# Conventions

When you're working with the framework classes, you'll notice that Objective-C code is very easy to read. Class and method names are much more descriptive than you might find with general C code functions or the C Standard Library, and camel case is used for names with multiple words. You should follow the same conventions used by Cocoa and Cocoa Touch when you're writing your own classes to make your code more readable, both for you and for other Objective-C developers that may need to work with your projects, and to keep your codebase consistent.

In addition, many Objective-C and framework features require you to follow strict naming conventions in order for various mechanisms to work correctly. Accessor method names, for example, must follow the conventions in order to work with techniques such as Key-Value Coding (KVC) or Key-Value Observing (KVO).

This chapter covers some of the most common conventions used in Cocoa and Cocoa Touch code, and explains the situations when it's necessary for names to be unique across an entire app project, including its linked frameworks.

## Some Names Must Be Unique Across Your App

Each time you create a new type, symbol, or identifier, you should first consider the scope in which the name must be unique. Sometimes this scope might be the entire application, including its linked frameworks; sometimes the scope is limited just to an enclosing class or even just a block of code.

### Class Names Must Be Unique Across an Entire App

Objective-C classes must be named uniquely not only within the code that you're writing in a project, but also across any frameworks or bundles you might be including. As an example, you should avoid using generic class names like `ViewController` or `TextParser` because it's possible a framework you include in your app may fail to follow conventions and create classes with the same names.

In order to keep class names unique, the convention is to use prefixes on all classes. You'll have noticed that Cocoa and Cocoa Touch class names typically start either with `NS` or `UI`. Two-letter prefixes like these are reserved by Apple for use in framework classes. As you learn more about Cocoa and Cocoa Touch, you'll encounter a variety of other prefixes that relate to specific frameworks:

| Prefix | Framework |
|---|---|
| `NS` | Foundation (OS X and iOS) and Application Kit (OS X) |
| `UI` | UIKit (iOS) |
| `AB` | Address Book |
| `CA` | Core Animation |
| `CI` | Core Image |

Your own classes should use three letter prefixes. These might relate to a combination of your company name and your app name, or even a specific component within your app. As an example, if your company were called Whispering Oak, and you were developing a game called Zebra Surprise, you might choose `WZS` or `WOZ` as your class prefix.

You should also name your classes using a noun that makes it clear what the class represents, like these examples from Cocoa and Cocoa Touch:

| | | | |
|---|---|---|---|
| `NSWindow` | `CAAnimation` | `NSWindowController` | `NSManagedObjectContext` |

If multiple words are needed in a class name, you should use *camel case* by capitalizing the first letter of each subsequent word.

### Method Names Should Be Expressive and Unique Within a Class

Once you've chosen a unique name for a class, the methods that you declare need only be unique within that class. It's common to use the same name as a method in another class, for example, either to override a superclass method, or take advantage of polymorphism. Methods that perform the same task in multiple classes should have the same name, return type and parameter types.

Method names do not have a prefix, and should start with a lowercase letter; camel case is used again for multiple words, like these examples from the `NSString` class:

| | | |
|---|---|---|

| length | characterAtIndex: | lengthOfBytesUsingEncoding: |
|---|---|---|

If a method takes one or more arguments, the name of the method should indicate each parameter:

| substringFromIndex: | writeToURL:atomically:encoding:error: | enumerateSubstringsInRange:options:usingBlock: |
|---|---|---|

The first portion of the method name should indicate the primary intent or result of calling the method. If a method returns a value, for example, the first word normally indicates what will be returned, like the `length`, `character...` and `substring...` methods shown above. Multiple words are used if you need to indicate something important about the return value, as with the `mutableCopy`, `capitalizedString` or `lastPathComponent` methods from the `NSString` class. If a method performs an action, such as writing to disk or enumerating the contents, the first word should indicate that action, as shown by the `write...` and `enumerate...` methods.

If a method includes an *error* pointer parameter to be set if an error occurred, this should be the last parameter to the method. If a method takes a *block*, the block parameter should be the last parameter in order to make any method invocations as readable as possible when specifying a block inline. For the same reason, it's best to avoid methods that take multiple block arguments, wherever possible.

It's also important to aim for clear but concise method names. Clarity doesn't necessarily mean verbosity but brevity doesn't necessarily result in clarity, so it's best to aim for a happy medium:

| | |
|---|---|
| stringAfterFindingAndReplacingAllOccurrencesOfThisString:withThisString: | Too verbose |
| strReplacingStr:str: | Too concise |
| stringByReplacingOccurrencesOfString:withString: | Just right |

You should avoid abbreviating words in method names unless you are sure that the abbreviation is well known across multiple languages and cultures. A list of common abbreviations is given in Acceptable Abbreviations and Acronyms.

### Always Use a Prefix for Method Names in Categories on Framework Classes

When using a category to add methods to an existing framework class, you should include a prefix on the method name to avoid clashes, as described in Avoid Category Method Name Clashes.

## Local Variables Must Be Unique Within The Same Scope

Because Objective-C is a superset of the C language, the C variable scope rules also apply to Objective-C. A local variable name must not clash with any other variables declared within the same scope:

```
- (void)someMethod {
    int interestingNumber = 42;
    ...
    int interestingNumber = 44; // not allowed
}
```

Although the C language does allow you to declare a new local variable with the same name as one declared in the *enclosing* scope, like this:

```
- (void)someMethod {
    int interestingNumber = 42;
    ...
    for (NSNumber *eachNumber in array) {
        int interestingNumber = [eachNumber intValue]; // not advisable
        ...
    }
}
```

it makes the code confusing and less readable, so it's best practice to avoid this wherever possible.

# Some Method Names Must Follow Conventions

In addition to considering uniqueness, it's also essential for a few important method types to follow strict conventions. These conventions are used by some of the underlying mechanisms of Objective-C, the compiler and runtime, in addition to behavior that is required by classes in Cocoa and Cocoa Touch.

## Accessor Method Names Must Follow Conventions

When you use the `@property` syntax to declare properties on an object, as described in Encapsulating Data, the compiler automatically synthesizes the relevant getter and setter methods (unless you indicate otherwise). If you need to provide your own accessor method implementations for any reason, it's important to make sure that you use the right method names for a property in order for your methods to be called through dot syntax, for example.

Unless specified otherwise, a getter method should use the same name as the property. For a property called `firstName`, the accessor method should also be called `firstName`. The exception to this rule is for Boolean properties, for which the getter method should start with `is`. For a property called `paused`, for example, the getter method should be called `isPaused`.

The setter method for a property should use the form `setPropertyName:`. For a property called `firstName`, the setter method should be called `setFirstName:`; for a Boolean property called `paused`, the setter method should be called `setPaused:`.

Although the `@property` syntax allows you to specify different accessor method names, you should only do so for situations like a Boolean property. It's essential to follow the conventions described here, otherwise techniques like Key Value Coding (the ability to get or set a property using `valueForKey:` and `setValue:forKey:`) won't work. For more information on KVC, see *Key-Value Coding Programming Guide*.

## Object Creation Method Names Must Follow Conventions

As you've seen in earlier chapters, there are often multiple ways to create instances of a class. You might use a combination of allocation and initialization, like this:

```
NSMutableArray *array = [[NSMutableArray alloc] init];
```

or use the `new` convenience method as an alternative to calling `alloc` and `init` explicitly:

```
NSMutableArray *array = [NSMutableArray new];
```

Some classes also offer class factory methods:

```
NSMutableArray *array = [NSMutableArray array];
```

Class factory methods should always start with the name of the class (without the prefix) that they create, with the exception of subclasses of classes with existing factory methods. In the case of the `NSArray` class, for example, the factory methods start with `array`. The `NSMutableArray` class doesn't define any of its own class-specific factory methods, so the factory methods for a mutable array still begin with `array`.

There are various memory management rules underpinning Objective-C, which the compiler uses to ensure objects are kept alive as long as necessary. Although you typically don't need to worry too much about these rules, the compiler judges which rule it should follow based on the name of the creation method. Objects created via factory methods are managed slightly differently from objects that are created through traditional allocation and initialization or `new` because of the use of autorelease pool blocks. For more information on autorelease pool blocks and memory management in general, see *Advanced Memory Management Programming Guide*.