Methods

Methods represent the actions that an object knows how to perform. They're the logical counterpart to properties, which represent an object's data. You can think of methods as functions that are attached to an object; however, they have a very different syntax.

In this module, we'll explore Objective-C's method naming conventions, which can be frustrating for experienced developers coming from C++, Java, Python, and similar languages. We'll also briefly discuss Objective-C's access modifiers (or lack thereof), and we'll learn how to refer to methods using selectors.

Naming Conventions

Objective-C methods are designed to remove all ambiguities from an API. As a result, method names are horribly verbose, but undeniably descriptive. Accomplishing this boils down to three simple rules for naming Objective-C methods:

1. Don't abbreviate anything.
2. Explicitly state parameter names in the method itself.
3. Explicitly describe the return value of the method.

Keep these rules in mind as you read through the following exemplary interface for a `Car` class.

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

@interface Car : NSObject
```

```
// Accessors
- (BOOL)isRunning;
- (void)setRunning:(BOOL)running;
- (NSString *)model;
- (void)setModel:(NSString *)model;

// Calculated values
- (double)maximumSpeed;
- (double)maximumSpeedUsingLocale:(NSLocale *)locale;

// Action methods
- (void)startEngine;
- (void)driveForDistance:(double)theDistance;
- (void)driveFromOrigin:(id)theOrigin toDestination:(id)theDestination;
- (void)turnByAngle:(double)theAngle;
- (void)turnToAngle:(double)theAngle;

// Error handling methods
- (BOOL)loadPassenger:(id)aPassenger error:(NSError **)error;

// Constructor methods
- (id)initWithModel:(NSString *)aModel;
- (id)initWithModel:(NSString *)aModel mileage:(double)theMileage;

// Comparison methods
- (BOOL)isEqualToCar:(Car *)anotherCar;
- (Car *)fasterCar:(Car *)anotherCar;
- (Car *)slowerCar:(Car *)anotherCar;

// Factory methods
+ (Car *)car;
+ (Car *)carWithModel:(NSString *)aModel;
+ (Car *)carWithModel:(NSString *)aModel mileage:(double)theMileage;

// Singleton methods
+ (Car *)sharedCar;

@end
```

Abbreviations

The easiest way to make methods understandable and predictable is to simply avoid abbreviations. Most Objective-C programmer *expect* methods to be written out in full, as this is the convention for all of the standard frameworks, from Foundation to UIKit. This is why the above interface chose `maximumSpeed` over the more concise `maxSpeed`.

Parameters

One of the clearest examples of Objective-C's verbose design philosophy is in the naming conventions for method parameters. Whereas C++, Java, and other Simula-style languages treat a method as a separate entity from its parameters, an Objective-C method name actually contains the names of all its parameters.

For example, to make a `Car` turn by 90 degrees in C++, you would call something like `turn(90)`. But, Objective-C finds this too ambiguous. It's not clear what kind of argument `turn()` should take—it could be the new orientation, or it could be an angle by which to increment your current orientation. Objective-C methods make this explicit by describing the argument with a preposition. The resulting API ensures the method will never be misinterpreted: it's either `turnByAngle:90` or `turnToAngle:90`.

When a method accepts more than one parameter, the name of each argument is also included in the method name. For instance, the above `initWithModel:mileage:` method explicitly labels both the `Model` argument and the `mileage` argument. As we'll see in a moment, this makes for very informative method invocations.

Return Values

You'll also notice that any methods returning a value explicitly state what that value is. Sometimes this is as simple as stating the class of the return type, but other times you'll need to prefix it with a descriptive adjective.

For example, the factory methods start with `car`, which clearly states that the method will return an instance of the `Car` class. The comparison methods `fasterCar:` and `slowerCar:` return the faster/slower of the receiver and the argument, and this is also clearly expressed in the API. It's worth noting that singleton methods should also follow this pattern (e.g., `sharedCar`), since the conventional `instance` method name is ambiguous.

For more information about naming conventions, please visit the official Cocoa

Coding Guidlines.

Calling Methods

As discussed in the Instantiation and Usage section, you invoke a method by placing the object and the desired method in square brackets, separated by a space. Arguments are separated from the method name using a colon:

```
[porsche initWithModel:@"Porsche"];
```

When you have more than one parameter, it comes after the initial argument, following the same pattern. Each parameter is paired with a label, separated from other arguments by a space, and set off by a colon:

```
[porsche initWithModel:@"Porsche" mileage:42000.0];
```

It's a lot easier to see the purpose behind the above naming conventions when you approach it from an invocation perspective. They make method calls read more like a human language than a computer one. For example, compare the following method call from Simula-style languages to Objective-C's version:

```
// Python/Java/C++
porsche.drive("Home", "Airport");

// Objective-C
[porsche driveFromOrigin:@"Home" toDestination:@"Airport"];
```

It might be more to type, but that's why Xcode comes with such a nice auto-completion feature. You'll appreciate the verbosity when you leave your code for a few months and come back to fix a bug. This clarity also makes it much easier to work with third-party libraries and to maintain large code bases.

Nested Method Calls

Nesting method calls is a common pattern in Objective-C programs. It's a natural way to pass the result of one call to another. Conceptually, it's the exact same as chaining methods, but the square-bracket syntax makes them look a little bit different:

```
// JavaScript
Car.alloc().init()

// Objective-C
[[Car alloc] init];
```

First, the `[Car alloc]` method is invoked, then the `init` method is called on its return value.

Protected and Private Methods

There are no protected or private access modifiers for Objective-C methods—they are all public. However, Objective-C does provide alternative organizational paradigms that let you *emulate* these features.

“Private” methods can be created by defining them in a class’s implementation file while omitting them from its interface file. Since other objects (including subclasses) are never supposed to import the implementation, these methods are effectively hidden from everything but the class itself.

In lieu of protected methods, Objective-C provides categories, which are a more general solution for isolating portions of an API. A full example can be found in “Protected” Methods.

Selectors

Selectors are Objective-C’s internal representation of a method name. They let you treat a method as an independent entity, enabling you to separate an action from the object that needs to perform it. This is the basis of the Target-Action design pattern, and it is an integral part of Objective-C’s dynamic typing system.

There are two ways to get the selector for a method name. The `@selector()` directive lets you convert a source-code method name to a selector, and the `NSSelectorFromString()` function lets you convert a string to a selector (the latter is not as efficient). Both of these return a special data type for selectors called `SEL`.

You can use SEL the exact same way as BOOL, int, or any other data type.

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *porsche = [[Car alloc] init];
        porsche.model = @"Porsche 911 Carrera";

        SEL stepOne = NSSelectorFromString(@"startEngine");
        SEL stepTwo = @selector(driveForDistance:);
        SEL stepThree = @selector(turnByAngle:quickly:);

        // This is the same as:
        // [porsche startEngine];
        [porsche performSelector:stepOne];

        // This is the same as:
        // [porsche driveForDistance:[NSNumber numberWithDouble:5.7]];
        [porsche performSelector:stepTwo
                      withObject:[NSNumber numberWithDouble:5.7]];

        if ([porsche respondsToSelector:stepThree]) {
            // This is the same as:
            // [porsche turnByAngle:[NSNumber numberWithDouble:90.0]
            //              quickly:[NSNumber numberWithBool:YES]];
            [porsche performSelector:stepThree
                          withObject:[NSNumber numberWithDouble:90.0]
                          withObject:[NSNumber numberWithBool:YES]];
        }
        NSLog(@"Step one: %@", NSStringFromSelector(stepOne));
    }
    return 0;
}
```

Selectors can be executed on an arbitrary object via performSelector: and related methods. The `withObject:` versions let you pass an argument (or two) to the method, but require those arguments to be objects. If this is too limiting for your needs, please see `NSInvocation` for advanced usage. When you're not sure if the target object defines the method, you should use the `respondsToSelector:` check before trying to perform the selector.

The technical name for a method is the primary method name concatenated with all of its parameter labels, separated by colons. This makes colons an integral aspect of method names, which can be confusing to Objective-C beginners. Their usage can be summed up as follows: *parameterless methods never contain a colon, while methods that take a parameter always end in a colon.*

A sample interface and implementation for the above `Car` class is included below. Notice how we have to use `NSNumber` instead of `double` for the parameter types, since the `performSelector:withObject:` method doesn't let you pass primitive C data types.

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

@interface Car : NSObject

@property (copy) NSString *model;

- (void)startEngine;
- (void)driveForDistance:(NSNumber *)theDistance;
- (void)turnByAngle:(NSNumber *)theAngle
            quickly:(NSNumber *)useParkingBrake;

@end
```

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

@implementation Car

@synthesize model = _model;
```

```
- (void)startEngine {
    NSLog(@"Starting the %@'s engine", _model);
}

- (void)driveForDistance:(NSNumber *)theDistance {
    NSLog(@"The %@ just drove %0.1f miles",
          _model, [theDistance doubleValue]);
}

- (void)turnByAngle:(NSNumber *)theAngle
            quickly:(NSNumber *)useParkingBrake {
    if ([useParkingBrake boolValue]) {
        NSLog(@"The %@ is drifting around the corner!", _model);
    } else {
        NSLog(@"The %@ is making a gentle %0.1f degree turn",
              _model, [theAngle doubleValue]);
    }
}

@end
```

Summary

This module explained the reasoning behind Objective-C's method naming conventions. We also learned that there are no access modifiers for Objective-C methods, and how to use `@selector` to dynamically invoke methods.

Adapting to new conventions can be a frustrating process, and the dramatic syntactic differences between Objective-C and other OOP languages won't make your life any easier. Instead of forcing Objective-C into your existing mental model of the programming universe, it helps to approach it in its own right. Try designing a few simple programs before passing judgement on Objective-C's verbose philosophy.

That covers the basics of object-oriented programming in Objective-C. The rest of this tutorial explores more advanced ways to organize your code. First on the list are protocols, which let you share an API between several classes.

Continue to *Protocols* ›