iOS Dev Accelerator Week 7 Day 3

- Memory Management & Balloons
- App Distribution
- Multiple Storyboards
- KVO & Key value coding

Memory Management

Memory Management

- “Application memory management is the process of allocating memory during your program’s runtime, using it, and freeing it when you are done with it”
- Objective-C has two methods of application memory management:
  - Manual retain-release, or MRR, allows you to explicitly manage memory by keeping track of objects you own using reference counting.
  - Automatic Reference Counting, or ARC, uses the same reference counting that MRR does, but it inserts the appropriate memory management method calls for you at compile-time. Swift uses ARC as well.
- **Understanding the specifics of a languages memory management system is critical to effectively learning the language.**

Memory Problems

- There are two general problems that good memory practices will help you avoid:
  - Freeing or overwriting data that is still in use, causing corruption and crashes (dangling pointer)
  - Not freeing data that is no longer in use, which is a memory leak. Leaks needlessly increase the memory footprint of your app, often times to levels that will cause your app to be terminated.
- The main difference between ARC and a garbage collection environment is that ARC does not automatically break retain cycles for you. You must manually manage strong and weak references.

Disabling ARC

- To disable ARC for your entire project, go to the Build Settings of your target, scroll down until you see ‘Apple LLVM 6.0 - Language - Objective-C’ and set Automatic Reference Counting to No
- To disable ARC for an individual file, go to the Build Phases of your target, look in Compile Sources, and click to the right of the name of the file you want to disable ARC. enter in “-fno-objc-arc” without the quotes.

Demo

Reference Counting

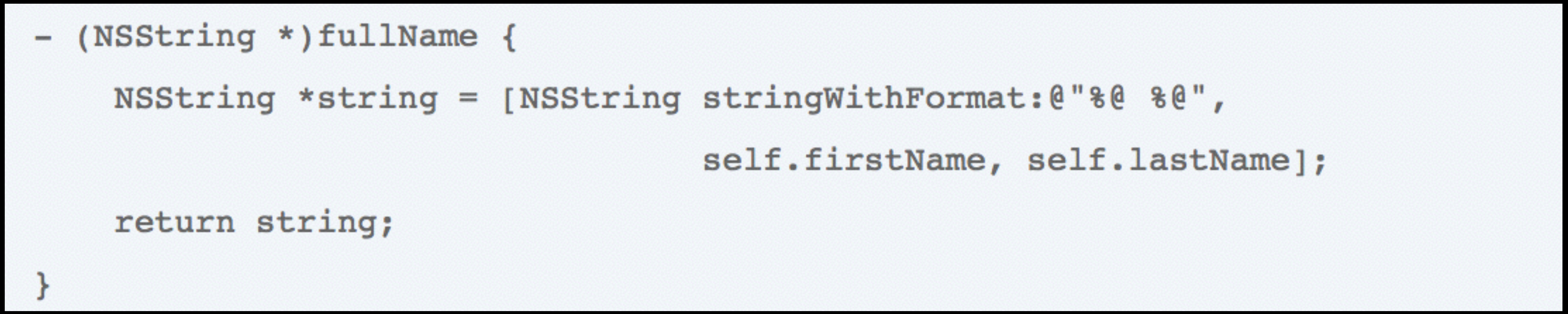
- Objective-C uses reference counting for its memory management system.
- EVERY object has a reference counter that is increased or decreased
- When you want to keep an object alive, you can increase its reference counter, or retain count.
- When you are no longer interested in keeping that object alive, you can decrease the retain count.
- An object with a retain count of 0 is free to be destroyed by the system.

Object Ownership

- Memory management is modeled after the concept of object ownership.
- An object can have many owners.
- As long as an object has one owner, it is kept alive.
- If an object has no owners, it is destroyed by the run time.
- Your retain count is your number of owners

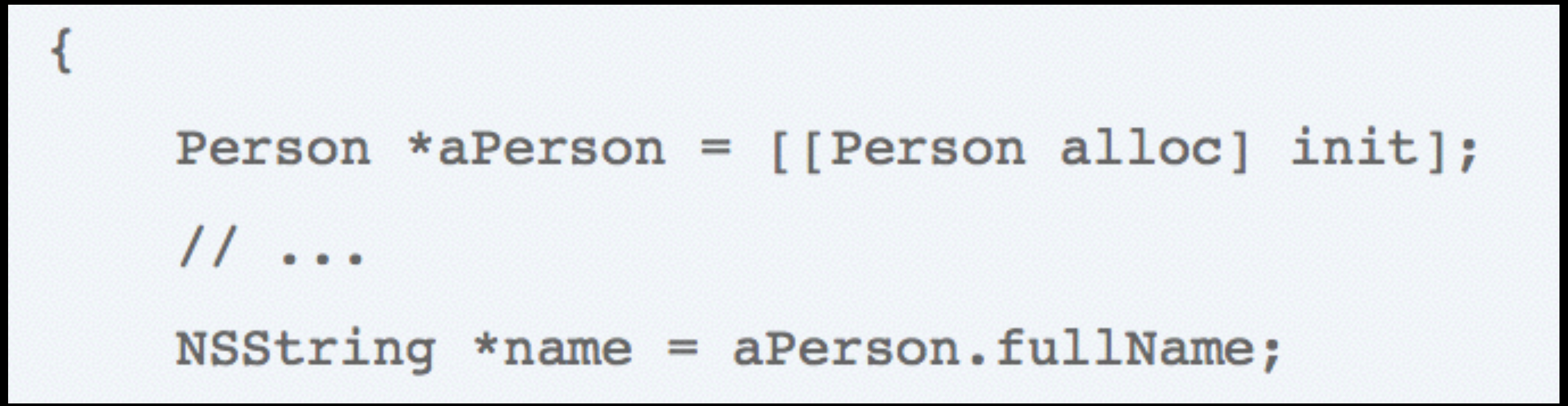
Being the owner

- Anytime you instantiate an object using alloc, new, copy, or mutable copy, the retain count is set to 1. Which means you can say you ‘own’ the object.
- **If a method you called to create an object doesn't contain those words, you don't own that object.**
- You can also ‘own’ any object by calling retain on it, which increases the retain count by 1



```
- (NSString *)fullName {
    NSString *string = [NSString stringWithFormat:@"%@ %@",
                                 self.firstName, self.lastName];
    return string;
}
```

We don’t own string because we didn't use alloc,new,copy

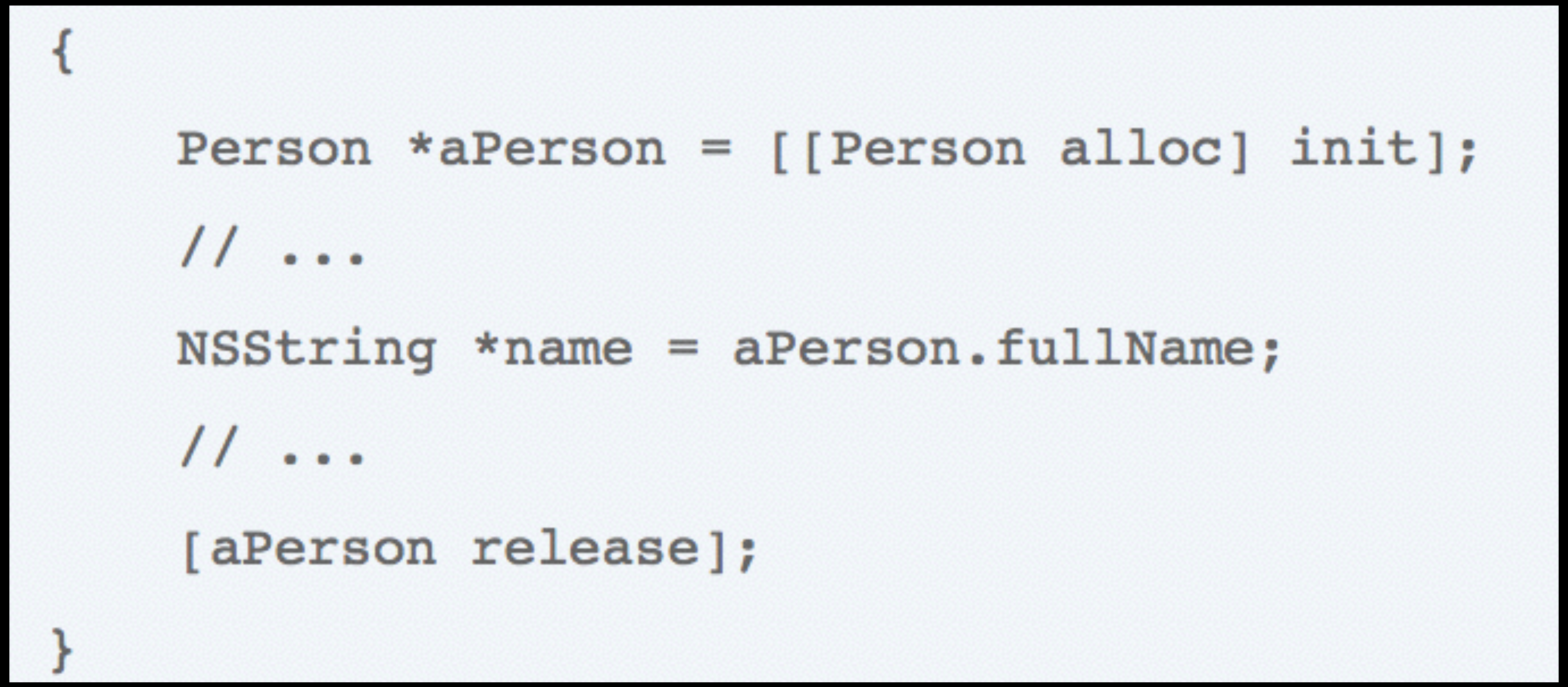


```
{
    Person *aPerson = [[Person alloc] init];
    // ...
    NSString *name = aPerson.fullName;
```

We own aPerson because we used alloc

Relinquishing ownership

- When you are no longer interested in keeping the object alive, you can call release on the object
- This decreases the object’s retain count by 1.
- If the object’s retain count is now 0 after your release, it is free to be destroyed by the system



```
{
    Person *aPerson = [[Person alloc] init];
    // ...
    NSString *name = aPerson.fullName;
    // ...
    [aPerson release];
}
```

Autorelease

- You can use autorelease when you need to send a deferred release message.
- Typically when returning an object from a method.
- You need autorelease because regular release would possibly release the object before you could return it from the method:

```
- (NSString *)fullName {
    NSString *string = [[[NSString alloc] initWithFormat:@"%@ %@",
                                          self.firstName, self.lastName] autorelease];
    return string;
}
```

This gives the object that sent this message a chance to retain this object before it is released

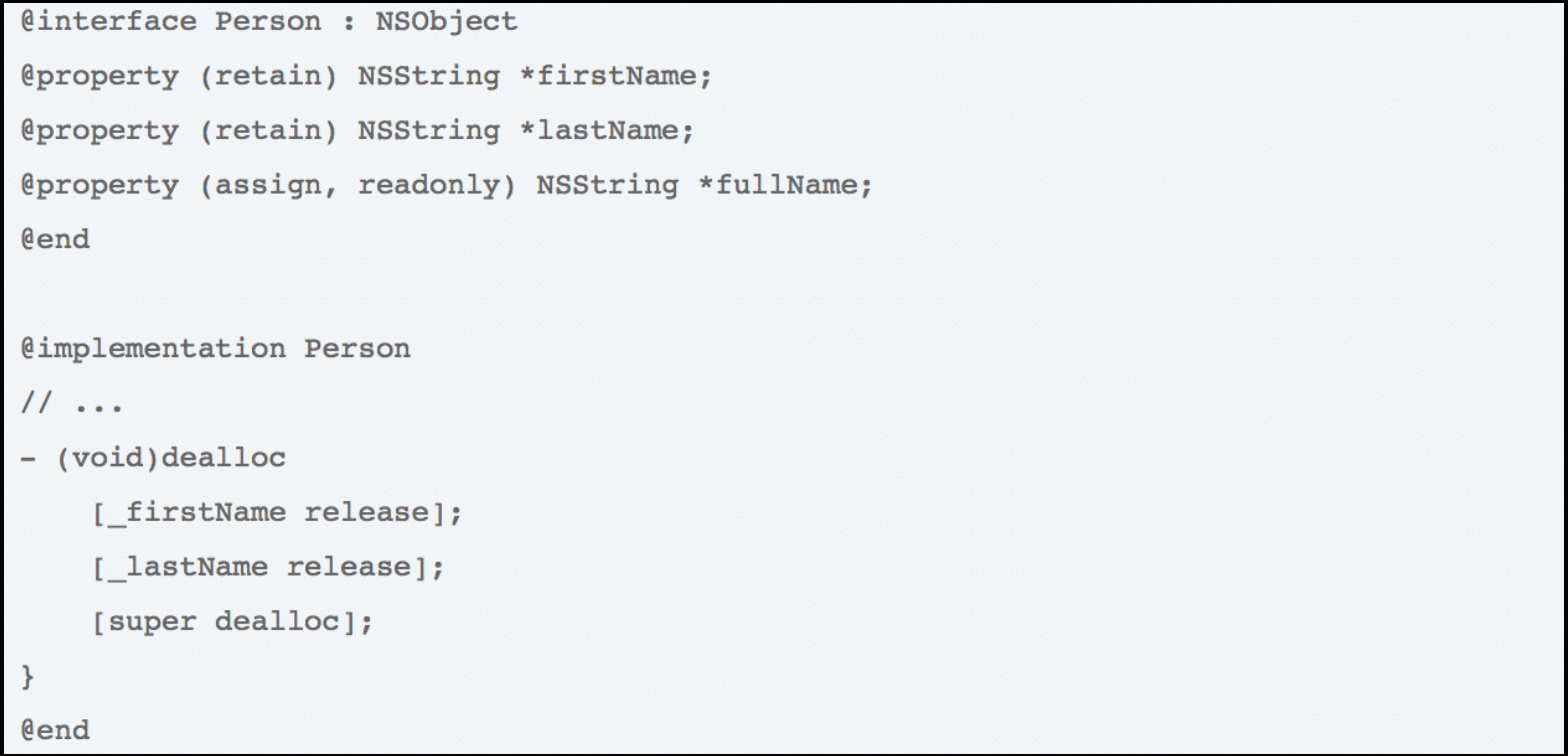
Retain count

- Every object has a retain count property.
- **Never use this property**
- **You should never need to manually access an object’s retain count; The value returned is not accurate. Don’t do it.**

Dealloc

- NSObject provides a method called dealloc.
- Dealloc is invoked automatically when an object has no owners and its memory is reclaimed.
- The role of dealloc is to free the objects own memory, and to dispose of any other objects or resources its holding onto, like properties.
- **You never call dealloc directly, the system calls it for you.**

Dealloc Example



```
@interface Person : NSObject
@property (retain) NSString *firstName;
@property (retain) NSString *lastName;
@property (assign, readonly) NSString *fullName;
@end

@implementation Person
// ...
- (void)dealloc
    [_firstName release];
    [_lastName release];
    [super dealloc];
}
@end
```

Accessor methods and MM

- When your class has a property thats an object, you need to make sure that object isn't released while your class is still using it.
- You need to claim ownership for this. And then you need to relinquish ownership when appropriate.
- Harness the power of accessor methods to make this simple and easy.

Accessor methods and MM

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

getter can just return the object

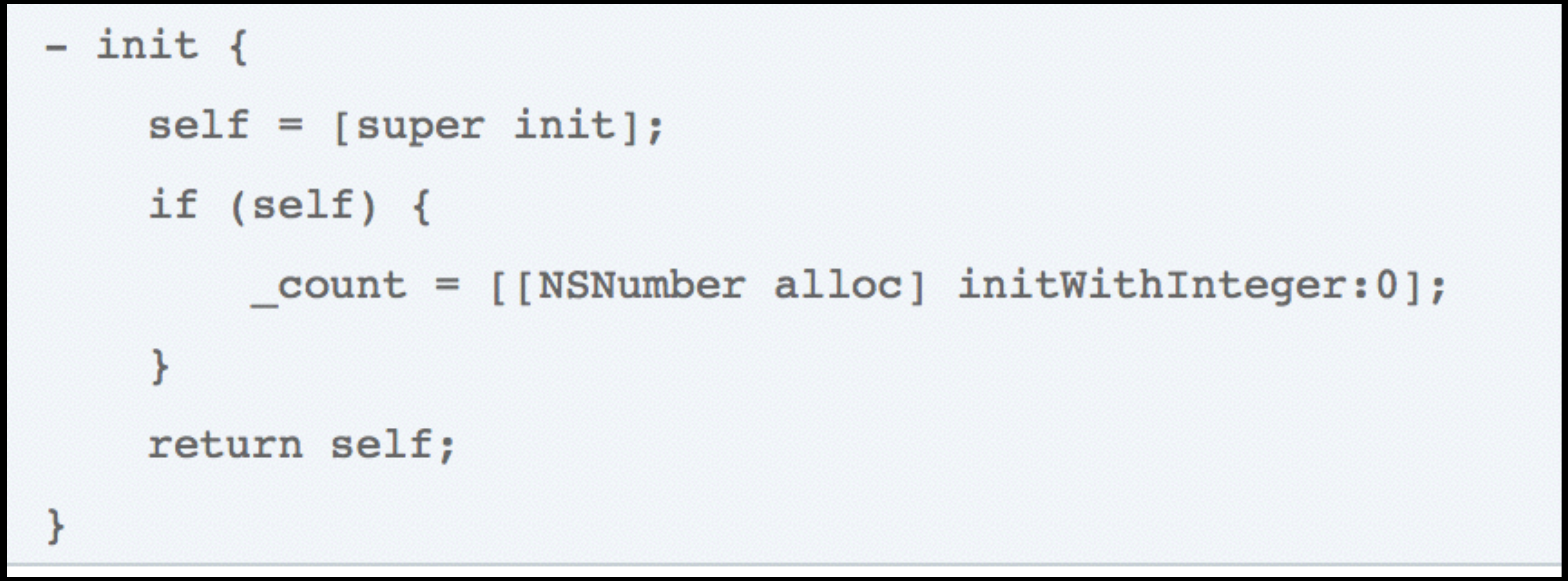
```
- (void)setCount:(NSNumber *)newCount {
    [newCount retain];
    [_count release];
    // Make the new assignment.
    _count = newCount;
}
```

setter must release the old object and retain the new

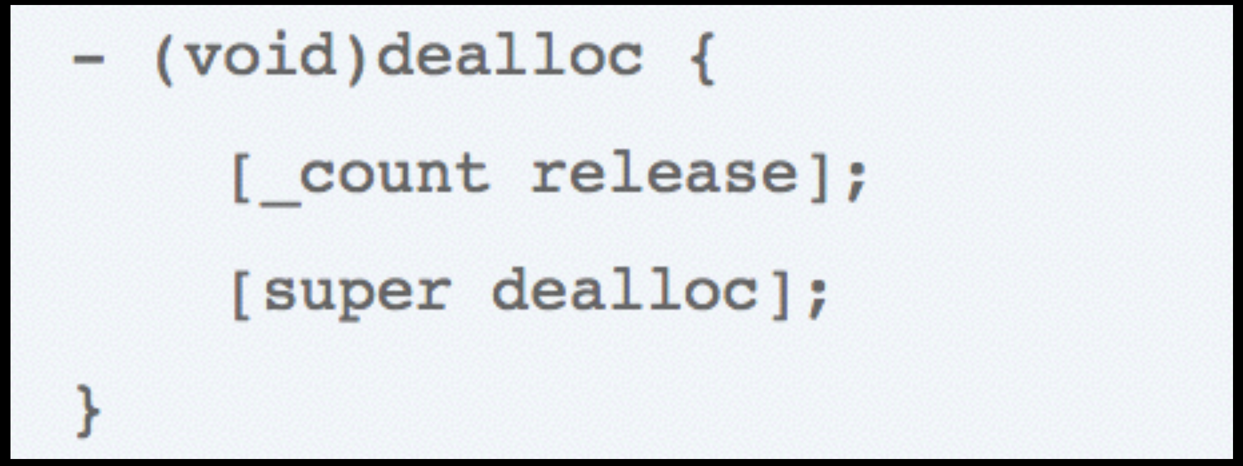
Using the retain attribute on a property does this for you automatically but obviously if you override the setter, you need to do it yourself

When not to use Accessor Methods

- Don’t use the accessor methods in your init methods and dealloc. This eliminates the chance of a bug related to a setter executing or accessing code that hasn't been setup yet.



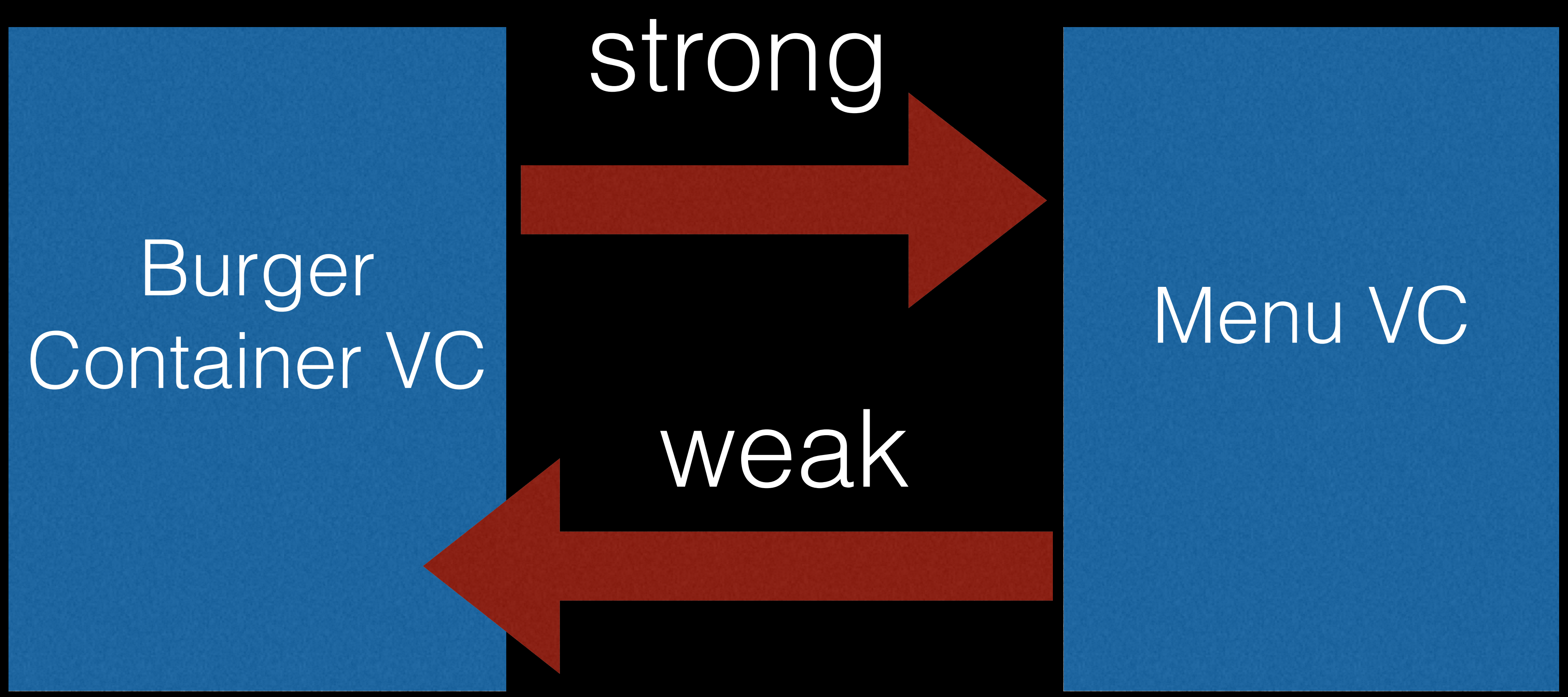
```
- init {
    self = [super init];
    if (self) {
        _count = [[NSNumber alloc] initWithInteger:0];
    }
    return self;
}
```

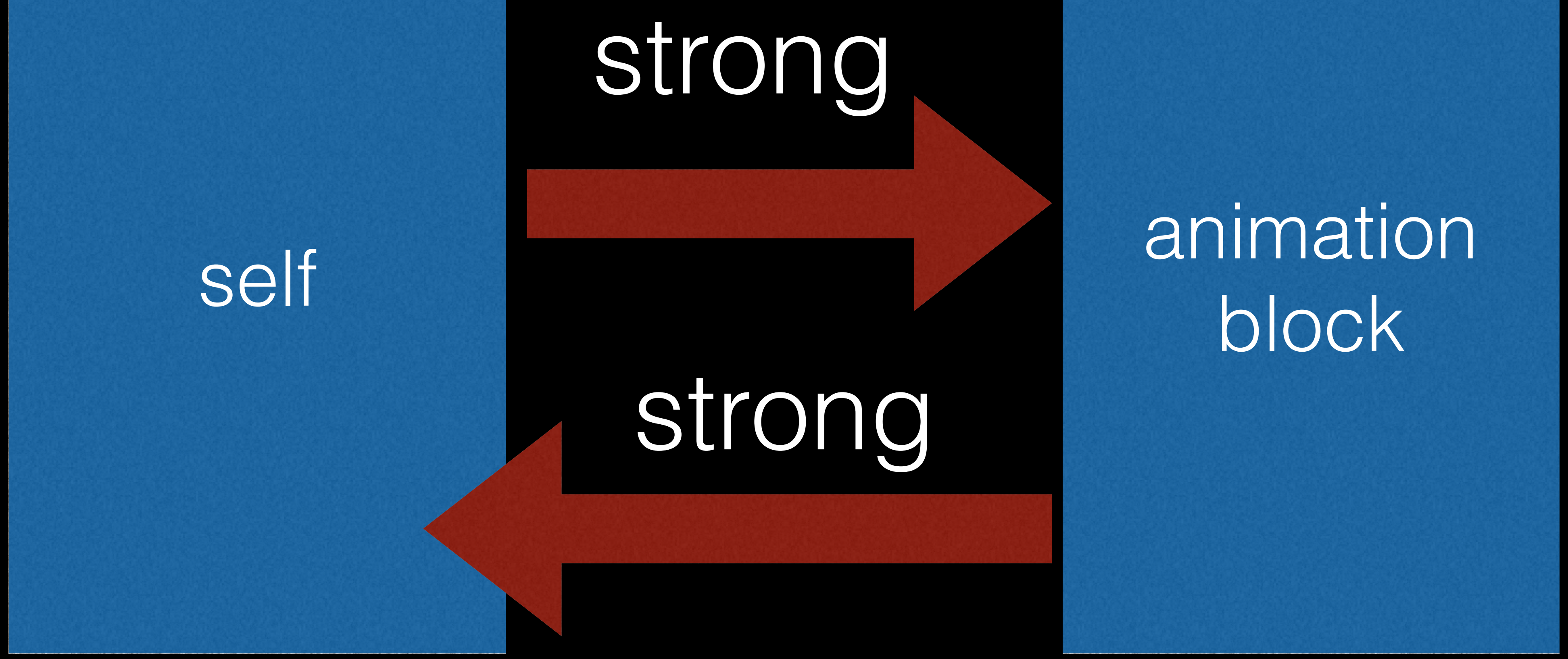


```
- (void)dealloc {
    [_count release];
    [super dealloc];
}
```

Retain Cycles

- Retaining an object creates a strong reference to that object.
- An object cannot be dealloc’d until all of its strong references are released.
- A retain cycle happens when two objects have strong references to each other.


Burger Container VC
strong
Menu VC
weak


self
strong
animation block
strong

Use weak references to avoid retain cycles

- A weak reference is a non-owning relationship where the source object does not retain the object to which it has a reference.
- Cocoa conventions dictate that parent objects should have strong references to their children, and children have weak references to their parents.
- Examples of weak/assign references in Cocoa: data sources, delegates, and notification observers.
- Thats why you need to specifically unregister for notifications when your object is about to be released, since the notification center only keeps **assign** references to objects who sign up for notifications. If you didn't unregistered, the notification center may try to deliver a notification to a released object. CRASH
- **Whenever you can, use weak. Weak properties will automatically be set to nil when the object is set to nil. Assign doesn't do this!**

AutoRelease Pool Blocks

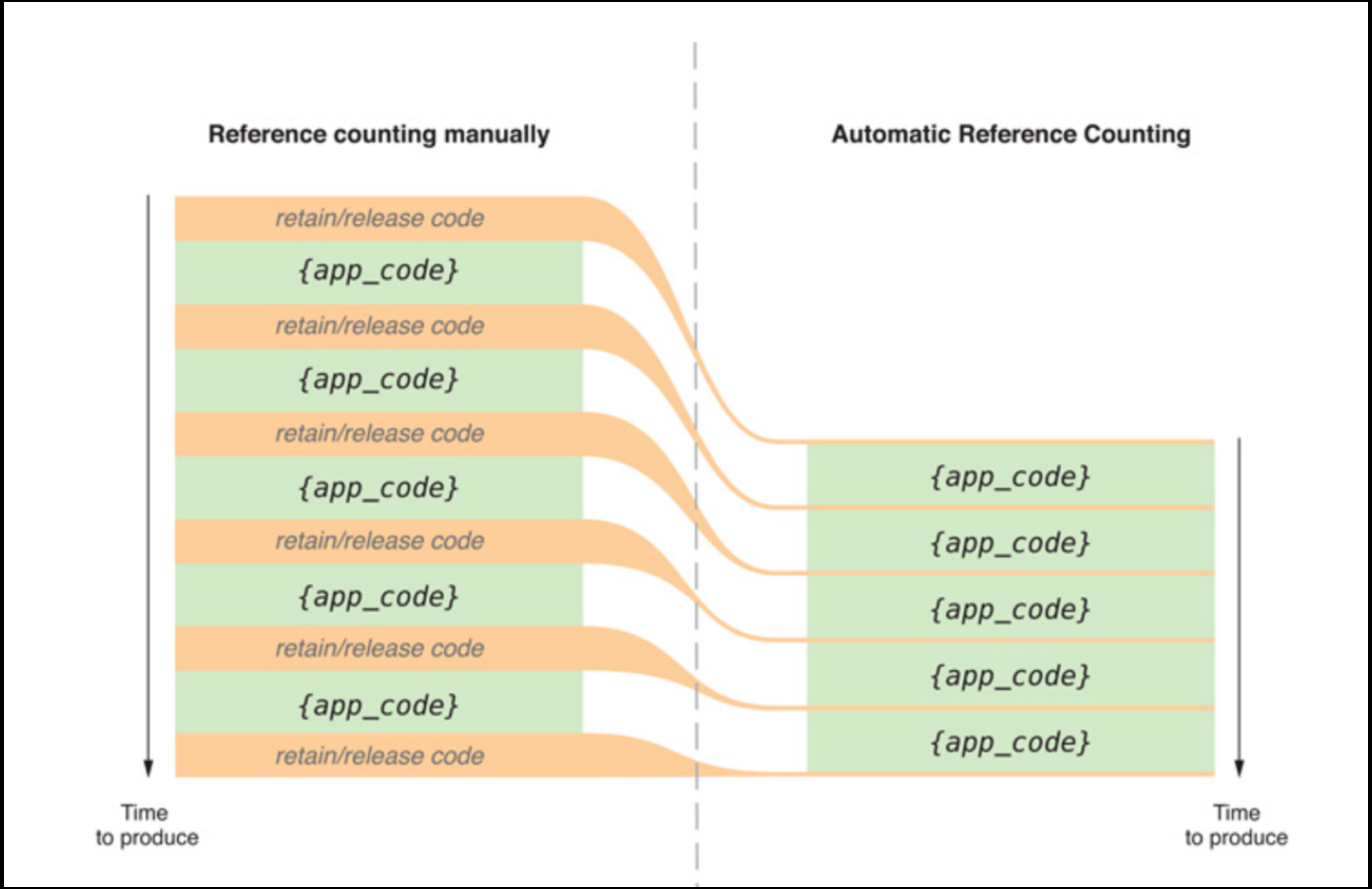
- “Autorelease pool blocks provide a mechanism whereby you can relinquish ownership of an object, but avoid the possibility of it being deallocated immediately (such as when you return an object from a method)”
- You usually don't need to create your own.
- 3 common scenarios when you create one:
  - If you are writing a command-line tool.
  - You write a loop that creates many temporary objects.
  - If you spawn a secondary thread.

AutoRelease Pool Blocks

```
for (NSURL *url in urls) {

    @autoreleasepool {
        NSError *error;
        NSString *fileContents = [NSString stringWithContentsOfURL:url
                                         encoding:NSUTF8StringEncoding error:&error];
        /* Process the string, creating and autoreleasing more objects. */
    }
}
```

ARC


Reference counting manually
Automatic Reference Counting
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
{app_code}
{app_code}
{app_code}
{app_code}
Time to produce
Time to produce

Keeping objects alive while in use

- Objects can be released at any time.
- Like when an object’s parent is released, and the parent was the only owner of the child, the child is released too.
- Collections own the objects they contain. When an object is removed from a collection like an array or dictionary, if the collection was the only owner then the object is released.
- So when you really need to keep an object alive while in use, call retain on it, execute your code, and then release it.

Automatic Reference Counting (ARC)

- ARC enforces new rules: You cannot invoke dealloc, retain,release,retainCount, or autorelease.
- ARC introduced weak and strong property attributes. Strong is the default.
- You can explicitly create weak references with __weak qualifier while declaring a variable.
- Compiler will insert appropriate memory management calls for you.

ARC

```
// The following declaration is a synonym for: @property(retain) MyClass *myObject;
@property(strong) MyClass *myObject;

// The following declaration is similar to "@property(assign) MyClass *myObject;"
// except that if the MyClass instance is deallocated,
// the property value is set to nil instead of remaining as a dangling pointer.
@property(weak) MyClass *myObject;
```

Weak self and blocks

- When referring to self inside a block (or closure), best practice is to create a weak reference in self before the block and use that reference instead of self.
- Since blocks and closures capture strong references to all objects accessed in their bodies, if the object represented by self were to get a strong pointer to the block or closure, then we would have a retain cycle.

Blocks and __weak self

- Be careful accessing self inside a block, or you could create a retain cycle
- Use __weak self and __strong self to avoid this
- You may see this referred to as the @weakify pattern

http://aceontech.com/objc/ios/2014/01/10/weakify-a-more-elegant-solution-to-weakself.html

Blocks and __weak self

```
// Create a weak reference to self
__weak typeof(self)weakSelf = self;

[self.context performBlock:^{
    // Create a strong reference to self, based on the previous weak reference.
    // This prevents a direct strong reference so we don't get
    // into a retain cycle to self.
    // Also prevents self from becoming nil half-way through execution.

    __strong typeof(weakSelf)strongSelf = weakSelf;

    // Do something else

    NSError *error;
    [strongSelf.context save:&error];

    // Do something else
}];
```

App Distribution

Bundle ID

- Your Bundle ID is how both Apple and your device recognizes your app.
- Your app’s Bundle Identifier must be unique to be registered with Apple
- Usually written out in reverse DNS notation (ie com.myCompany.myApp)
- The Bundle ID you have set in your App’s Xcode project MUST match the Bundle ID you have assigned to the App on the iOS Dev Center.
- In Xcode, the Bundle ID is stored in the Info.plist, and is later copied into your app’s bundle when you build.
- In Member center, you create an App ID that matches the app’s bundle ID.
- In iTunes Connect, you enter the Bundle ID to identify your app, after your first version is available on the store, you cannot change your bundle ID EVER AGAIN.

Teams!

- Each Xcode project is associated with a single team.
- If you enroll as an individual, you're considered a one-person team.
- The team account is used to store the certificates, identifiers, and profiles needed to provision your app.
- All iOS apps needs to be provisioned to run on a device.

Team Provisioning Profile

- When you set your team, Xcode ‘may’ attempt to create your code signing identity and development provisioning profile.
- Xcode creates a specialized development provisioning profile called a team provisioning profile that it manages for you.
- A team provisioning profile allows an app to be signed and run by all team members on all their devices.

Provisioning Profile Creation

- Here are the steps Xcode takes when creating your provisioning profile:
    1. Requests your development certificate
    2. Registers the iOS device chosen in the Scheme popup menu
    3. Creates an App ID that matches your app's bundle ID and enables services
    4. Creates a team provisioning profile that contains these assets
    5. Sets your project's code signing build settings accordingly

Version Number & Build String

- The version number of an app is 3 positive integers separated by periods (ex: 1.0.4)
- The first digit is a major release, the 2nd is a minor release, and the third is a maintenance release.
- Build String represents an iteration of the bundle and contain letters and numbers. Change it whenever you distribute a new build of your app for testing.

Code Signing

- “Code Signing your app lets users trust your app has been created by a source known to Apple and that it hasn't been tampered with”
- The **signing identity** is a public-private key pair that Apple issues. The private key is stored in your keychain and used to generate a signature. The certificate contains the public key and identifies you as the owner of the key pair.
- To sign an app, you also need an intermediate certificate, which is automatically installed in your keychain when you install Xcode.
- You use Xcode to create your signing identity and sign your app. Your signing identity is added to your keychain after creation and the corresponding certificate is stored in the member center.
- A **development certificate** identifies you, as a team member, in a development provisioning profile that allows your apps signed by you to launch on devices.
- A **distribution certificate** identifies your team or organization in a distribution provisioning profile and allows you to submit your app to the store.
- You can view your Signing Identifies and Provisioning Profiles in Xcode if you need to troubleshoot them! Everything should match what you see in Member Center.

Submitting your app

1. Create a distribution certificate for your app in Xcode.
2. Create a store distribution provisioning profile on Member Center.
3. Archive and Validate your app in Xcode.
4. Create your App on iTunes Connect
5. Submit your app binary using Xcode or application Loader.
6. Finalize all the info on iTunes Connect and add the build you submitted from Xcode

Demo

Multiple Storyboards

Multiple Storyboards

- When you are building an app with a storyboard, sometimes the storyboard can be become so large and all-encompassing that it becomes unwieldy to use (both hard to find your way around, and also takes forever to load)
- In these cases, it can be very beneficial to break the large storyboard into separate smaller storyboards.

Multiple Storyboards Workflow

1. Add new storyboard's via File->New->File->iOS->Interface->Storyboard
2. Give the storyboard a name that describes what it feature or area of the app it is gong to encompass
3. In code, when you need to transition to a VC in another storyboard, use UIStoryboard's storyboardWithName:bundle: method to get a reference to your next storyboard.
4. Then use instantiateViewControllerWithIdentifier: on that storyboard to get a reference to first VC you want to show from that storyboard.

Demo

Key Value Coding

NSKeyValueCoding

- Protocol that defines mechanisms for indirectly accessing object properties
- Object properties are accessed using the property name rather than directly using the accessor method
- Object properties can be accessed in a consistent manner
- Dynamic and flexible piece of Foundation architecture

NSKeyValueCoding

- Two basic methods for accessing and setting values

```
- (void)setValue:(id)value forKey:(NSString *)key;

- (id)valueForKey:(NSString *)key;
```

- Variations for setting nil values and supporting undefined keys
- Pretty similar to a dictionary

NSKeyValueCoding Example

- Suppose we have a View Controller class with a name property:

```
@interface ViewController ()
@property (strong,nonatomic) NSString *name;
@end
```

- Normally you would set and get the property with the setter and getter or by directly accessing its underlying ivar:

```
self.name = @"Hello";
[self setName:@"Goodbye"];
NSString *myName = _name;
```

NSKeyValueCoding Example

- But with Key-Value Coding, we can set and access the properties of objects sort of like how we interact with dictionaries:

```
[self setValue:@"Brad" forKey:@"name"];
//achieves the same thing as self.name = @"Brad"

NSString *myName = [self valueForKey:@"name"];
//achieves the same thing as NSString *myName = self.name
```

NSKeyValueCoding Example

- We can also use something called a KeyPath, which lets us use dot syntax in our KeyPath string to drill into other object's properties, and even that objects properties, and so forth:

```
NSNumber *nameLength = [self valueForKeyPath:@"name.capitalizedString.length"];
//3 properties deep!

 self.childViewController = [ViewController new];
  [self setValue:@"John" forKeyPath:@"childViewController.name"];
  //2 properties deep!
```

Why is this useful?

- Promotes loose coupling between two objects!
- KVO (next topic)

KVO
(Key Value Observing)

Key-value Observing

- Another protocol. It is considered informal, meaning you wont see it advertised in the docs when objects conform, because NSObject is assumed to conform to it
- Object properties can be observed indirectly by their key
- One object can observe the property of another object, taking action any time the property value changes
- Closely related with the last topic, Key-value Coding
- Is available in Swift, but your classes must inherit from NSObject or be marked as dynamic

KVO workflow

1. The object that you want to observe must be KVO compliant. (done for us, unless its an array — more on this in a bit)
2. The object that will be doing the observing will be set to be the observer by calling addObserver:forKeyPath:options:context on the object you want to observe:
3. observeValueForKeyPath:ofObject:change:context needs to be implemented by the observing class

- The last 2 parameters for step 2 are kind of weird. Typically you pass in the 0 and NULL for them.

Demo

NSKeyValueObserving

- When properties are set, using generated setter, these methods are called before and after the change occurs.
- If you mutate properties outside of the generated setter, you'll need to call these methods manually to be KVO compliant:

```
- (void)willChangeValueForKey:(NSString *)key;
- (void)didChangeValueForKey:(NSString *)key;
```

Demo

Options

- There are 4 options, 2 that you might actually use:
  - New : makes it so the change dictionary contains the new value if applicable
  - Old: same but for the old value

Context

- The context parameter is kind of weird.
- Its an arbitrary value that is used to help you figure out which property you are getting notified by.
- Here is the suggested value for contexts:

```
static void *MyContext = &MyContext;
```

- You can then pass different contexts in when you register to observe and then do some conditional checking on the context when the observation callback is triggered. Meh.

KVO Gotchas

- **Remember to remove observers.**
  - Use `deinit()` in Swift, `dealloc` in ObjC
  - There is no way to test if you are an observer before removing youself, and if you remove yourself when you are an actual observer, an exception will be raised!

http://stackoverflow.com/a/14162363

KVO and arrays

- KVO and arrays are a special case.
- There are just a few more steps you need to take in order to get KVO working with arrays
- Arrays are not KVC compliant by default, you need manually add methods to them to get it to work, its weird
- Check out this tutorial for more info:

  http://www.appcoda.com/understanding-key-value-observing-coding/