10 iPhone Memory Management Tips

Adrian Kosmaczewski

2009-01-28

Memory management in the iPhone is a hot topic. And since tonight I'm talking about it on tonight's monthly meetup of the French-speaking Swiss iPhone Developers group, I might as well share some tips here from my own experience.

I won't go dive through the basics; I think that Scott Stevenson did a great job in his "Learn Objective-C" tutorial at CocoaDevCentral, from where the image below comes. I'm just going to highlight some iPhone-specific issues here and there, and provide some hints on how to solve them.

To begin with, some important background information:

- The iPhone 3G has 128 MB of RAM, but at least half of it might be used by the OS; this might leave as little as 40 MB to your application… but remember: you will get memory warnings even if you only use 3 MB;
- The iPhone does not use garbage collection, even if it uses Objective-C 2.0 (which can use garbage collection on Leopard, nevertheless);
- The basic memory management rule is: for every [alloc | retain | copy] you have to have a [release] somewhere;
- The Objective-C runtime does not allow objects to be instantiated on the stack, but only on the heap; this means that you don't have "automatic objects", nor things like auto_ptr objects to help you manage memory;
- You can use autorelease objects; but watch out! Since they are not released until their pool is released, they can become de facto memory leaks for you…;
- The iPhone does not have a swap file, so forget about virtual memory. When there is no more memory, there is no more memory.

Having said this, here's my list of tips:

- Respond to Memory Warnings
- Avoid Using Autoreleased Objects
- Use Lazy Loading and Reuse
- Avoid UIImage's imageNamed:
- Build Custom Table Cells and Reuse Them Properly
- Override Setters Properly
- Beware of Delegation

- Use Instruments
- Use a Static Analysis Tool
- Use NSZombieEnabled

Respond to Memory Warnings

Whatever you do in your code, please do not forget to respond to memory warnings! I can't stress this much. I have seen application crashes just because the handler methods were not present on the controllers, which means that, even if you do not have anything to clear in your controller, at least do this:

```
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
}
```

And you might as well respond to them on your application delegate, as follows:

```
- (void)applicationDidReceiveMemoryWarning:(UIApplication *)application
{
    [[ImageCache sharedImageCache] removeAllImagesInMemory];
}
```

For the description of the ImageCache class, continue reading ;)

Or finally, as an NSNotification:

```
NSNotificationCenter *center = [NSNotificationCenter defaultCenter];
[center addObserver:self
           selector:@selector(whatever:)
               name:UIApplicationDidReceiveMemoryWarningNotification
             object:nil];
```

Avoid Using Autoreleased Objects

Autoreleasing objects is easy and useful, but on the iPhone you should be careful with it. By default there is an NSAutoreleasePool instance created for you at the beginning of the main() function, but this pool is not cleared up until your application quits! This means that during runtime, your autoreleased objects are de facto memory leaks, since they are retained until the application quits. (please see the comments below; I have experienced better performance when avoiding autoreleased objects, but my understanding of pools is misleading :)

I started getting a better performance from my iPhone apps when I stopped using some methods creating autoreleased objects, for example:

```
// Instead of
NSString *string = [NSString stringWithFormat:@"value = %d", intVariable];

// use
```

```
NSString *string = [[NSString alloc] initWithFormat:@"value = %d", intVariable];
...
[string release];
```

In version 2.0 of the iPhone OS there was also the problem that some "convenience methods" did not work at all; I'm sure you've experienced your application crashing when using NSDictionary's dictionaryWithObjects:forKeys: and then finding out that a replacing that with initWithObjects:forKeys: made your application run just fine. The NDA did not help at the time!

This does not mean that you can't use autoreleased objects; you should use them, for example, when you have factory methods returning objects not owned neither by the factory nor by the client calling it. You can also use autorelease pools in loops, when you need to allocate lots of small objects, but remember to release the pool right afterwards:

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
for (id item in array)
{
    id anotherItem = [item createSomeAutoreleasedObject];
    [anotherItem doSomethingWithIt];
}
[pool release];
```

Remember to always release the pool in the same context where it was created. CocoaDev has an interesting discussion about using NSAutoreleasePools in loops.

Oh, and please, never release an autoreleased object on the iPhone: your application will crash almost instantly.

Use Lazy Loading and Reuse

If your application consists of several different controllers embedded into each other, defer their instantiation until the last possible moment; this means in practical terms that your init method is minimalistic, and that you do more stuff when you need it; the example below is a typical list + detail layout, using a UITableViewController subclass inside a UINavigationController:

```
@interface UITableViewControllerSubclass
{
@private
    NSMutableArray *items;
    DetailController *detailController;
    UINavigationController *navigationController;
}
@end

@implementation UITableViewControllerSubclass
```

```
#pragma mark -
#pragma mark Constructors and destructors

- (id)init
{
    if (self = [self initWithStyle:UITableViewStylePlain])
    {
        // only basic stuff
        items = [[NSMutableArray alloc] initWithCapacity:20];
        navigationController = [[UINavigationController alloc]
                                initWithRootViewController:self];
    }
    return self;
}

- (void)dealloc
{
    [items release];
    [detailController release];
    [navigationController release];
    [super dealloc];
}

// ...

#pragma mark -
#pragma mark UITableViewDelegate methods

- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    Item *item = [items objectAtIndex:indexPath.row];
    if (detailController == nil)
    {
        detailController = [[DetailController alloc] init];
    }
    detailController.item = item;
    [self.navigationController
        pushViewController:detailController
                  animated:YES];
}

// ...

@end
```

As you can see in the example above, not only we are creating a DetailController instance only when needed (that is, when the user taps on an item in the UITableView), but we're reusing it every time the user taps on another cell of the table; this has another benefit: a reduction of object allocation and instantiation, which also helps increasing performance a little bit.

You could also use UIViewController's viewWillAppear: and viewDidDisappear: methods to perform some kind of lazy loading initialization and release, and if you really need to go further, you could use the UITabBarControllerDelegate's tabBarController:didSelectViewController: method to load and unload parts of your application from memory, as you need it.

Avoid UIImage's imageNamed:

Alex Curylo has written an absolutely great article about the problems with UIImage's imageNamed: static method. It seems (and in my tests this appears to be true) that the iPhone OS (versions 2.0 and 2.1 at least) uses an internal cache for images loaded from disk using imageNamed:, and that in cases of low memory this cache is not cleared up completely (this seems to be corrected with version 2.2, though, but I cannot confirm).

Since I have projects that must run on version 2.0 of the iPhone OS, I have created a UIImage category with the following method:

```
@implementation UIImage (AKLoadingExtension)

+ (UIImage *)newImageFromResource:(NSString *)filename
{
    NSString *imageFile = [[NSString alloc] initWithFormat:@"%@/%@",
                           [[NSBundle mainBundle] resourcePath], filename];
    UIImage *image = nil;
    image = [[UIImage alloc] initWithContentsOfFile:imageFile];
    [imageFile release];
    return image;
}

@end
```

The name of the method includes the word "new", to comply with Objective-C's naming guidelines, since the object we're returning to the caller is not autoreleased and has a retain count of 1. The caller is then owner of the UIImage and responsible to release it.

Once I have this UIImage instance, I place it in an image cache with this interface:

```
#import <Foundation/Foundation.h>

@interface ImageCache : NSObject
```

```
{
@private
    NSMutableArray *keyArray;
    NSMutableDictionary *memoryCache;
    NSFileManager *fileManager;
}

+ (ImageCache *)sharedImageCache;

- (UIImage *)imageForKey:(NSString *)key;
- (BOOL)hasImageWithKey:(NSString *)key;
- (void)storeImage:(UIImage *)image withKey:(NSString *)key;
- (BOOL)imageExistsInMemory:(NSString *)key;
- (BOOL)imageExistsInDisk:(NSString *)key;
- (NSUInteger)countImagesInMemory;
- (NSUInteger)countImagesInDisk;
- (void)removeImageWithKey:(NSString *)key;
- (void)removeAllImages;
- (void)removeAllImagesInMemory;
- (void)removeOldImages;

@end
```

Basically, ImageCache can be configured to have a fixed size in memory, and the images that were added first are removed first. It loads the images from the disk as required, keeping a copy in memory, and as suggested by Alex, you can remove them from memory in case of a warning:

```
- (void)applicationDidReceiveMemoryWarning:(UIApplication *)application
{
    [[ImageCache sharedImageCache] removeAllImagesInMemory];
}
```

The complete source code of this ImageCache class, together with some unit tests (thanks to the Google Toolkit for Mac), is available on the Projects section of this blog for you to download and play with.

Build Custom Table Cells and Reuse Them Properly

Remember to always use static NSString identifiers for your cells, which helps the UITableView class to reuse them and reduce memory consumption:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    Item *item = [items objectAtIndex:indexPath.row];
    static NSString *identifier = @"ItemCell";
    ItemCell *cell = (ItemCell *)[tableView
          dequeueReusableCellWithIdentifier:identifier];

    if (cell == nil)
    {
        cell = [[[ItemCell alloc] initWithIdentifier:identifier]
                                                       autorelease];
    }
    cell.item = item;
    return cell;
}
```

I also avoid using NIBs when working with table cells, for performance reasons. I prefer to draw the cells through my own subclasses of UITableViewCell, themselves using overridden setters for their properties, which takes me to the next point.

Override Setters Properly

As I said above, I tend to create my own subclasses of UITableViewCell, providing a simple property through which I change the model class holding the data that the cell is supposed to show. This has the effect of changing all the values of the fields and labels to the values corresponding to those of the model instance.

To do that, I override the setters as follows; for the following class definition…

```
@interface SomeClass
{
@private
    NSArray *items;
    NSString *name;
    id<SomeProtocol> delegate;
}

@property (nonatomic, retain) NSArray *items;
@property (nonatomic, copy) NSString *name;
@property (nonatomic, assign) id<SomeProtocol> delegate;
@end
```

… I use the following implementation:

```
@implementation SomeClass

@synthesize items;
@synthesize name;
@synthesize delegate;
```

```
- (void)dealloc
{
    [items release];
    [name release];
    delegate = nil;
}

#pragma mark -
#pragma mark Overridden setters

- (void)setItems:(NSArray *)obj
{
    if (obj == items)
    {
        return;     // thanks Marco! (see comment #3 below)
    }
    [items release];
    items = nil;
    items = [obj retain];

    if (items != nil)
    {
        // create the internal structure of the cell
        // if not present, and change the widget values
    }
}

- (void)setName:(NSString *)obj
{
    [name release];
    name = nil;
    name = [obj copy];   // I always copy NSStrings!

    if (name != nil)
    {
        // create the internal structure of the cell
        // if not present, and change the widget values
    }
}

- (void)setDelegate:(id<SomeProtocol>)obj
{
    // do not retain! This is an "assign" property
    delegate = obj;

    if (delegate != nil)
    {
        // create the internal structure of the cell
        // if not present, and change the widget values
    }
}

@end
```

Overriding setters properly is important because you might be introducing memory leaks if done wrong. I usually copy all my NSString properties too, as a rule of thumb.

Beware of Delegation

If your code is delegate of some other object which you are about to release, remember to set its delegate property to nil before releasing it; otherwise, the object might “think” that its delegate is still there, and will send a message to an invalid pointer. To see what I’m talking about, consider this code:

```
@interface SomeClass <WidgetDelegate>
{
@private
    Widget *widget;
}
@end

@implementation SomeClass

- (id)init
{
    if (id = [super init])
    {
        widget = [[Widget alloc] init];
        widget.delegate = self;
    }
    return self;
}

- (void)dealloc
{
    // widget might be retained by someone else!
    widget.delegate = nil;
    [widget release];
    [super dealloc];
}

#pragma mark -
```

```
#pragma mark WidgetDelegate methods

- (void)widget:(Widget *)obj callsItsDelegate:(BOOL)value
{
    // and here something happens...
}

@end
```

SomeClass is delegate of Widget. Widget instances might be retained by someone else, which means that even after the release message in the dealloc method, widget might still be alive and call its delegate; if this variable is not nil, widget will send a message to a non-existent object, which will surely crash your application.

Use Instruments

The "Leaks" instrument is your friend, and you should use it after you write the first line of code. Typically, I launch it every time before doing a checkin of some new code. In Xcode, select "Run / Start with Performance Tool / Leaks" and you're done. You can use it in the simulator or on your device.

Use a Static Analysis Tool

Use the LLVM/Clang Static Analyzer tool. This amazing tool will catch naming errors (regarding the Objective-C naming conventions) and some hidden memory leaks, which are particularly nasty when using CoreFoundation libraries (Address Book, sound, CoreGraphics, etc). You can add it to your daily build script, it's very easy to use.

But you must use it. Enough said.

Use NSZombieEnabled

Lou Franco has posted an excellent article about how to use NSZombieEnabled in your development cycle. The idea is to be able to find which messages are being sent to invalid pointers, referencing objects which have been released somewhere in your code. Always remember who's the owner of your objects, and check for existence elsewhere!

And You?

How about you? What are your tips or best practices you usually use for your iPhone apps? Feel free to share them in the form below.

Update, 2009-01-29: I am overwhelmed with the response and traffic that this post has gotten so far! Yesterday evening I had the pleasure of discussing this subject with the guys of the iPhone Developers Facebook group, and I got interesting remarks from Marco Scheurer from Sen:te (including a comment below), which I've added to this post today.

Oh, and by the way, I've uploaded the slides here! They have a Creative Commons license, so feel free to use them if you find them useful.