those simpler methods to make copies of your class's instances.

If the distinction between mutable and immutable doesn't matter for your class (all instances are mutable or all are immutable), just adopt NSCopying. If instances may be one or the other kind, adopt both NSCopying and NSMutableCopying, using the first for returning immutable copies and the second for mutable ones.

If your class doesn't inherit any -copyWithZone: method, implement it using +allocWithZone: and an initialization method. For example:

```
-(id)copyWithZone:(NSZone*)zone {
  return [[self class] allocWithZone:zone] init];
}
```

In this example, you evaluate [**self** class] to get the receiver of the allocation message. Don't use the name of your class here. If you do, descendants won't be able to inherit this method because the kind of object it creates will be hard-wired.

If your class inherits a -copyWithZone: method, you might not need to change it, for example if your class doesn't add any fields. If you do override this method, it should first call the same method on **super** before doing any further work. For example:

```
-(id)copyWithZone:(NSZone*)zone {
  MyClass* copy = [super copyWithZone:zone];
  // Further class-specific setup of copy.
  return copy;
}
```

If the inherited method returns a shallow copy, the copy will have two properties you will have to correct:

- Its reference count will be identical to your own, and you will have to set the count to 1.
- If it has any pointers to reference-counted objects, the counts of those objects will not be properly incremented

by 1 to reflect their additional owner. You should set these fields to **nil** and then initialize them using the appropriate setter method.

WARNING

If you don't set a reference to **nil** first, the setter method may call -release on the referent, leaving its reference count permanently off by one.

If objects of your class are immutable, or there is only one instance that is shared, just call -retain on the receiver and return it to the caller. For example:

```
-(id)copyWithZone:(NSZone*)zone {
  return [self retain];
}
```

Deallocating an Object

The deallocation method of a class complements both the allocation and initializers, undoing the work they did.

Calling deallocation methods

The Object class provides a -free method. Calling this method releases the memory directly associated with the object. Memory directly associated with an object includes the space taken up by that object's fields. If a field is a pointer to an object or other structure, only the pointer will be freed. Subclasses should override this method to adapt its behavior to free additional resources held by the receiver.

The NSObject class provides an analogous method called -dealloc. It has the same behavior as -free, but you don't normally call -dealloc yourself on an object. Instead you use the reference counting methods for memory management, provided by Cocoa. When an object's reference count goes to zero, -dealloc is automatically called. See the "Memory

Management” section for information about using reference counting.

Writing deallocation methods

If you’re using the `Object` class, your code will call the `-free` methods directly. In Cocoa, the runtime will send a `-dealloc` message to your object when its memory is being freed via the `-release` message. In either case you use the same approach to writing the deallocators.

Your class’s deallocator should first release all resources that it has acquired, *then* call its parent class’s deallocator method:

```
-(id)free {
  // Release held resources.
  [super free];  // Tail call.
}
```

Notice that this chaining order is reverse that of initializing: a tail call instead of a head call.

NOTE

Descendants of `NSObject` will use `-release` instead of `-free`.

Resources to be released include Objective-C objects held by reference (call `-free` or `-release` on these) and external shared entities like network sockets. The root class deallocator releases the memory of the object (i.e., its fields) itself.