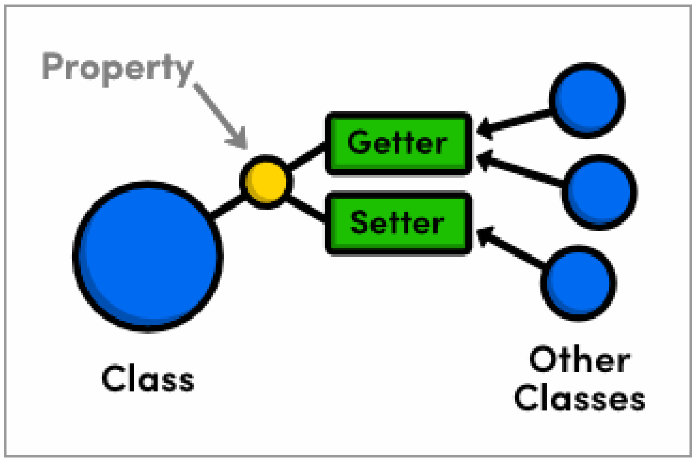
- RyPress
- Tutorials
- Sponsors
- About
- Contact

‹ Back to *Ry's Objective-C Tutorial*

Properties

An object's properties let other objects inspect or change its state. But, in a well-designed object-oriented program, it's not possible to directly access the internal state of an object. Instead, **accessor methods** (getters and setters) are used as an abstraction for interacting with the object's underlying data.



Interacting with a property via accessor methods

The goal of the `@property` directive is to make it easy to create and configure properties by automatically generating these accessor methods. It allows you to specify the behavior of a public property on a semantic level, and it takes care of the implementation details for you.

This module surveys the various attributes that let you alter getter and setter behavior. Some of these attributes determine how properties handle their underlying memory, so this module also serves as a practical introduction to memory management in Objective-C. For a more detailed discussion, please refer to Memory Management.

The @property Directive

First, let's take at what's going on under the hood when we use the `@property` directive. Consider the following interface for a simple `Car` class and it's corresponding implementation.

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

@interface Car : NSObject

@property BOOL running;

@end
```

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

@implementation Car

@synthesize running = _running;    // Optional for Xcode 4.4+

@end
```

The compiler generates a getter and a setter for the `running` property. The default naming convention is to use the property itself as the getter, prefix it with `set` for the setter, and prefix it with an underscore for the instance variable, like so:

```
- (BOOL)running {
    return _running;
}
- (void)setRunning:(BOOL)newValue {
    _running = newValue;
}
```

After declaring the property with the `@property` directive, you can call these methods as if they were included in your class's interface and implementation files. You can also override them in `Car.m` to supply custom getter/setters, but this makes the `@synthesize` directive mandatory. However, you should rarely need custom accessors, since `@property` attributes let you do this on an abstract level.

Properties accessed via dot-notation get translated to the above accessor methods behind the scenes, so the following `honda.running` code actually calls `setRunning:` when you assign a value to it and the `running` method when you read a value from it:

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *honda = [[Car alloc] init];
        honda.running = YES;                // [honda setRunning:YES]
        NSLog(@"%d", honda.running);        // [honda running]
    }
    return 0;
}
```

To change the behavior of the generated accessors, you can specify attributes in parentheses after the `@property` directive. The rest of this module introduces the available attributes.

The getter= and setter= Attributes

If you don't like `@property`'s default naming conventions, you can change the getter/setter method names with the `getter=` and `setter=` attributes. A common use case for this is Boolean properties, whose getters are conventionally prefixed with `is`. Try changing the property declaration in `Car.h` to the following.

```
@property (getter=isRunning) BOOL running;
```

The generated accessors are now called `isRunning` and `setRunning`. Note that the public property is still called running, and this is what you should use for dot-notation:

```
Car *honda = [[Car alloc] init];
honda.running = YES;                // [honda setRunning:YES]
NSLog(@"%d", honda.running);        // [honda isRunning]
NSLog(@"%d", [honda running]);      // Error: method no longer exists
```

These are the only attributes that take an argument (the accessor method name)—all of the others are Boolean flags.

The readonly Attribute

The `readonly` attribute is an easy way to make a property read-only. It omits the setter method and prevents assignment via dot-notation, but the getter is unaffected. As an example, let's change our `Car` interface to the following. Notice how you can specify multiple attributes by separating them with a comma.

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

@interface Car : NSObject

@property (getter=isRunning, readonly) BOOL running;

- (void)startEngine;
- (void)stopEngine;

@end
```

Instead of letting other objects change the `running` property, we'll set it internally via the `startEngine` and `stopEngine` methods. The corresponding implementation can be found below.

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

```
@implementation Car

- (void)startEngine {
    _running = YES;
}
- (void)stopEngine {
    _running = NO;
}

@end
```

Remember that `@property` also generates an instance variable for us, which is why we can access `_running` without declaring it anywhere (we can't use `self.running` here because the property is read-only). Let's test this new `Car` class by adding the following snippet to `main.m`.

```
Car *honda = [[Car alloc] init];
[honda startEngine];
NSLog(@"Running: %d", honda.running);
honda.running = NO;                         // Error: read-only property
```

Up until this point, properties have really just been convenient shortcuts that let us avoid writing boilerplate getter and setter methods. This will not be the case for the remaining attributes, which significantly alter the behavior of their properties. They also only apply to properties that store Objective-C objects (opposed to primitive C data types).

The nonatomic Attribute

Atomicity has to do with how properties behave in a threaded environment. When you have more than one thread, it's possible for the setter and the getter to be called at the same time. This means that the getter/setter can be interrupted by another operation, possibly resulting in corrupted data.

Atomic properties lock the underlying object to prevent this from happening, guaranteeing that the get or set operation is working with a complete value. However, it's important to understand that this is only one aspect of thread-safety—

using atomic properties does not necessarily mean that your code is thread-safe.

Properties declared with `@property` are atomic by default, and this does incur some overhead. So, if you're not in a multi-threaded environment (or you're implementing your own thread-safety), you'll want to override this behavior with the `nonatomic` attribute, like so:

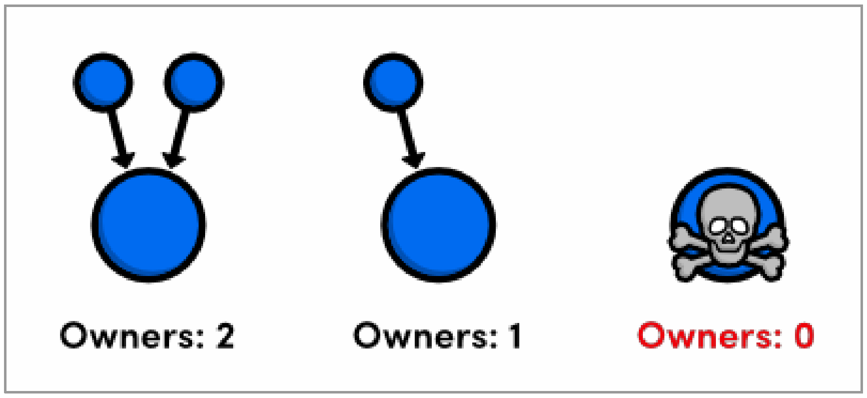
```
@property (nonatomic) NSString *model;
```

There is also a small, practical caveat with atomic properties. Accessors for atomic properties must *both* be either generated or user-defined. Only non-atomic properties let you mix-and-match synthesized accessors with custom ones. You can see this by removing `nonatomic` from the above code and adding a custom getter in `Car.m`.

Memory Management

In any OOP language, objects reside in the computer's memory, and—especially on mobile devices—this is a scarce resource. The goal of a memory management system is to make sure that programs don't take up any more space than they need to by creating and destroying objects in an efficient manner.

Many languages accomplish this through garbage collection, but Objective-C uses a more efficient alternative called **object ownership**. When you start interacting with an object, you're said to *own* that object, which means that it's guaranteed to exist as long as you're using it. When you're done with it, you relinquish ownership, and—if the object has no other owners—the operating system destroys the object and frees up the underlying memory.



Destroying an object with no owners

With the advent of Automatic Reference Counting, the compiler manages all of your object ownership automatically. For the most part, this means that you'll never to worry about how the memory management system actually works. But, you do have to understand the `strong`, `weak` and `copy` attributes of `@property`, since they tell the compiler what kind of relationship objects should have.

The strong Attribute

The `strong` attribute creates an owning relationship to whatever object is assigned to the property. This is the implicit behavior for all object properties, which is a safe default because it makes sure the value exists as long as it's assigned to the property.

Let's take a look at how this works by creating another class called `Person`. It's interface just declares a `name` property:

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

@interface Person : NSObject

@property (nonatomic) NSString *name;

@end
```

The implementation is shown below. It uses the default accessors generated by `@property`. It also overrides `NSObject`'s `description` method, which returns the string representation of the object.

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

@implementation Person

- (NSString *)description {
    return self.name;
}

@end
```

Next, let's add a `Person` property to the `Car` class. Change `Car.h` to the following.

```
// Car.h
#import <Foundation/Foundation.h>
#import "Person.h"

@interface Car : NSObject

@property (nonatomic) NSString *model;
@property (nonatomic, strong) Person *driver;

@end
```

Then, consider the following iteration of `main.m`:

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *john = [[Person alloc] init];
        john.name = @"John";

        Car *honda = [[Car alloc] init];
        honda.model = @"Honda Civic";
        honda.driver = john;

        NSLog(@"%@ is driving the %@", honda.driver, honda.model);
    }
    return 0;
}
```

Since `driver` is a strong relationship, the `honda` object takes ownership of `john`. This ensures that it will be valid as long as `honda` needs it.

The weak Attribute

Most of the time, the `strong` attribute is intuitively what you want for object properties. However, strong references pose a problem if, for example, we need a reference from `driver` back to the `Car` object he's driving. First, let's add a `car` property to `Person.h`:

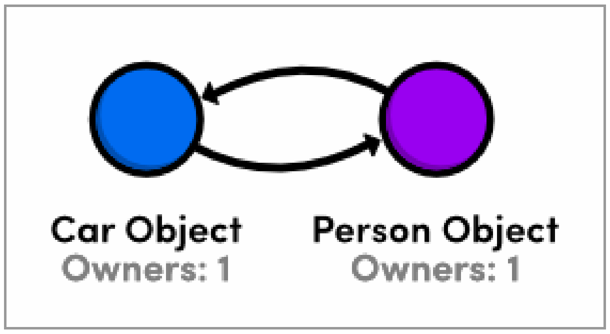
```
// Person.h
#import <Foundation/Foundation.h>

@class Car;

@interface Person : NSObject

@property (nonatomic) NSString *name;
@property (nonatomic, strong) Car *car;

@end
```

The `@class Car` line is a forward declaration of the `Car` class. It's like telling the compiler, "Trust me, the `Car` class exists, so don't try to find it right now." We have to do this instead of our usual `#import` statement because `Car` also imports `Person.h`, and we would have an endless loop of imports. (Compilers don't like endless loops.)

Next, add the following line to `main.m` right after the `honda.driver` assignment:

```
honda.driver = john;
john.car = honda;       // Add this line
```

We now have an owning relationship from `honda` to `john` and another owning relationship from `john` to `honda`. This means that both objects will *always* be owned by another object, so the memory management system won't be able to destroy them even if they're no longer needed.

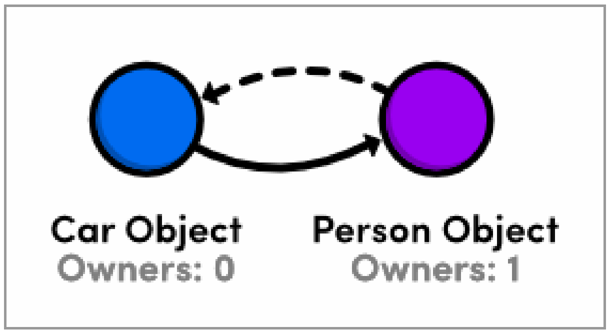


A retain cycle between the `Car` *and* `Person` *classes*

This is called a **retain cycle**, which is a form of memory leak, and memory leaks are bad. Fortunately, it's very easy to fix this problem—just tell one of the properties to maintain a **weak reference** to the other object. In `Person.h`, change the `car` declaration to the following:

```
@property (nonatomic, weak) Car *car;
```

The `weak` attribute creates a non-owning relationship to `car`. This allows `john` to have a reference to `honda` while avoiding a retain cycle. But, this also means that there is a possibility that `honda` will be destroyed while `john` still has a reference to it. Should this happen, the `weak` attribute will conveniently set `car` to `nil` in order to avoid a dangling pointer.



A weak reference from the `Person` *class to* `Car`

A common use case for the `weak` attribute is parent-child data structures. By convention, the parent object should maintain a strong reference with it's children, and the children should store a weak reference back to the parent. Weak references are also an inherent part of the delegate design pattern.

The point to take away is that two objects should never have strong references to each other. The `weak` attribute makes it possible to maintain a cyclical relationship without creating a retain cycle.

The copy Attribute

The `copy` attribute is an alternative to `strong`. Instead of taking ownership of the existing object, it creates a copy of whatever you assign to the property, then takes ownership of that. Only objects that conform to the NSCopying protocol can use this attribute.

Properties that represent values (opposed to connections or relationships) are good candidates for copying. For example, developers usually copy `NSString` properties instead of strongly reference them:

```
// Car.h
@property (nonatomic, copy) NSString *model;
```

Now, `Car` will store a brand new instance of whatever value we assign to `model`. If you're working with mutable values, this has the added perk of freezing the object at whatever value it had when it was assigned. This is demonstrated below:

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *honda = [[Car alloc] init];
        NSMutableString *model = [NSMutableString stringWithString:@"Honda Civic"];
        honda.model = model;

        NSLog(@"%@", honda.model);
        [model setString:@"Nissa Versa"];
        NSLog(@"%@", honda.model);            // Still "Honda Civic"
    }
    return 0;
}
```

NSMutableString is a subclass of `NSString` that can be edited in-place. If the `model` property didn't create a copy of the original instance, we would be able to see the altered string (`Nissan Versa`) in the second `NSLog()` output.

Other Attributes

The above `@property` attributes are all you should need for modern Objective-C applications (iOS 5+), but there are a few others that you may encounter in older libraries or documentation.

The retain Attribute

The `retain` attribute is the Manual Retain Release version of `strong`, and it has the exact same effect: claiming ownership of assigned values. You shouldn't use this in an Automatic Reference Counted environment.

The unsafe_unretained Attribute

Properties with the `unsafe_unretained` attribute behave similar to `weak` properties, but they don't automatically set their value to `nil` if the referenced object is destroyed. The only reason you should need to use `unsafe_unretained` is to make your class compatible with code that doesn't support the `weak` property.

The assign Attribute

The `assign` attribute doesn't perform any kind of memory-management call when assigning a new value to the property. This is the default behavior for primitive data types, and it used to be a way to implement weak references before iOS 5. Like `retain`, you shouldn't ever need to explicitly use this in modern applications.

Summary

This module presented the entire selection of `@property` attributes, and we hope that you're feeling relatively comfortable modifying the behavior of generated accessor methods. Remember that the goal of all these attributes is to help you focus on *what* data needs to be recorded by letting the compiler automatically determine *how* it's represented. They are summarized below.

Attribute	Description
`getter=`	Use a custom name for the getter method.
`setter=`	Use a custom name for the setter method.
`readonly`	Don't synthesize a setter method.

`nonatomic`	Don't guarantee the integrity of accessors in a multi-threaded environment. This is more efficient than the default atomic behavior.
`strong`	Create an owning relationship between the property and the assigned value. This is the default for object properties.
`weak`	Create a non-owning relationship between the property and the assigned value. Use this to prevent retain cycles.
`copy`	Create a copy of the assigned value instead of referencing the existing instance.

Now that we've got properties out of the way, we can take an in-depth look at the other half of Objective-C classes: methods. We'll explore everything from the quirks behind their naming conventions to dynamic method calls.

Continue to *Methods* ›