

---

# Memory Management Programming Guide for Cocoa

Cocoa > Objective-C Language



2008-02-08



Apple Inc.  
© 2008 Apple Inc.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.  
1 Infinite Loop  
Cupertino, CA 95014  
408-996-1010

Apple, the Apple logo, Carbon, Cocoa, Mac, Mac OS, Objective-C, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND

YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

# Contents

---

## [Introduction to Memory Management Programming Guide for Cocoa](#) 7

---

[Who Should Read This Document](#) 7

[Organization of This Document](#) 7

---

## [Object Ownership and Disposal](#) 9

---

[Object Ownership Policy](#) 9

[Creating Objects Using Convenience Methods](#) 10

[Objects Returned by Reference](#) 11

[Delayed Release](#) 11

[Owning Objects](#) 12

[Validity of Shared Objects](#) 13

[Retain Cycles](#) 13

[Weak References to Objects](#) 14

[Deallocating an Object](#) 15

[Summary](#) 16

---

## [Practical Memory Management](#) 17

---

[Basics](#) 17

[Simple Examples](#) 18

[Using Accessor Methods](#) 19

[Implementing a reset method](#) 20

[Common Mistakes](#) 20

[Cases which Often Cause Confusion](#) 21

---

## [Allocating and Initializing Objects](#) 23

---

[Objective-C Creation and Initialization Methods](#) 23

[How to Allocate and Initialize Objective-C Objects](#) 23

---

## [Autorelease Pools](#) 25

---

[Overview of Autorelease Pools](#) 25

[Autorelease Pools in Non-AppKit Programs](#) 26

[Autorelease Pools and Threads](#) 27

[Scope of Autorelease Pools and Implications of Nested Autorelease Pools](#) 27

[Guaranteeing the Foundation Ownership Policy](#) 28

## Accessor Methods 31

---

Value Objects and Copying 32

## Implementing Object Copy 33

---

Deep Versus Shallow Copies 33

Independent Copy 34

Inheriting NSCopying from the Superclass 34

Using the “alloc, init...” Approach 35

Using NSCopyObject() 35

Copying Mutable Versus Immutable Objects 37

## Memory Management of Core Foundation Objects in Cocoa 39

---

### Using Memory Zones 41

---

Creating and Managing Zones 41

Allocating Memory in Zones 42

### Memory Management of Nib Objects 43

---

### Memory Management Rules 45

---

### Document Revision History 47

---

# Figures and Listings

## Object Ownership and Disposal 9

---

Figure 1 An illustration of retain cycles 14

## Autorelease Pools 25

---

Listing 1 Example of a `main` function for a non-AppKit program 26

## Implementing Object Copy 33

---

Figure 1 Copying instance variables both shallowly and deeply 34

Figure 2 Initialization of the reference count during a copy 37

## Memory Management of Nib Objects 43

---

Figure 1 Top-level objects in a nib file 43



# Introduction to Memory Management Programming Guide for Cocoa

---

Memory management, especially as it concerns Objective-C programs, is an important subject. Some of the more common problems encountered by novice application developers derive from poor memory management. Cocoa provides mechanisms and a policy to assist you in the proper creation, retention, and disposal of objects.

Memory management in Cocoa is largely object-oriented. This programming topic addresses the object-ownership policy and related techniques for creating, copying, retaining, and disposing of objects using “managed memory”.

**Important:** In Mac OS X v10.5 and later, you can use automatic memory management by adopting garbage collection. This is described in *Garbage Collection Programming Guide*.

## Who Should Read This Document

---

The material in this programming topic relates to Objective-C objects and is of interest primarily to Objective-C programmers. If you are starting a new project targeted at Mac OS X v10.5 and later, you should typically use garbage collection unless you have good reason to use the techniques described here.

## Organization of This Document

---

This document contains the following articles:

- [“Object Ownership and Disposal”](#) (page 9) describes the primary policy.
- [“Allocating and Initializing Objects”](#) (page 23) explains techniques for allocating and initializing Objective-C objects.
- [“Autorelease Pools”](#) (page 25) describes the use of autorelease pools—a mechanism for deferred deallocation—in Cocoa programs.
- [“Accessor Methods”](#) (page 31) describes how to implement accessor methods.

- [“Implementing Object Copy”](#) (page 33) discusses issues related to object copying, such as deciding whether to implement a deep or shallow copy and approaches for implementing object copy in your subclasses.
- [“Memory Management of Core Foundation Objects in Cocoa”](#) (page 39) gives guidelines and techniques for memory management of Core Foundation objects in Cocoa code.
- [“Using Memory Zones”](#) (page 41) discusses the use of memory zones.
- [“Memory Management Rules”](#) (page 45) summarizes the rules for object ownership and disposal.

Additional information about memory management and nib files can be found in *Resource Programming Guide* > Nib Files and Cocoa > “The Nib Object Life Cycle.”



# Object Ownership and Disposal

---

This document discusses the policy for ownership of Objective-C objects and how and when to dispose of objects.

To fully understand how the object ownership policy is implemented in Cocoa, you must also read [“Autorelease Pools”](#) (page 25).

## Object Ownership Policy

---

In an Objective-C program, objects are constantly creating and disposing of other objects. It is important to dispose of objects when they are no longer needed to ensure that your application does not use more memory than necessary. Much of the time an object creates things for private use and can dispose of them as needed. However, when an object passes something to another object through a method invocation, the lines of ownership—and responsibility for disposal—blur. For example, suppose you have a Thingamajig object that contains a number of Sprocket objects, which other objects access using the following method:

```
- (NSArray *)sprockets
```

This declaration says nothing about who should dispose of the returned array. It is reasonable to suggest, however, that if your Thingamajig object returns an instance variable, it is responsible for the array. If on the other hand you create a new Thingamajig object, then you are responsible for disposing of the new object. This, though, introduces a possible source of confusion. “Disposal” tends to imply “get rid of” or “deallocate”.

As noted earlier, it is possible (in fact common) for one object to create another object and then pass it to another. It is important not to get rid of the new object until the third party has finished using it. It is better, therefore, to think of memory management in terms of object ownership, where any object may have one or more owner. So long as an object has at least one owner, it continues to exist. If an object has no owners, the runtime system disposes of it (deallocates it) automatically.

To make sure it is clear when you own an object and when you do not, and what responsibilities you have as an owner, Cocoa sets the following policy:

- You own any object you create.

You “create” an object using a method whose name begins with “alloc” or “new” or contains “copy” (for example, `alloc`, `newObject`, or `mutableCopy`).

- If you own an object, you are responsible for relinquishing ownership when you have finished with it.

One way to relinquish ownership of an object is to send it a `release` message. In Cocoa terminology, relinquishing ownership of an object is typically referred to as "releasing" an object.

- If you do not own an object, you must not release it.

This policy applies both to GUI-based Cocoa applications and to command-line Foundation tools.

Consider the following example:

```
Thingamajig *thingamajig = [[Thingamajig alloc] init];
// ...
NSArray *sprockets = [thingamajig sprockets];
// ...
[thingamajig release];
```

This example properly adheres to the policy. You create the `Thingamajig` object using the `alloc` method, so you subsequently send it a `release` message. You obtain the `sprockets` array from the `Thingamajig` object—you do not "create" the array—so you do not send it a `release` message.

## Creating Objects Using Convenience Methods

---

Many classes provide methods of the form `+className...` that you can use to obtain a new instance of the class. Often referred to as "convenience constructors", these methods create a new instance of the class, initialize it, and return it for you to use. Although you might think you are responsible for releasing objects created in this manner, that is not the case according to the policy Cocoa set—the method name does not contain "alloc" or "copy", or begin with "new". Because *the class* creates the new object, *it* is responsible for disposing of the new object. As an illustration, the following code example is *wrong*:

```
Thingamajig *thingamajig = [Thingamajig thingamajig];
[thingamajig release]; // wrong
```

Although if you try this you will not see an error as soon as the `release` message is sent, it will cause an exception later (for some definition of later, as discussed in ["Autorelease Pools"](#) (page 25)).

This does, though, raise the issue of how the `Thingamajig` class can abide by the ownership policy. It is responsible for releasing the new object, but it must not do so before the recipient has had a chance to claim ownership. To illustrate, consider two possible implementations of the `thingamajig` method.

1. This is *wrong* because after the new `Thingamajig` is returned to the caller the class loses its reference to the new object so cannot send it a `release` message to relinquish ownership:

```
+ (Thingamajig *)thingamajig
{
    id newThingamajig = [[Thingamajig alloc] init];
    return newThingamajig;
}
```

2. This is also *wrong* because although the class properly relinquishes ownership of the new object, after the `release` message is sent the new `Thingamajig` object has no owner so is immediately disposed of by the system:

```

+ (Thingamajig *)thingamajig
{
    id newThingamajig = [[Thingamajig alloc] init];
    [newThingamajig release];
    return newThingamajig; // newThingamajig is invalid here
}

```

The Thingamajig class needs a way to mark an object for relinquish ownership at a later time, after the recipient has had a chance to use it. Cocoa provides a mechanism to do this, called "autoreleasing", discussed in [“Delayed Release”](#) (page 11).

## Objects Returned by Reference

---

Some methods in Cocoa specify that an object is returned by reference. There are several examples that use an NSError object that contains information about an error if one occurs, such as:

- initWithContentsOfURL:ofType:error: (NSDocument)
- initWithContentsOfURL:options:error: (NSData)
- initWithContentsOfFile:encoding:error: (NSString)

In these cases, the same rules apply as have already been described. When you invoke any of these methods, you do not create the NSError object so you do not own it—there is therefore no need to release it.

```

NSString *fileName = ... ;
NSError *error;
NSString *string = [[NSString alloc] initWithContentsOfFile:fileName
                                encoding:NSUTF8StringEncoding error:&error];
if (string == nil) {
    // deal with error ...
}
// ...
[string release];

```

If for any reason ownership of returned object does not follow the basic rules, this is stated explicitly in the documentation for the method (see for example, `dataFromPropertyList:format:errorDescription:`).

## Delayed Release

---

The `autorelease` method, defined by NSObject, marks the receiver for later release. By autoreleasing an object—that is, by sending it an `autorelease` message—you declare that you don't want to own the object beyond the scope in which you sent `autorelease`. The scope is defined by the current autorelease pool—see [“Autorelease Pools”](#) (page 25).

The sprockets method mentioned above could be implemented in this way:

```

- (NSArray *)sprockets
{

```

```

NSArray *array;

array = [[NSArray alloc] initWithObjects:mainSprocket,
                                         auxiliarySprocket, nil];
return [array autorelease];
}

```

When another method gets the array of Sprocket objects, that method can assume that the array will be disposed of when it is no longer needed, but can still be safely used anywhere within its scope (see [“Validity of Shared Objects”](#) (page 13)). It can even return the array to its invoker, since the application object defines the bottom of the call stack for your code. The `autorelease` method thus allows every object to use other objects without worrying about disposing of them.

Just as it is an error to release an object after it is already been deallocated, it’s an error to send so many `autorelease` messages that the object would later be released after it had already been deallocated. You should send `release` or `autorelease` to an object only as many times as are allowed by its creation (one) plus the number of `retain` messages you have sent it (`retain` messages are described below).

## Owning Objects

---

There are times when you don’t want a received object to be disposed of; for example, you may need to cache the object in an instance variable. In this case, only you know when the object is no longer needed, so you need the power to ensure that the object is not disposed of while you are still using it. You do this with a `retain` message, which stays the effect of a pending `autorelease` (or preempts a later `release` or `autorelease` message). By retaining an object you ensure that it won’t be deallocated until you are done with it. For example, if your object allows its main Sprocket to be set, you might want to retain that Sprocket like this:

```

- (void)setMainSprocket:(Sprocket *)newSprocket
{
    [mainSprocket autorelease];
    mainSprocket = [newSprocket retain]; /* Claim the new Sprocket. */
    return;
}

```

Now, `setMainSprocket:` might get invoked with a Sprocket that the invoker intends to keep around, which means your object would be sharing the Sprocket with that other object. If that object changes the Sprocket, your object’s main Sprocket changes. You might want that, but if your Thingamajig needs to have its own Sprocket the method should make a private copy:

```

- (void)setMainSprocket:(Sprocket *)newSprocket
{
    [mainSprocket autorelease];
    mainSprocket = [newSprocket copy]; /* Get a private copy. */
    return;
}

```

Note that both of these methods `autorelease` the original main sprocket, so they don’t need to check that the original main sprocket and the new one are the same. If they simply released the original when it was the same as the new one, that sprocket would be released and possibly deallocated, causing an error as soon as it was retained or copied. The following code solves that problem:

```
- (void)setMainSprocket:(Sprocket *)newSprocket
{
    if (mainSprocket != newSprocket) {
        [mainSprocket release];
        mainSprocket = [newSprocket retain];
    }
}
```

## Validity of Shared Objects

---

Cocoa’s ownership policy specifies that received objects should remain valid throughout the scope of the calling method. It should also be possible to return a received object from the current scope without fear of it being released. It should not matter to your application that the getter method of an object returns a cached instance variable or a computed value. What matters is that the object remains valid for the time you need it.

There are exceptions to this rule. For example, collection classes do not attempt to extend the lifetime of objects placed inside them. Removing an object from a mutable array could invalidate any copies of the object previously acquired, as in the following example:

```
value = [array objectAtIndex:n];
[array removeObjectAtIndex:n];
// value could now be invalid.
```

Another problem situation is when an object is deallocated after a call to one of its getter methods:

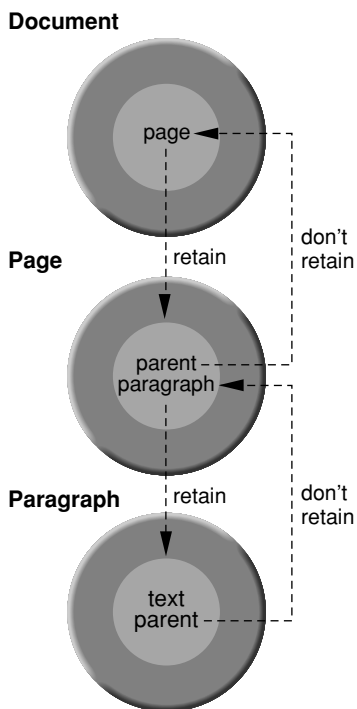
```
sprocket = [thingamajig mainSprocket];
[thingamajig release];
// sprocket could now be invalid.
```

To protect against situations like this, you could retain `sprocket` upon receiving it and release it when you have finished with it. Because it may not always be obvious when a caller should retain an object in this manner, the objects themselves should strive to return results that are valid in the current calling scope. In many cases, understanding how accessor methods are implemented, and implementing accessor methods appropriately, will resolve any confusion—see [“Accessor Methods”](#) (page 31).

## Retain Cycles

---

In some situations, two objects may have cyclical references; that is, each object contains an instance variable that refers to the other object. For example, consider a text program with the object relationships shown in [Figure 1](#) (page 14). The Document object creates a Page object for each page in the document. Each Page object has an instance variable that keeps track of which document it is in. If the Document object retained the Page object and the Page object retained the Document object, neither object would ever be released. The Document’s reference count cannot become 0 until the Page object is released, and the Page object won’t be released until the Document object is deallocated.

**Figure 1** An illustration of retain cycles

The solution to the problem of retain cycles is that the “parent” object should retain its “children,” but that the children should not retain their parents. So, in [Figure 1](#) (page 14) the document object retains its page objects but the page object does not retain the document object. The child’s reference to its parent is an example of a weak reference, which is described more fully in [“Weak References to Objects”](#) (page 14).

## Weak References to Objects

Retaining an object creates a “strong” reference to that object. An object cannot be deallocated until all of its strong references are released. An object’s lifetime is thereby determined by the owners of its strong references. In some cases, this behavior may not be desired. You may want to have a reference to an object without preventing the object from deallocating itself. For these cases, you can obtain a “weak” reference. A weak reference is created by storing a pointer to an object without retaining the object.

Weak references are essential in cases where a circular reference would otherwise be set up. For example, if Object A and Object B communicate with each other, each needs a reference to the other. If each retains the other, neither object ever gets deallocated until the connection is broken, but the connection is not broken until one of the objects is deallocated. Catch-22. To break the circle, one object takes a subordinate role and obtains a weak reference to the other. As a concrete example, in a view hierarchy, a parent view owns, and hence retains, its child views, but a child view does not own its parent; the child still needs to know who its parent is, so it keeps a weak reference to its parent.

Additional cases of weak references in Cocoa include, but are not restricted to, table data sources, outline view items, notification observers, and miscellaneous targets and delegates.

**Important:** In Cocoa, references to table data sources, outline view items, notification observers, and delegates are all considered weak (for example, an `NSTableView` object does not retain its data source and the `NSApplication` object does not retain its delegate). The documentation only describes exceptions to this convention.

You need to be careful about sending messages to objects for which you only hold a weak reference. If you send a message to an object after it has been deallocated, your application will crash. You must have well-defined conditions for when the object is valid. In most cases, the weak-referenced object is aware of the other object's weak reference to it, as is the case for circular references, and is responsible for notifying the other object when it deallocates. For example, when you register an object with a notification center, the notification center stores a weak reference to the object and sends messages to it when the appropriate notifications are posted. When the object is deallocated, you need to unregister it with the notification center to prevent the notification center from sending any further messages to the object, which no longer exists. Likewise, when a delegate object is deallocated, you need to remove the delegate link by sending a `setDelegate:` message with a `nil` argument to the other object. These messages are normally sent from the object's `dealloc` method.

## Deallocating an Object

---

Cocoa implements its ownership policy through a mechanism called “reference counting” or “retain counting”—see *The Runtime System*. When you create an object, it has a retain count of 1. When you send an object a `retain` message, its retain count is increased by 1. When you send an object a `release` message, its retain count is decreased by 1 (`autorelease` causes the retain count to be decremented in the future).

**Important:** Typically there should be no reason to explicitly ask an object what its retain count is. The result is often misleading, as you may be unaware of what framework objects have retained an object in which you are interested. In debugging memory management issues, you should be concerned only with ensuring that your code adheres to the ownership rules—see [“Summary”](#) (page 16).

When its retain count drops to 0, an object's memory is reclaimed—in Cocoa terminology it is “freed” or “deallocating”. When an object is deallocated, its `dealloc` method is invoked automatically. The role of the `dealloc` method is to free the object's own memory, and dispose of any resources it holds, including its object instance variables.

If your class has object instance variables, you must implement a `dealloc` method that releases them, and then invokes super's implementation. For example, if the `Thingamajig` class had `name` and `sprockets` instance variables, you would implement its `dealloc` method as follows:

```
- (void)dealloc
{
    [sprockets release];
    [name release];
    [super dealloc];
}
```

You should never invoke another object's `dealloc` method directly.

Note that when an application terminates, objects may not be sent a `dealloc` message since the process's memory is automatically cleared on exit—it is more efficient simply to allow the operating system to clean up resources than to invoke all the memory management methods. For more details about object creation and deallocation, see [Object Creation](#).

## Summary

---

Now that the concepts behind the Cocoa's object ownership policy have been introduced, they can be expressed as a short list of rules—see [“Memory Management Rules”](#) (page 45).



# Practical Memory Management

---

This article provides a practical perspective on memory management.

Cocoa does not use garbage collection. You must do your own memory management through reference counting. Following a few simple rules can make memory management easy. Failure to adhere to the rules will almost certainly lead at some point to memory leaks, or runtime exceptions due to messages being sent to freed objects.

## Basics

---

To keep memory consumption as low as possible in an application, you should get rid of objects that are not being used, but you need to make sure that you don't get rid of an object that is being used. You therefore need a mechanism that allows you to mark an object as still being useful. In many respects, memory management is thus best understood in terms of "object ownership."

- An object may have one or more owners.  
(By way of an analogy, consider a timeshare apartment.)
- When an object has no owners, it is destroyed.  
(To stretch an analogy, consider a timeshare complex that is not loved by the local population. If there are no owners, the complex will be torn down.)
- To make sure an object you're interested in is not destroyed, you must become an owner.  
(You can either build a new apartment, or take a stake in an existing one.)

To support this model, Cocoa provides a mechanism called "reference counting" or "retain counting." Every object has a retain count. An object is created with a retain count of 1. When the retain count drops to 0, an object is deallocated (destroyed). You manipulate the retain count (take and relinquish ownership) using a variety of methods:

`alloc`

Allocates memory for an object, and returns it with retain count of 1  
You own objects you create using `alloc`.

`copy`

Makes a copy of an object, and returns it with retain count of 1  
If you copy an object, you own the copy.

#### retain

Increases the reference count of an object by 1

Takes ownership of an object.

#### release

Decreases the reference count of an object by 1

Relinquishes ownership of an object.

#### autorelease

Decreases the reference count of an object by 1 at some stage in the future

Relinquishes ownership of an object at some stage in the future.

The following rules apply:

- Within a given block of code, the number of times you use `copy`, `alloc` and `retain` should equal the number of times you use `release` and `autorelease`.
- You only own objects you created using a method whose name begins with “alloc” or “new” or contains “copy” (for example, `alloc`, `newObject`, or `mutableCopy`), or if you send it a `retain` message.
- Implement a `dealloc` method to release the instance variables you own.
- You should never invoke `dealloc` directly (other than when you invoke `super`’s implementation in a custom `dealloc` method).

Many classes provide methods of the form `+className...` that you can use to obtain a new instance of the class. Often referred to as “convenience constructors”, these methods create a new instance of the class, initialize it, and return it for you to use. You do not own objects returned from convenience constructors, or from other accessor methods.

## Simple Examples

---

The following simple examples illustrate the contrast between creating a new object using `alloc`, using a convenience constructor, and using an accessor method.

The first example creates a new string object using `alloc`. It must therefore be released.

```
- (void)printHello
{
    NSString *string;
    string = [[NSString alloc] initWithString:@"Hello"];
    NSLog(string);
    [string release];
}
```

The second example creates a new string object using a convenience constructor. There is no additional work to do.

```
- (void)printHello
{
    NSString *string;
    string = [NSString stringWithFormat:@"Hello"];
    NSLog(string);
}
```

```
}
```

The third example retrieves a string object using an accessor method. As with the convenience constructor, there is no additional work to do.

```
- (void)printWindowTitle
{
    NSString *string;
    string = [myWindow title];
    NSLog(string);
}
```

## Using Accessor Methods

---

Sometimes it might seem tedious or pedantic, but if you use accessor methods consistently the chances of having problems with memory management decrease considerably. If you are using `retain` and `release` on a class's instance variables throughout your code, you are almost certainly doing the wrong thing.

Consider a `Counter` object whose count you want to set.

```
@interface Counter : NSObject
{
    NSNumber *count;
}
```

To get and set the count, you define two accessor methods. In the get accessor, you just pass back a variable so there is no need for `retain` or `release`:

```
- (NSNumber *)count
{
    return count;
}
```

In the set method, if everyone else is playing by the same rules you have to assume the new count may be disposed of at any time so you have to take ownership of the object—by sending it a `retain` message—to ensure it won't be. You must also relinquish ownership of the old count object here by sending it a `release` message. (Sending a message to `nil` is allowed in Objective-C, so this will still work if count hasn't yet been set.) You must send this after `[newCount retain]` in case the two are the same object—you don't want to inadvertently cause it to be deallocated.

```
- (void)setCount:(NSNumber *)newCount
{
    [newCount retain];
    [count release];
    // make the new assignment
    count = newCount;
}
```

These examples present a simple perspective on accessor methods. They are described in greater detail in [“Accessor Methods”](#) (page 31).

Since the `Counter` class has an object instance variable, you must also implement a `dealloc` method:

```
- (void)dealloc
{
    [self setCount:nil];
    [super dealloc];
}
```

## Implementing a reset method

---

Suppose you want to implement a method to reset the counter. You have a couple of choices. The first uses a convenience constructor to create a new `NSNumber` object—there is therefore no need for any retain or release messages. Note that both use the class’s set accessor method.

```
- (void)reset
{
    NSNumber *zero = [NSNumber numberWithInt:0];
    [self setCount:zero];
}
```

The second creates the `NSNumber` instance with `alloc`, so you balance that with a `release`.

```
- (void)reset
{
    NSNumber *zero = [[NSNumber alloc] initWithInt:0];
    [self setCount:zero];
    [zero release];
}
```

## Common Mistakes

---

The following sections illustrate common mistakes.

### Accessor not used

---

The following will almost certainly work correctly for simple cases, but tempting as it may be to eschew accessor methods, this will also almost certainly lead to a mistake at some stage.

```
- (void)reset
{
    NSNumber *zero = [[NSNumber alloc] initWithInt:0];
    [count release];
    count = zero;
}
```

Note in particular that if you are using key-value observing (see *Key-Value Observing Programming Guide*), then changing the variable in this way is not KVO-compliant.

### Instance leaks

---

```
- (void)reset
{
    NSNumber *zero = [[NSNumber alloc] initWithInt:0];
    [self setCount:zero];
}
```

```
}
```

The retain count of the new number is 1 (from `alloc`) and is not balanced by a release within the scope of the method. The new number is unlikely ever to be freed, which will result in a memory leak.

### Instance you don't own is sent release

---

```
- (void)reset
{
    NSNumber *zero = [NSNumber numberWithInt:0];
    [self setCount:zero];
    [zero release];
}
```

Absent any other invocations of `retain`, this will fail the first time you subsequently access `count` after the current autorelease pool has been released. The convenience constructor method returns an autoreleased object, so you don't have to send another release. Doing so will mean that when the release due to `autorelease` is sent, it will reduce the retain count to 0, and the object will be freed. When you next access `count` you will be sending a message to a freed object (typically you'll get a SIGBUS 10 error).

## Cases which Often Cause Confusion

---

When you add an object to a collection such as an array, dictionary, or set, the collection takes ownership of it. The collection will relinquish ownership when the object is removed from the collection or when the collection is itself released. Thus, for example, if you want to create an array of numbers you might do either of the following:

```
NSMutableArray *array;
int i;
// ...
for (i = 0; i < 10; i++) {
    NSNumber *convenienceNumber = [NSNumber numberWithInt: i];
    [array addObject: convenienceNumber];
}
```

In this case, you didn't invoke `alloc`, so there's no need to call `release`. There is no need to retain the new numbers (`convenienceNumber`), since the array will do so.

```
NSMutableArray *array;
int i;
// ...
for (i = 0; i < 10; i++) {
    NSNumber *allocatedNumber = [[NSNumber alloc] initWithInt: i];
    [array addObject: allocatedNumber];
    [allocatedNumber release];
}
```

In this case *do* you need to send `allocatedNumber` a release message within the scope of the `for` loop to balance the `alloc`. Since the array retained the number when it was added by `addObject:`, it will not be deallocated while it's in the array.

To understand this, put yourself in the position of the person who implemented the collection class. You want to make sure that no objects you're given to look after disappear out from under you, so you send them a `retain` message as they're passed in. If they're removed, you have to send a balancing `release` message, and any remaining objects should be sent a `release` message during your own `dealloc` method.

# Allocating and Initializing Objects

---

This document describes how to allocate memory for and initialize Cocoa objects. It also discusses some related issues.

## Objective-C Creation and Initialization Methods

---

The class methods `alloc`, `allocWithZone:` and `new` allocate memory for an object *and* initialize that object's reference to its class. Every object that inherits directly or indirectly from `NSObject` is connected to the run-time system through its `isa` instance variable. `isa` identifies the object's class; it references a structure that is compiled from the class definition. Through `isa`, an object can find whatever information it needs at run-time—such as its place in the inheritance hierarchy, the size and structure of its instance variables, and the location of the method implementations it can perform in response to messages. The object-creation methods also initialize all instance variables to zero (or `nil`, `NULL`, or whatever is appropriate for that type).

**Note:** Objects created and returned by these methods are owned by the receiving object, which is responsible for their disposal.

The initialization methods—instance methods of each class with the name of `init` or a name beginning with `init`—initialize individual objects by setting their instance variables to initial values.

In addition, classes frequently define “factory” methods—convenience class methods—that allocate and initialize instances for the receiver.

## How to Allocate and Initialize Objective-C Objects

---

Invoke `alloc` or `allocWithZone:` on a class to create an instance of that class. The `alloc` method uses the default zone (that is, the zone returned by the `NSDefaultMallocZone` function) when it allocates memory for an object.

```
TheClass *newObject = [TheClass alloc];
```

Once you have created an object with `alloc`, `allocWithZone:`, or `new`, you can initialize the object with an `init...` method. Typically the allocation and initialization messages are combined in one statement.

```
TheClass *newObject = [[TheClass alloc] init];
```

In some cases, an `init` method might release the new object and return a substitute. Programs should therefore always use the object returned by `init` and not necessarily the one returned by `alloc` or `allocWithZone:` in subsequent code as illustrated in the following code sample. Note the assignment of `self` in the test, and use of two pairs of parentheses to avoid compiler warnings with some strict flags.

```
- (id)init {
    if ((self = [super init])) { // superclass may return nil
        // your initialization code goes here
    }
    return self;
}
```

Subclasses should implement `init...` to return the new object (`self`) after it has been successfully initialized; these methods should first invoke `super` to incorporate the initialization code for the classes they inherit from. If the instance cannot be initialized, they should release the object and return `nil`.

When one object creates another, it is often a good idea to make sure they are both allocated from the same region of memory. The `zone` method (declared in the `NSObject` protocol) can be used for this purpose; it returns the zone where the receiver is located. For example:

```
id myCompanion = [[TheClass allocWithZone:[self zone]] init];
```

Note that it is your responsibility to release objects (with either `release` or `autorelease`) returned by the object-creation methods.



# Autorelease Pools

---

This document contains information on fine-tuning your application's handling of autorelease pools; see the document [“Object Ownership and Disposal”](#) (page 9) for general information on using the autorelease mechanism.

## Overview of Autorelease Pools

---

An autorelease pool is an instance of `NSAutoreleasePool` that “contains” other objects that have received an `autorelease` message; when the autorelease pool is deallocated it sends a `release` message to each of those objects. An object can be put into an autorelease pool several times, and receives a `release` message for each time it was put into the pool. Thus, sending `autorelease` instead of `release` to an object extends the lifetime of that object at least until the pool itself is released (the object may survive longer if it is retained in the interim).

Cocoa always expects there to be an autorelease pool available. If a pool is not available, autoreleased objects do not get released and you leak memory. If you send an autorelease message when a pool is not available, Cocoa logs a suitable error message.

You create an `NSAutoreleasePool` object with the usual `alloc` and `init` messages, and dispose of it with `release` (an exception is raised if you send `autorelease` or `retain` to an autorelease pool). An autorelease pool should always be released in the same context (invocation of a method or function, or body of a loop) in which it was created.

Autorelease pools are arranged in a stack, although they are commonly referred to as being “nested.” When you create a new autorelease pool, it is added to the top of the stack. When pools are deallocated, they are removed from the stack. When an object is sent an `autorelease` message, it is added to the current topmost pool for the current thread.

The ability to nest autorelease pools means that you can include them in any function or method. For example, a `main` function may create an autorelease pool and call another function that creates another autorelease pool. Or a single method might have an autorelease pool for an outer loop, and another autorelease pool for an inner loop. The ability to nest autorelease pools is a definite advantage, but there are side effects when exceptions occur (see [“Scope of Autorelease Pools and Implications of Nested Autorelease Pools”](#) (page 27)).

The Application Kit automatically creates a pool at the beginning of an event cycle (or event-loop iteration), such as a mouse down event, and releases it at the end, so your code normally does not have to worry about them. There are three cases, though, where you might create and destroy your own autorelease pools:

- If you are writing a program that is not based on the Application Kit, such as a command-line tool, there is no built-in support for autorelease pools; you must create and destroy them yourself.
- If you spawn a secondary thread, you must create your own autorelease pool as soon as the thread begins executing; otherwise, you will leak objects. (See [“Autorelease Pools and Threads”](#) (page 27) for details.)
- If you write a loop that creates many temporary objects, you may create an autorelease pool inside the loop to dispose of those objects before the next iteration. This can help reduce the maximum memory footprint of the application.

Autorelease pools are used “in line”. *There should typically be no reason why you should make an autorelease pool an instance variable of an object.*

## Autorelease Pools in Non-AppKit Programs

---

Enabling the autorelease mechanism in a program that is not based on the Application Kit is easy. You can simply create an autorelease pool at the beginning of the `main()` function, and release it at the end—this is the pattern used by the Foundation Tool template in Xcode. This establishes a pool for the lifetime of the task. However, this also means that any autoreleased objects created during the lifetime of the task are not disposed of until the task completes. This may lead to the task’s memory footprint increasing unnecessarily. You can also consider creating pools with a narrower scope.

Many programs have a high-level loops where they do much of their work. To enable the autorelease mechanism you can create an autorelease pool at the beginning of an iteration through this loop and release it at the end.

Your `main` function might look like the code in Listing 1.

### Listing 1 Example of a `main` function for a non-AppKit program

```
void main()
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    NSArray *args = [[NSProcessInfo processInfo] arguments];
    unsigned count, limit = [args count];

    for (count = 0; count < limit; count++)
    {
        NSAutoreleasePool *loopPool = [[NSAutoreleasePool alloc] init];
        NSString *fileContents;
        NSString *fileName;

        fileName = [args objectAtIndex:count];
        fileContents = [[[NSString alloc] initWithContentsOfFile:fileName]
autorelease];
        // this is equivalent to using stringWithContentsOfFile:
```

```

        /* Process the file, creating and autoreleaseing more objects. */

        [loopPool release];
    }

    /* Do whatever cleanup is needed. */
    [pool release];

    exit (EXIT_SUCCESS);
}

```

This program processes files passed in on the command line. The `for` loop processes one file at a time. An `NSAutoreleasePool` object is created at the beginning of this loop and released at the end. Therefore, any object sent an `autorelease` message inside the loop (such as `fileContents`) is added to `loopPool`, and when `loopPool` is released at the end of the loop those objects are also released. Additionally, any autoreleased objects created in the context of the `for` loop (such as `fileName`) are released when `loopPool` is released even if they're not explicitly sent an `autorelease` message.

## Autorelease Pools and Threads

---

Each thread in a Cocoa application maintains its own stack of `NSAutoreleasePool` objects. When a thread terminates, it automatically releases all of the autorelease pools associated with itself. Autorelease pools are automatically created and destroyed in the main thread of applications based on the Application Kit, so your code normally does not have to deal with them there. If you are making Cocoa calls outside of the Application Kit's main thread, however, you need to create your own autorelease pool. This is the case if you are writing a Foundation-only application or if you detach a thread.

If your application or thread is long-lived and potentially generates a lot of autoreleased objects, you should periodically destroy and create autorelease pools (like the Application Kit does on the main thread); otherwise, autoreleased objects accumulate and your memory footprint grows. If your detached thread does not make Cocoa calls, you do not need to create an autorelease pool.

**Note:** If you create secondary threads using the POSIX thread APIs instead of `NSThread`, you cannot use Cocoa—including `NSAutoreleasePool`—unless Cocoa is in multithreading mode. Cocoa enters multithreading mode only after detaching its first `NSThread` object. To use Cocoa on secondary POSIX threads, your application must first detach at least one `NSThread` object, which can immediately exit. You can test whether Cocoa is in multithreading mode with the `NSThread` class method `isMultiThreaded`.

## Scope of Autorelease Pools and Implications of Nested Autorelease Pools

---

It is common to speak of autorelease pools as being nested because of the enclosure evident in code, as illustrated in [Listing 1](#) (page 26). But you can also think of nested autorelease pools as being on a stack, with the “innermost” autorelease pool being on top of the stack. As noted earlier, this is actually

how nested autorelease pools are implemented: Each thread in a program maintains a stack of autorelease pools. When you create an autorelease pool, it is pushed onto the top of the current thread's stack. *When an object is autoreleased—that is, when an object is sent an autorelease message or when it passed as the argument to the `addObject:` class method—it is always put in the autorelease pool at the top of the stack.*

The scope of an autorelease pool is therefore defined by its position in the stack and the simple fact of its existence. The topmost pool is the pool to which autoreleased objects are added. If another pool is created, the current topmost pool effectively goes out of scope until the new pool is released (at which point the original pool once again becomes the topmost pool). It (obviously) goes out of scope permanently when it is itself released.

If you release an autorelease pool that is not the top of the stack, this causes all (unreleased) autorelease pools above it on the stack to be released, along with all their objects. If you neglect to send `release` to an autorelease pool when you are finished with it (something not recommended), it is released when one of the autorelease pools in which it nests is released.

This behavior has implications for exceptional conditions. If an exception occurs, and the thread suddenly transfers out of the current context, the pool associated with that context is released. However, if that pool is not the top pool on the thread's stack, all the pools above the released pool are also released (releasing all their objects in the process). The top autorelease pool on the thread's stack then becomes the pool previously underneath the released pool associated with the exceptional condition. Because of this behavior, exception handlers do not need to release objects that were sent `autorelease`. Neither is it necessary or even desirable for an exception handler to send `release` to its autorelease pool, unless the handler is re-raising the exception.

## Guaranteeing the Foundation Ownership Policy

---

By creating an autorelease pool instead of simply releasing objects, you extend the lifetime of temporary objects to the lifetime of that pool. After an autorelease pool is deallocated, you should regard any object that was autoreleased while that pool was active as “disposed of”, and not send a message to that object or return it to the invoker of your method.

If you must use a temporary object beyond an autorelease context, you can do so by sending a `retain` message to the object within the context and then send it `autorelease` after the pool has been released as in:

```
- findMatchingObject:anObject
{
    id match = nil;

    while (match == nil) {
        NSAutoreleasePool *subPool = [[NSAutoreleasePool alloc] init];

        /* Do a search that creates a lot of temporary objects. */
        match = [self expensiveSearchForObject:anObject];

        if (match != nil)
            [match retain]; /* Keep match around. */

        [subPool release];
    }
}
```

```
    return [match autorelease];    /* Let match go and return it. */  
}
```

By sending `retain` to `match` while `subpool` is in effect and sending `autorelease` to it after `subpool` has been released, `match` is effectively moved from `subpool` to the pool that was previously active. This extends the lifetime of `match` and allows it to receive messages outside the loop and be returned to the invoker of `findMatchingObject:`.



# Accessor Methods

---

For simple object values, getters and setters can be written in one of the following three ways:

1. Getter retains and autoreleases the value before returning it; setter releases the old value and retains (or copies) the new value.
2. Getter returns the value; setter autoreleases the old value and retains (or copies) the new value.
3. Getter returns the value; setter releases the old value and retains (or copies) the new value.

In technique 1, values returned by the getter are autoreleased within the calling scope:

```
- (NSString*) title
{
    return [[title retain] autorelease];
}

- (void) setTitle: (NSString*) newTitle
{
    if (title != newTitle) {
        [title release];
        title = [newTitle retain]; // or copy depending on your needs
    }
}
```

As with values manufactured by class convenience methods, the returned object is autoreleased in the current scope and thus remains valid if the property value is changed. One issue with this technique is performance. If you expect your getter method to be called frequently, the added cost of retaining and autoreleasing the object may not be worth the performance cost.

Technique 2 also uses an autorelease technique, but this time does so in the setter method:

```
- (NSString*) title
{
    return title;
}

- (void) setTitle: (NSString*) newTitle
{
    [title autorelease];
    title = [newTitle retain];
}
```

The performance of technique 2 is significantly better than technique 1 in situations where the getter is called much more often than the setter.

Technique 3 avoids the use of autorelease altogether:

```
- (NSString*) title
{
    return title;
}

- (void) setTitle: (NSString*) newTitle
{
    if (newTitle != title) {
        [title release];
        title = [newTitle retain];
    }
}
```

The approach used by technique 3 is good for frequently called getter and setter methods. It is also good for objects that do not want to extend the lifetime of their values, such as collection classes. However, because of the potential dangers of invalidating objects prematurely, use of this technique should be used sparingly and well documented.

## Value Objects and Copying

---

It is common practice in Objective-C code to copy value objects—objects that represent attributes. C-type variables can usually be substituted for value objects, but value objects have the advantage of encapsulating convenient utilities for common manipulations. For example, `NSString` objects are used instead of character pointers because they encapsulate encoding and storage. Despite `NSString` functionality, the role played by `NSString` objects parallels the role played by character pointers.

When value objects are passed as method arguments or returned from a method, it is common to use a copy instead of the object itself. For example, consider the following method for assigning a string to an object's `name` instance variable.

```
- (void) setName: (NSString *) aName
{
    [name autorelease];
    name = [aName copy];
}
```

Storing a copy of `aName` has the effect of producing an object that is independent of the original, but has the same contents. Subsequent changes to the copy don't affect the original, and changes to the original don't affect the copy. Similarly, it is common to return a copy of an instance variable instead of the instance variable itself. For example, this method returns a copy of the `name` instance variable:

```
- (NSString *) name
{
    return [[name copy] autorelease];
}
```



# Implementing Object Copy

---

This article describes two approaches to implementing the `NSCopying` protocol's `copyWithZone:` method for the purpose of copying objects.

There are two basic approaches to creating copies by implementing the `NSCopying` protocol's `copyWithZone:` method. You can use `alloc` and `init...`, or you can use `NSCopyObject`. To choose the one that is right for your class, you need to consider the following questions:

- Do I need a deep or shallow copy?
- Do I inherit `NSCopying` behavior from my superclass?

These are described in the following sections.

## Deep Versus Shallow Copies

---

Generally, copying an object involves creating a new instance and initializing it with the values in the original object. Copying the values for non-pointer instance variables, such as booleans, integers, and floating points, is straightforward. When copying pointer instance variables there are two approaches. One approach, called a shallow copy, copies the pointer value from the original object into the copy. Thus, the original and the copy share referenced data. The other approach, called a deep copy, duplicates the data referenced by the pointer and assigns it to the copy's instance variable.

The implementation of an instance variable's set method should reflect the kind of copying you need to use. You should deeply copy the instance variable if the corresponding set method copies the new value as in this method:

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue copy];
}
```

You should shallowly copy the instance variable if the corresponding set method retains the new value as illustrated by this method:

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
```

```

        myVariable = [newValue retain];
    }

```

Similarly, you should shallowly copy the instance variable if its set method simply assigns the new value to the instance variable without copying or retaining it as in the following example—although this is typically rare:

```

- (void)setMyVariable:(id)newValue
{
    myVariable = newValue;
}

```

## Independent Copy

---

To produce a copy of an object that is truly independent of the original, the entire object must be deeply copied. Every instance variable must be duplicated. If the instance variables themselves have instance variables, those too must be duplicated, and so on. In many cases, a mixed approach is more useful. Pointer instance variables that can be thought of as data containers are generally deeply copied, while more sophisticated instance variables like delegates are shallowly copied.

```

@interface Product : NSObject <NSCopying>
{
    NSString *productName;
    float price;
    id delegate;
}

@end

```

For example, a `Product` class adopts `NSCopying`. `Product` instances have a name, a price, and a delegate as declared in this interface.

Copying a `Product` instance produces a deep copy of `productName` because it represents a flat data value. On the other hand, the `delegate` instance variable is a more complex object capable of functioning properly for both `Products`. The copy and the original should therefore share the delegate. [Figure 1](#) (page 34) represents the images of a `Product` instance and a copy in memory.

**Figure 1** Copying instance variables both shallowly and deeply

The different pointer values for `productName` illustrate that the original and the copy each have their own `productName` string object. The pointer values for `delegate` are the same, indicating that the two product objects share the same object as their delegate.

## Inheriting NSCopying from the Superclass

---

If the superclass does not implement `NSCopying`, your class's implementation has to copy the instance variables it inherits as well as those declared in your class. Generally, the safest way to do this is by using `alloc`, `init...`, and `set` methods.

On the other hand, if your class inherits `NSCopying` behavior and has declared additional instance variables, you need to implement `copyWithZone:`, too. In this method, invoke the superclass's implementation to copy inherited instance variables and then copy the additional instance variables. How you handle the new instance variables depends on your familiarity with the superclass's implementation. If the superclass used or might have used `NSCopyObject`, you must handle instance variables differently than you would if `alloc` and `init...` were used.

## Using the “alloc, init...” Approach

---

If a class does not inherit `NSCopying` behavior, you should implement `copyWithZone:` using `alloc`, `init...`, and `set` methods. For example, an implementation of `copyWithZone:` for the `Product` class described in “[Independent Copy](#)” (page 34) might be implemented in the following way:

```
- (id)copyWithZone:(NSZone *)zone
{
    Product *copy = [[self class] allocWithZone: zone]
        initWithProductName:[self productName]
        price:[self price]];
    [copy setDelegate:[self delegate]];

    return copy;
}
```

Because implementation details associated with inherited instance variables are encapsulated in the superclass, it is generally better to implement `NSCopying` with the `alloc, init...` approach. Doing so uses policy implemented in `set` methods to determine the kind of copying needed of instance variables.

## Using `NSCopyObject()`

---

When a class inherits `NSCopying` behavior, you must consider the possibility that the superclass's implementation uses the `NSCopyObject` function. `NSCopyObject` creates an exact shallow copy of an object by copying instance variable values but not the data they point to. For example, `NSCell`'s implementation of `copyWithZone:` could be defined in the following way:

```
- (id)copyWithZone:(NSZone *)zone
{
    NSCell *cellCopy = NSCopyObject(self, 0, zone);
    /* Assume that other initialization takes place here. */

    cellCopy->image = nil;
    [cellCopy setImage:[self image]];

    return cellCopy;
}
```

In the implementation above, `NSCopyObject` creates an exact shallow copy of the original cell. This behavior is desirable for copying instance variables that are not pointers or are pointers to non-retained data that is shallowly copied. Pointer instance variables for retained objects need additional treatment.

In the `copyWithZone:` example above, `image` is a pointer to a retained object. The policy to retain the image is reflected in the following implementation of the `setImage:` accessor method.

```
- (void)setImage:(NSImage *)anImage
{
    [image autorelease];
    image = [anImage retain];
}
```

Notice that `setImage:` autoreleases `image` before it reassigns it. If the above implementation of `copyWithZone:` had not explicitly set the copy's `image` instance variable to `nil` before invoking `setImage:`, the image referenced by the copy and the original would be released without a corresponding `retain`.

Even though `image` points to the right object, it is conceptually uninitialized. Unlike the instance variables that are created with `alloc` and `init...`, these uninitialized variables are not `nil`-valued. You should explicitly assign initial values to these variables before using them. In this case, `cellCopy`'s `image` instance variable is set to `nil`, then it is set using the `setImage:` method.

The effects of `NSCopyObject` extend to a subclass's implementation. For example, an implementation of `NSSliderCell` could copy a new `titleCell` instance variable in the following way.

```
- (id)copyWithZone:(NSZone *)zone
{
    id cellCopy = [super copyWithZone:zone];
    /* Assume that other initialization takes place here. */

    cellCopy->titleCell = nil;
    [cellCopy setTitleCell:[self titleCell]];

    return cellCopy;
}
```

where it is assumed the super's `copyWithZone:` method does something like this:

```
id copy = [[[self class] allocWithZone: zone] init];
```

The superclass's `copyWithZone:` method is invoked to copy inherited instance variables. When you invoke a superclass's `copyWithZone:` method, assume that new object instance variables are uninitialized if there is any chance that the superclass implementation uses `NSCopyObject`. Explicitly assign a value to them before using them. In this example, `titleCell` is explicitly set to `nil` before `setTitleCell:` is invoked.

The implementation of an object's retain count is another consideration when using `NSCopyObject`. If an object stores its retain count in an instance variable, the implementation of `copyWithZone:` must correctly initialize the copy's retain count. [Figure 2](#) (page 37) illustrates the process.

**Figure 2** Initialization of the reference count during a copy

original	0xf2ae4	copy	0x104074	copy	0x104074
isa	0x8028	isa	0x8028	isa	0x8028
refCount	3	refCount	3	refCount	1
productName	0xf2bd8	productName	0xf2bd8	productName	0xe81f4
price	0.00	price	0.00	price	0.00
delegate	0xe83c8	delegate	0xe83c8	delegate	0xe83c8

The copy produced by  
**NSCopyObject**

The copy after initialized  
instance variables are assigned  
in **copyWithZone:**

The first object in [Figure 2](#) (page 37) represents a Product instance in memory. The value in `refCount` indicates that the instance has been retained three times. The second object is a copy of the Product instance produced with `NSCopyObject`. Its `refCount` value matches the original. The third object represents the copy returned from `copyWithZone:` after `refCount` is correctly initialized. After `copyWithZone:` creates the copy with `NSCopyObject`, it assigns the value 1 to the `refCount` instance variable. The sender of `copyWithZone:` implicitly retains the copy and is responsible for releasing it.

## Copying Mutable Versus Immutable Objects

Where the concept “immutable vs. mutable” applies to an object, `NSCopying` produces immutable copies whether the original is immutable or not. Immutable classes can implement `NSCopying` very efficiently. Since immutable objects don’t change, there is no need to duplicate them. Instead, `NSCopying` can be implemented to retain the original. For example, `copyWithZone:` for an immutable string class can be implemented in the following way.

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

Use the `NSMutableCopying` protocol to make mutable copies of an object. The object itself does not need to be mutable to support mutable copying. The protocol declares the method `mutableCopyWithZone:`. Mutable copying is commonly invoked with the convenience `NSObject` method `mutableCopy`, which invokes `mutableCopyWithZone:` with the default zone.



# Memory Management of Core Foundation Objects in Cocoa

---

A number of Core Foundation and Cocoa instances can simply be type-cast to each other, such as `CFString` and `NSString` objects. This document explains how to manage Core Foundation objects in Cocoa. See [“Object Ownership and Disposal”](#) (page 9) for general information on object ownership.

**Important:** This article describes using Cocoa and Core Foundation in a managed memory environment. The semantics are different if you are using garbage collection—see *Garbage Collection Programming Guide*.

Core Foundation's memory allocation policy is that functions with “Copy” or “Create” in their name return values the caller needs to release; all other functions return values you should not release.

In Cocoa, objects created with “alloc”, “copy”, or “new” functions or methods need to be released by the caller.

The conventions used by both Core Foundation and Cocoa are very similar, and because the allocation/retain/release implementations are compatible, equivalent functions and methods from each environment can be used in an intermixed fashion. So,

```
NSString *str = [[NSString alloc] initWithCharacters: ...];  
...  
[str release];
```

is equivalent to

```
CFStringRef str = CFStringCreateWithCharacters(...);  
...  
CFRelease(str);
```

and

```
NSString *str = (NSString *)CFStringCreateWithCharacters(...);  
...  
[str release];
```

and

```
NSString *str = (NSString *)CFStringCreateWithCharacters(...);  
...  
[str autorelease];
```

As these code samples show, once created, the type-casted objects can be treated as Cocoa or Core Foundation and look “native” in each environment.

It has been observed that a larger percentage of Core Foundation functions tend to be “Create” or “Copy” functions when compared to Cocoa’s “alloc”, “copy”, or “new” functions and methods, so it is important to remember to release or autorelease Core Foundation-created objects if appropriate.

Additional information about working with Core Foundation and Carbon data types can be found in the Interchangeable Data Types section of *Carbon-Cocoa Integration Guide*.



# Using Memory Zones

---

Zones are page-aligned areas of memory that hold the objects and data allocated by an application. Each zone contains a private memory heap, with its own free list and pool of memory pages. The system assigns each application a “default” zone initially and applications can create additional zones later. The use of additional zones has both advantages and disadvantages and should be considered with great care. In most circumstances, using the default zone is faster and more efficient than creating a separate zone.

Because zones maintain their own pool of memory, creating new zones increases the memory footprint of your application. However, this increased memory footprint can yield performance advantages in other areas. For example, allocating a group of related objects in the same zone co-locates those objects in the same area of memory. If a page fault occurs when trying to access one of the objects, loading the page brings in all of the related objects, which could significantly reduce the number of future page faults.

Zones are represented in Cocoa by the opaque data type `NSZone`.

## Creating and Managing Zones

---

Creating a zone for your application is not required since the system creates a default zone for you automatically. In fact, creating zones is generally the exception to the rule and should only be used in situations where the need for better performance or memory efficiency outweighs the overhead of maintaining a zone. Most of your memory allocations will otherwise occur in your application’s default zone.

To create a new zone, you use the `NSCreateZone` function. This function uses the system zone allocation routines to set aside an area of memory for your zone. When you call this function, you specify the initial size of the zone and the amount by which to grow the zone. If you attempt to allocate memory beyond the size of your zone, the system automatically expands the zone by the amount you indicate in the second parameter. Thus, if you allocate memory for a contiguous block of structures, you can grow the memory by the exact size of the structure. The following example allocates an 8K block of memory and specifies that the block should grow by 4K when more space is needed.

```
NSZone* tempZone = NSCreateZone(8192, 4096, YES);
```

The Foundation Kit defines several functions for setting and getting the name of a zone and for finding existing zones. You can use the `NSZoneSetName` function to assign a name to a zone and the `NSZoneName` function to retrieve that name later. The `NSZoneFromPointer` function takes a pointer to a block of memory and returns the memory zone in which the block was allocated.

To destroy a zone completely, you must use the system level function `malloc_destroy_zone`. However, destroying a zone that still contains referenced objects can cause severe problems in your application. Before you destroy a zone, you should make absolutely sure that it does not contain any referenced objects or memory blocks. You should never destroy the default zone assigned to your application.

## Allocating Memory in Zones

---

To allocate Cocoa objects in a zone, use the `allocWithZone:` class method of `NSObject`. This method allocates memory for your object in the zone you specify and returns the object pointer to you. The `alloc` method also allocates memory in a zone, the default zone in that case, and is equivalent to calling `allocWithZone:` with a parameter of `nil`. To release an object, you use the same `release` and `autorelease` methods you would normally use.

**Note:** Although the `allocWithZone:` method lets you specify a zone in which to allocate the object, the provided zone is only a suggestion. The object's implementation may ignore the zone if it chooses to.

In addition to allocating Cocoa objects in zones, you can allocate your own custom data structures using functions defined in Foundation Kit. You might use these functions to allocate memory for a C-type data structure or to optimize the storage implementation of one of your classes. The Foundation Kit function names and behaviors are based on routines from the standard C library but include support for allocating memory in zones.

To allocate a block of memory, use the `NSZoneMalloc` function. This function allocates a memory block of a fixed size and returns a basic pointer for you to use. If you want to allocate a block of contiguous memory, use the `NSZoneCalloc` function instead. You can resize a pointer block using the `NSZoneRealloc` function and deallocate a block using `NSZoneFree`.

# Memory Management of Nib Objects

---

At various points in a Cocoa application’s runtime life, one or more nib files are loaded and the objects they contain are unarchived. Who is responsible for releasing those objects when they are no longer needed?

Let’s begin by briefly reviewing what’s in an nib file. When you open a nib file in Interface Builder, the items you see in the Instances pane of the nib file window are known as top-level objects (see Figure 1). Some of these items, such as File’s Owner and First Responder, have special significance; they function as proxies or aliases that confer a certain status on true objects. For example, the File’s Owner instance represents an object external to the nib file that is considered its owner; it’s usually (but not necessarily) the object that loads the nib file. Other instances in a nib file window are the root objects of object graphs, such as windows, panels, and menus. And yet other instances can be without graph dependencies, such as mediating controllers used for bindings or custom controller proxies used for making target-action and outlet connections. When a nib file is loaded and its objects unarchived, all top-level objects are assigned a reference count of one.

**Figure 1** Top-level objects in a nib file

**Note:** For more on nib files and their contents, see *Resource Programming Guide* and the section on object archives and nib files in “The Core Application Architecture” of *Cocoa Fundamentals Guide*.

The File’s Owner of a nib file is typically responsible for releasing the top-level objects in a nib file as well as any non-object resources created by the objects in the nib. The release of the root object of an object graph sets in motion the release of all dependent objects. The File’s Owner of an application’s main nib file (which contains the application menu and possibly other items) is the global application object `NSApp`. However, when a Cocoa application terminates, top level objects in the main nib do not automatically get `dealloc` messages just because `NSApp` is being deallocated. In other words, even in the main nib file, you have to manage the memory of top-level objects.

The Application Kit offers a couple of features that help to ensure that nib objects are properly released:

- `NSWindow` objects (including panels) have an `isReleasedWhenClosed` attribute, which if set to `YES` instructs the window to release itself (and consequently all dependent objects in its view hierarchy) when it is closed. In Interface Builder, you set this option through the “Release when closed” check box in the Attributes pane of the inspector.
- If the File’s Owner of an nib file is an `NSWindowController` object (the default in document nibs in document-based applications), it automatically disposes of the windows it manages.

So in general, you are responsible for releasing top-level objects in a nib file. But in practice, if your nib file's owner is an instance of `NSWindowController` it releases the top-level object for you. If one of your objects loads the nib itself (and the owner is not an instance of `NSWindowController`), you can define outlets to each top-level object so that at the appropriate time you can release them using those references. If you don't want to have outlets to all top-level objects, you can use the `instantiateNibWithOwner:topLevelObjects:` method of the `NSNib` class to get an array of a nib file's top-level objects.

The issue of responsibility for nib object disposal becomes clearer when you consider the various kinds of applications. Most Cocoa applications are of two kinds: single window applications and document-based applications. In both cases, memory management of nib objects is automatically handled for you to some degree. With single-window applications, objects in the main nib file persist through the runtime life of the application and are released when the application terminates; however, `dealloc` is not guaranteed to be automatically invoked on objects from the main nib file when an application terminates. In document-based applications each document window is managed by an `NSWindowController` object which handles memory management for a document nib file.

Some applications may have a more complex arrangement of nib files and top-level objects. For example, an application could have multiple nib file with multiple window controllers, loadable panels, and inspectors. But in most of these cases, if you use `NSWindowController` objects to manage windows and panels or if you set the "released when closed" window attribute, memory management is largely taken care of. If you decide against using window controllers and do not want to set the "release when closed" attribute, you should explicitly free your nib file's windows and other top-level objects when the window is closed. Also, if your application uses an inspector panel, (after being lazily loaded) the panel should typically persist throughout the lifetime of the application—there is no need to dispose of the inspector and its resources.

# Memory Management Rules

---

This document summarizes the rules for memory management in Objective-C.

This is the fundamental rule:

- You take ownership of an object if you create it using a method whose name begins with “alloc” or “new” or contains “copy” (for example, `alloc`, `newObject`, or `mutableCopy`), or if you send it a `retain` message. You are responsible for relinquishing ownership of objects you own using `release` or `autorelease`. Any other time you receive an object, you must *not* release it.

The following rules derive from the fundamental rule, or cope with edge cases:

- As a corollary of the fundamental rule, if you need to store a received object as a property in an instance variable, you must retain or copy it. (This is not true for weak references, described at [“Weak References to Objects”](#) (page 14), but these are typically rare.)
- A received object is normally guaranteed to remain valid within the method it was received in (exceptions include multithreaded applications and some Distributed Objects situations, although you must also take care if you modify the object from which you received the object). That method may also safely return the object to its invoker.

Use `retain` in combination with `release` or `autorelease` when needed to prevent an object from being invalidated as a normal side-effect of a message (see [“Validity of Shared Objects”](#) (page 13)).

- `autorelease` just means “send a `release` message later” (for some definition of later—see [“Autorelease Pools”](#) (page 25)).

For a more complete discussion of memory management in Objective-C see [“Object Ownership and Disposal”](#) (page 9).



# Document Revision History

---

This table describes the changes to *Memory Management Programming Guide for Cocoa*.

Date	Notes
2008-02-08	Corrected a broken link to the "Carbon-Cocoa Integration Guide."
2007-12-11	Corrected typographical errors.
2007-10-31	Updated for Mac OS X v10.5. Corrected minor typographical errors.
2007-06-06	Corrected minor typographical errors.
2007-05-03	Corrected typographical errors.
2007-01-08	Added article on memory management of nib files.
2006-06-28	Added a note about dealloc and application termination.
2006-05-23	Reorganized articles in this document to improve flow; updated "Object Ownership and Disposal."
2006-03-08	Clarified discussion of object ownership and dealloc. Moved discussion of accessor methods to a separate article.
2006-01-10	Corrected typographical errors. Updated title from "Memory Management."
2004-08-31	Changed Related Topics links and updated topic introduction.
2003-06-06	Expanded description of what gets released when an autorelease pool is released to include both explicitly and implicitly autoreleased objects in <a href="#">"Autorelease Pools"</a> (page 25).
2003-06-03	Added link in <a href="#">"Memory Management of Core Foundation Objects in Cocoa"</a> (page 39) to <i>Integrating Carbon and Cocoa in Your Application</i> .
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.

