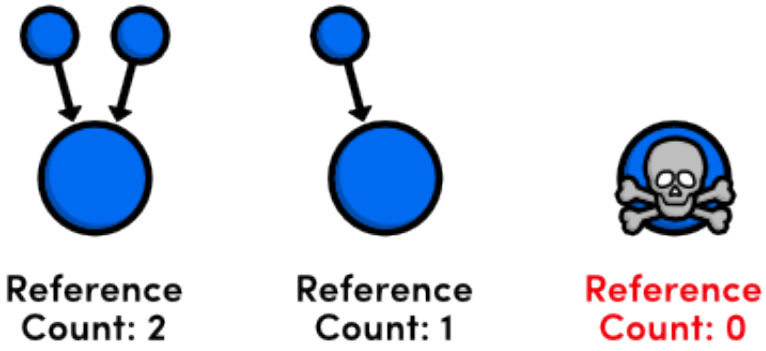
Memory Management

iOS and OS X applications accomplish this through object ownership, which makes sure objects exist as long as they have to, but no longer.

This object-ownership scheme is implemented through a reference-counting system that internally tracks how many owners each object has. When you claim ownership of an object, you increase it's reference count, and when you're done with the object, you decrease its reference count. While its reference count is greater than zero, an object is guaranteed to exist, but as soon as the count reaches zero, the operating system is allowed to destroy it.



In the past, developers manually controlled an object's reference count by calling special memory-management methods defined by the NSObject protocol. This is called Manual Retain Release (MRR). However, Xcode 4.2 introduced Automatic Reference Counting (ARC), which automatically inserts all of these method calls for you. Modern applications should always use ARC, since it's more reliable and lets you focus on your app's features instead of its memory management.

This module explains core reference-counting concepts in the context of MRR, then discusses some of the practical considerations of ARC

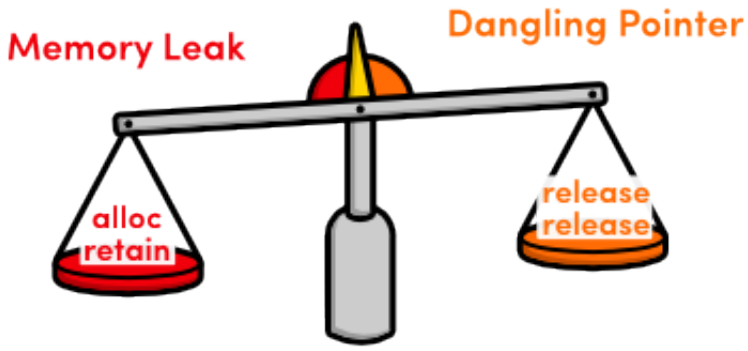
Manual Retain Release

In a Manual Retain Release environment, it's your job to claim and relinquish ownership of every object in your program. You do this by calling special memory-related methods, which are described below

Method	Behavior
alloc	Create an object and claim ownership of it.
retain	Claim ownership of an existing object.
copy	Copy an object and claim ownership of it.
release	Relinquish ownership of an object and destroy it immediately.
autorelease	Relinquish ownership of an object but defer its destruction.

Manually controlling object ownership might seem like a daunting task, but it's actually very easy. All you have to do is claim ownership of any object you need and remember to relinquish ownership when you're done with it. From a practical standpoint, this means that you have to balance every alloc, retain, and copy call with a release or autorelease on the same object.

When you forget to balance these calls, one of two things can happen. If you forget to release an object, its underlying memory is never freed, resulting in a memory leak. Small leaks won't have a visible effect on your program, but if you eat up enough memory, your program will eventually crash. On the other hand, if you try to release an object too many times, you'll have what's called a dangling pointer. When you try to access the dangling pointer, you'll be requesting an invalid memory address, and your program will most likely crash



The alloc Method

We've been using the alloc method to create objects throughout this tutorial. But, it's not just allocating memory for the object, it's also setting its reference count to 1. This makes a lot of sense, since we wouldn't be creating the object if we didn't want to keep it around for at least a little while.

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSMutableArray *inventory = [[NSMutableArray alloc] init];
```

```
        [inventory addObject:@"Honda Civic"];
        NSLog(@"%@", inventory);
    }
    return 0;
}
```

The above code should look familiar. All we're doing is instantiating a mutable array, adding a value, and displaying its contents. From a memory-management perspective, we now own the inventory object, which means it's our responsibility to release it somewhere down the road.

But, since we haven't released it, our program currently has a memory leak

The release Method
The release method relinquishes ownership of an object by decrementing its reference count. So, we can get rid of our memory leak by adding the following line after the NSLog() call in main.m.

[inventory release];

The retain Method

The retain method claims ownership of an existing object. It's like telling the operating system, "Hey! I need that object too, so don't get rid of it!" This is a necessary ability when other objects need to make sure their properties refer to a valid instance.

As an example, we'll use retain to create a strong reference to ourinventory array. Create a new class called CarStore and change its header to the following.

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

@interface CarStore : NSObject

- (NSMutableArray *)inventory;
- (void)setInventory:(NSMutableArray *)newInventory;

@end
```

This manually declares the accessors for a property called inventory. Our first iteration of CarStore.m provides a straightforward implementation of the getter and setter, along with an instance variable to record the object:

```
// CarStore.m
#import "CarStore.h"

@implementation CarStore {
    NSMutableArray *_inventory;
}

- (NSMutableArray *)inventory {
    return _inventory;
}

- (void)setInventory:(NSMutableArray *)newInventory {
    _inventory = newInventory;
}

@end
```

Back in main.m, let's assign our inventory variable to CarStore's inventory property:

```
```

```
// main.m
#import <Foundation/Foundation.h>
#import "CarStore.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSMutableArray *inventory = [[NSMutableArray alloc] init];
        [inventory addObject:@"Honda Civic"];

        CarStore *superstore = [[CarStore alloc] init];
        [superstore setInventory:inventory];
        [inventory release];

        // Do some other stuff...

        // Try to access the property later on (error!)
        NSLog(@"%@", [superstore inventory]);
    }
    return 0;
}
```

The inventory property in the last line is a dangling pointer because the object was already released earlier in main.m. Right now, the superstore object has a weak reference to the array. To turn it into a strong reference, CarStore needs to claim ownership of the array in its setInventory: accessor:

```
// CarStore.m
- (void)setInventory:(NSMutableArray *)newInventory {
    _inventory = [newInventory retain];
}
```

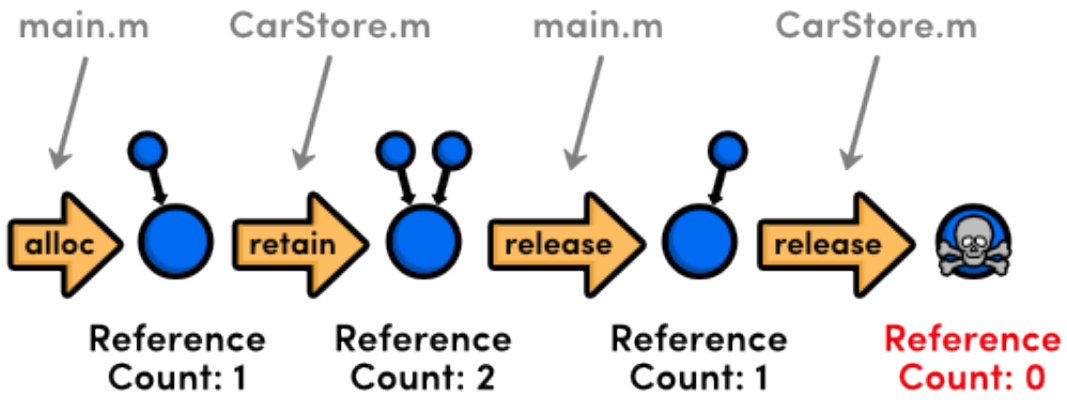
This ensures the inventory object won't be released while superstoreis using it. Notice that the retain method returns the object itself, which lets us perform the retain and assignment in a single line.

Unfortunately, this code creates another problem: the retain call isn't balanced with a release, so we have another memory leak. As soon as we pass another value to setInventory:, we can't access the old value,

which means we can never free it. To fix this, setInventory: needs to call release on the old value:

```
// CarStore.m
- (void)setInventory:(NSMutableArray *)newInventory {
    if (_inventory == newInventory) {
        return;
    }
    NSMutableArray *oldValue = _inventory;
    _inventory = [newInventory retain];
    [oldValue release];
}
```

This is basically what the retain and the strong property attributes do. Obviously, using @property is much more convenient than creating these accessors on our own.



The copy Method

An alternative to retain is the copy method, which creates a brand new instance of the object and increments the reference count on that, leaving the original unaffected. So, if you want to copy

the inventoryarray instead of referring to the mutable one, you can change setInventory: to the following.

```
// CarStore.m
- (void)setInventory:(NSMutableArray *)newInventory {
    if (_inventory == newInventory) {
        return;
    }
    NSMutableArray *oldValue = _inventory;
    _inventory = [newInventory copy];
    [oldValue release];
}
```

You may also recall from The copy Attribute that this has the added perk of freezing mutable collections at the time of assignment. Some classes provide multiple copy methods (much like multiple initmethods), and it's safe to assume that any method starting with copyhas the same behavior.

The autorelease method

Like release, the autorelease method relinquishes ownership of an object, but instead of destroying the object immediately, it defers the actual freeing of memory until later on in the program. This allows you to release objects when you are "supposed" to, while still keeping them around for others to use.

For example, consider a simple factory method that creates and returns a CarStore object:

```
// CarStore.h
+ (CarStore *)carStore;
```

Technically speaking, it's the carStore method's responsibility to release

the object because the caller has no way of knowing that he owns the returned object. So, its implementation should return an autoreleased object, like so:

```
// CarStore.m
+ (CarStore *)carStore {
    CarStore *newStore = [[CarStore alloc] init];
    return [newStore autorelease];
}
```

This relinquishes ownership of the object immediately after creating it, but keeps it in memory long enough for the caller to interact with it. Specifically, it waits until the end of the nearest @autoreleasepool{}block, after which it calls a normal release method. This is why there's always an @autoreleasepool{} surrounding the entire main() function—it makes sure all of the autoreleased objects are destroyed after the program is done executing.

All of those built-in factory methods like NSString's stringWithFormat:and stringWithContentsOfFile: work the exact same way as our carStore method. Before ARC, this was a convenient convention, since it let you create objects without worrying about calling releasesomewhere down the road.

If you change the superstore constructor from alloc/init to the following, you won't have to release it at the end of main().

```
// main.m
CarStore *superstore = [CarStore carStore];
```

In fact, you aren't allowed to release the superstore instance now because you no longer own it—the carStore factory method does. It's very important to avoid explicitly releasing autoreleased objects (otherwise, you'll have a dangling pointer and a crashed program).

The dealloc Method

An object's dealloc method is the opposite of its init method. It's called right before the object is destroyed, giving you a chance to clean up any internal objects. This method is called automatically by the runtime—you should never try to call dealloc yourself.

In an MRR environment, the most common thing you need to do in adealloc method is release objects stored in instance variables. Think about what happens to our current CarStore when an instance is deallocated: its _inventory instance variable, which has been retained by the setter, never has the chance to be released. This is another form of memory leak. To fix this, all we have to do is add a custom deallocto CarStore.m:

```
// CarStore.m
- (void)dealloc {
    [_inventory release];
    [super dealloc];
}
```

Note that you should always call the superclass's dealloc to make sure that all of the instance variables in parent classes are properly released. As a general rule, you want to keep custom dealloc methods as simple as possible, so you shouldn't try to use them for logic that can be handled elsewhere.

Automatic Reference

Counting

Now that you've got your head wrapped around manual memory management, you can forget all about it. Automatic Reference Counting works the exact same way as MRR, but it automatically inserts the appropriate memory-management methods for you. This is a big deal for Objective-C developers, as it lets them focus entirely on whattheir application needs to do rather than how it does it.

ARC takes the human error out of memory management with virtually no downside, so the only reason not to use it is when you're interfacing with a legacy code base (however, ARC is, for the most part, backward compatible with MRR programs). The rest of this module explains the major changes between MRR and ARC