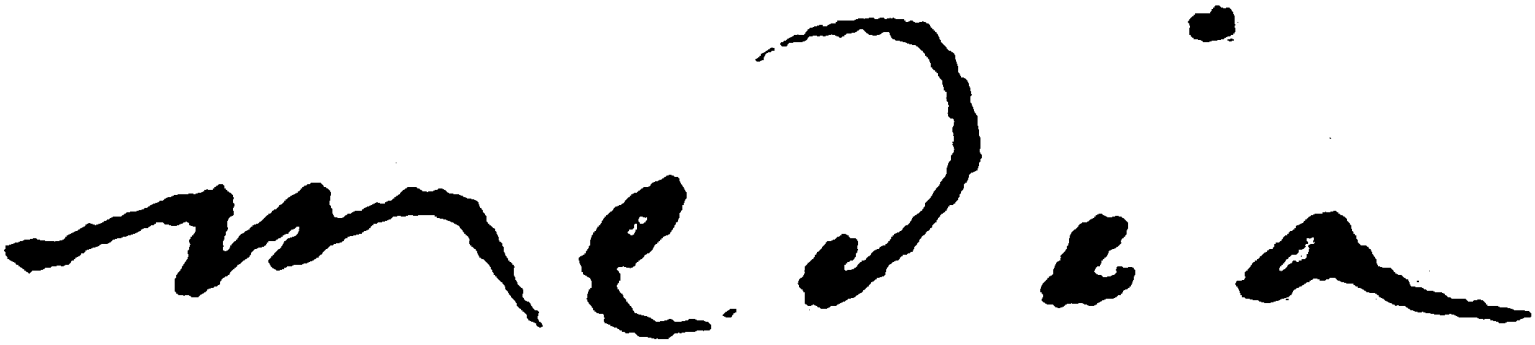
WebObjects Memory Management Survival Guide

by Theresa Ray
Tensor Information Systems

Sponsored by Apple Computer, Inc.
Developer Relations Group
for the Apple Media Program



WebObjects Memory Management

by Theresa Ray, Tensor Information Systems

Regardless of the technology used to create it, memory management techniques can make or break an application. This is especially true for web applications, where response times need to be very fast, and the memory footprint an application requires determines how many instances of that application that one server can manage. The memory management techniques required in a compiled WebObjects application (or any compiled OPENSTEP application) are fairly straightforward, but are usually not well understood by programmers new to this technology. Hopefully, this survival guide will make these issues clear so that developers will be able to create leak-free, crash-free applications.

First, let's briefly discuss the need for compiled code in a WebObjects application. Webscript, the interpreted language which exists in the .wos files, has many advantages. It's quick to write, eliminates the need for static typing, allows rapid changes since there is no compilation step to perform, and the shorthand notation ("dot notation") supported by webscript allows developers to focus on the logic of the application rather than the syntax. However, any interpreted language will, by nature, execute more slowly than it's compiled equivalent. Compiling the code also allows for more robust error checking and enhanced debugging capabilities, as well as speeding execution time by more than an order of magnitude.

When writing webscript, memory management (allocation and deallocation of the memory associated with objects and their instance variables) is taken care of for the developer automatically by the system. However, when using compiled code, the developer must assist in management of object allocation and deallocation.

From object-oriented theory we know that an object is usually a representation in code of a real-world item, such as a Customer. Objects have instance variables which store the information associated with that object, and methods which define the object's behavior. We also know that one of the key benefits of object-oriented programming is that objects can be shared (a Customer can have two Insurance Policies; so instead of duplicating the Customer object associated with each Insurance Policy, the two Insurance Policies are each associated with a single common Customer object). In Objective-C, when an object has nothing associated with it any longer, the memory that this object occupies is automatically freed by the

system. Therefore an object needs to have some sort of counter whose value represents the number of other objects that are associated with this object.

In OPENSTEP & WebObjects programming, this value is maintained by the retain count. The retain count is incremented every time another object creates an association to this object, and is decremented every time another object deletes an association to this object, but not automatically. The object needs to be notified that another object is adding or deleting an association. When the retain count of an object reaches zero, the system AUTOMATICALLY deallocates the memory used by the object. In object-oriented terminology, when object "A" has an association with object "B", we say that object "A" points to object "B", or that object "B" is pointed to by object "A". This terminology will be used throughout the rest of this guide.

For example, consider a customer who has two separate insurance policies. In code, this customer's Insurance Policy objects could be stored in a two element array. Each element of the array points to a different Insurance Policy object. Each of the two Insurance Policy objects are only associated with one item - an array element - so the retain count for each Insurance Policy object is one. The Insurance Policy objects need to have an association identifying the policy holder – the Customer. So the Customer object is pointed to by both Insurance Policy objects. Therefore the Customer object's retain count is two..

If the customer cancels one of the insurance policies, we would want to delete that Insurance Policy object. The Insurance Policy object notifies the Customer object that it will no longer have an association with the Customer object. The Customer object's retain count would be AUTOMATICALLY decremented by one, making it's value one. If the second insurance policy was cancelled, the Customer object's retain count would be decremented again to zero, and it would be automatically deallocated from memory.

You, as programmers, need to notify objects when you create a new association, or when you delete an existing association. Any time you assign an object to an instance variable, you are creating an association and need to think about which mechanism you plan to use to remove the association when you are finished with the object. The importance of this cannot be stressed enough. If you forget to free your associations, the memory required by your application will grow unbearably large. If you try to remove an association which doesn't exist, you can crash the application. Therefore, memory management needs to be properly handled every time.

When you assign an object to an instance variable, you will do so using one of four mechanisms:

1) You can create a new object by sending the class an alloc message (for example, newCust = [Customer alloc];). Sending an alloc message ALWAYS creates a new object (in this case, a new Customer object) and notifies that object that an association is made to it (setting it's retain count will to one).

2) You can copy an existing object by sending it a copy message (for example, insurancePolicies = [policyArray copy];). Sending a copy message creates a new object (in this case, a new array object), initializes it to the same value, and notifies the new object that an association is made to it so that it's retain count will be set to one.

3) You can notify an existing object that you are creating an additional association to it by sending it a retain message (for example, insurancePolicies = [policyArray retain];). Now insurancePolicies points to the same object as policyArray and you don't have to worry about policyArray being finished with the object before insurancePolicy and potentially freeing the memory for that object. The retain message increments the retain count of the object receiving the message by one.

4) You can do nothing. If you have assigned the instance variable to an autoreleased return value from a method (for example, qualifier = [EOQualifier qualifierWithQualifierFormat:...];) and use the value soon (in the next few lines of code or within most methods - such as assigning the qualifier to a fetchSpecification), you usually don't need to retain it. It will not be deallocated from memory until this object will no longer access it anyway.

When you are finished with an object (for example, a variable local to a method no longer needs a value when that method is finished executing) you will notify it that the association is no longer required by one of two mechanisms:

1) You can send that object a release message (for example, [newCust release]). The release message immediately decrements an object's retain count by one (well, you should assume that it's being done immediately).

2) You can send that object an autorelease message (for example, [newCust autorelease]). The autorelease message decrements an object's retain count by one, but not immediately. When, exactly, that the retain count is decremented is highly variable and too detailed to specify here. But it is guaranteed to NOT happen immediately.

When the retain count of an object reaches zero, it is AUTOMATICALLY sent a dealloc message by the system. Never send an object the dealloc method directly

yourself! But you SHOULD implement the dealloc method in your classes to release objects pointed to by your class's instance variables before your object itself is deallocated. For example, a Customer object has an instance variable called insurancePolicies which points to an array. When the Customer object's retain count reaches zero, it should release the object pointed to by insurancePolicies before being freed. An example is shown below:

```
-	(void) dealloc {
[insurancePolicies release];
[super dealloc];\
}
```

An implemented dealloc method should ALWAYS call [super dealloc] at the end of the method so that the superclass can free it's instance variable's objects and so that the object can be truly freed from memory.

Consider the following code:

```
-(Customer *) newCustomer: (NSString *)customerName {
        Customer *newCust;

        newCust=[[Customer alloc]
initWithName:customerName];
        [newCust release];
        return newCust;
        }
```

A Customer object is created (with the "alloc" message), initialized and pointed to by newCust. It's retain count is one (remember, alloc automatically increments the retain count). But the object executing this method doesn't need to keep a reference to this new customer. It's only purpose was to create the new Customer, initialize it to something useful, and to return a pointer to the new Customer object for whatever object sent the message. Therefore, newCust needs to remove it's association with the new Customer object. However, if it sends a "release" message, the new Customer object's retain count will be decremented immediately, and it may be deallocated from memory before the object which sent the "newCustomer:" message has a chance to capture it.

This is the exact situation for which the "autorelease" message was created. Replacing [newCust release] with [newCust autorelease] allows the object which requested the new customer to send a "retain" message to the new Customer object, but still notifies the new Customer object that newCust (and the object executing this method) no longer points it.

Again, what happens if you don't release objects which you create? Your application memory requirements (or "footprint") will grow over time. If your application runs for long periods of time, you will notice that your machine memory is rapidly depleted. The machine starts to swap, and performance degrades significantly. This situation is called a "memory leak".

And again, what happens if you send a "release" or an "autorelease" message to an object whose retain count has already reached zero? This causes a critical error in the application. Most commonly, the application will terminate giving the error message, "Error: message (dealloc) sent to object already freed". (HINT: In order to debug this situation, you can type "defaults write NSGlobalDomain NSZombiesEnabled YES" at a command prompt. This allows the application to continue past the illegal situation so that you can try to determine the cause of the problem.)

Following is a summary of the rules for memory management:

- If an object was assigned to an instance variable using "alloc", "copy" or "retain", the object should be released when it is no longer needed.
- If an object wasn't sent the message, "alloc", "copy", or "retain", it should NOT be released or autoreleased.
- You should autorelease objects used as return values from a method if that method (or the object executing the method) no longer needs to point to that object.
- Assume that any object received as a return value from a method has been autoreleased and will be deallocated from memory. This includes class methods such as [NSString stringWithFormat:] or [NSArray array] as well as instance methods such as [someString substringToIndex:5] and [someArray objectAtIndex:0].
- If you want an autoreleased object to stick around, send it a retain message.
- Objects that you add to containers such as an array, a dictionary or modifications to a mutable string are automatically retained and released by the container object. You are only responsible for retaining and releasing the container object and should NOT try to retain and release the container object's elements.

Let's try some examples. Answer the following questions, then check your answers to see how you've done.

1) In which of the following is the variable myName autoreleased and why?

```
myName=[NSString stringWithFormat:@"%s %s", [[cust
firstName]          cString], [[cust lastName] cString]];
myName=[customer name];
myName=[[NSString alloc] initWithString:@"John Doe"];
myName=[[customer name] retain];
myName=[myName appendString:@" and Mary Doe"];
```

2) Why is the autorelease message used in the following code?

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

3) When would you use copy and when would you use retain in the code for question 2? What is the effect of copy versus retain on the retain count?

4) You have a class called Animal with an instance variable called species. You have a subclass of Animal called Leopard with an instance variable called age. You implement a dealloc method in Leopard to release the object(s) pointed to by what instance variable(s)?

Answers:

1) myName is autoreleased in: myName = [NSString stringWithFormat:@"%s %s", [[cust firstName] cString], [[cust lastName] cString]];. If you look in the documentation for the NSString class, you will find that the method stringWithFormat: is a class method (usually used to create a new object and initialize it at one time - designated by a + instead of a – in the declaration). Class methods do create new objects, but they return it autoreleased. It's retain count will be decremented at some point in the future and it's memory will be freed unless it is sent a retain or copy message.

myName is autoreleased in myName = [customer name];. Name is an instance method returning some object. Since the alloc, copy or retain messages were not sent, the return value is autoreleased.

myName is NOT autoreleased in myName = [[NSString alloc] initWithString:@"John Doe"]; since the "alloc" message was sent.

myName is NOT autoreleased in myName = [[customer name] retain]; since the "retain" message was sent.

Assuming myName was retained previously, myName is NOT autoreleased in myName = [myName appendString:@" and Mary Doe"]; since the appendString message merely modifies the content of myName and does not return a new object.

2) If the release message was sent instead, name could be immediately deallocated. But what if aName was the same as name (the two variables shared a single object)? Since we'd already deallocated the object, when we tried to retain it we could just be retaining nil. The autorelease guarantees that if we're sent the same object that name already points to, that we'll be able to hang on to it.

3) Copy is typically used for objects which have adopted the copy protocol and are only used to contain values. Retain is typically used for business objects which have behavior and where you want to have multiple objects share a common object. By sharing a common object, and changes made to the common object are immediately visible to all objects using the common object. Without shared objects, each copy would have to be updated. When in doubt, use retain.

4) Only the object pointed to by the instance variable age would be released in Leopard's dealloc method. Animal's dealloc method would take care of releasing the object pointed to by species.

About the Author

Theresa Ray is a Senior Software Consultant for Tensor Information Systems in Fort Worth, TX. She has worked as a consultant on WebObjects projects for a wide variety of clients including the U.S. Navy and the United States Postal Service. Her experience spans all versions of WebObjects, from 1.0 to 3.5.

Tensor Information Systems is a systems integrator providing enterprise solutions to its customers. Tensor's employees are experienced in all NeXT/Apple technologies including OPENSTEP, NEXTSTEP and WebObjects. Tensor also provides Oracle consulting and training, as well as systems integration consulting on HP-UX.