# Customizing Existing Classes

Objects should have clearly-defined tasks, such as modeling specific information, displaying visual content or controlling the flow of information. As you've already seen, a class interface defines the ways in which others are expected to interact with an object to help it accomplish those tasks.

Sometimes, you may find that you wish to extend an existing class by adding behavior that is useful only in certain situations. As an example, you might find that your application often needs to display a string of characters in a visual interface. Rather than creating some string-drawing object to use every time you need to display a string, it would make more sense if it was possible to give the `NSString` class itself the ability to draw its own characters on screen.

In situations like this, it doesn't always make sense to add the utility behavior to the original, primary class interface. Drawing abilities are unlikely to be needed the majority of times any string object is used in an application, for example, and in the case of `NSString`, you can't modify the original interface or implementation because it's a framework class.

Furthermore, it might not make sense to subclass the existing class, because you may want your drawing behavior available not only to the original `NSString` class but also any subclasses of that class, like `NSMutableString`. And, although `NSString` is available on both OS X and iOS, the drawing code would need to be different for each platform, so you'd need to use a different subclass on each platform.

Instead, Objective-C allows you to add your own methods to existing classes through categories and class extensions.

## Categories Add Methods to Existing Classes

If you need to add a method to an existing class, perhaps to add functionality to make it easier to do something in your own application, the easiest way is to use a category.

The syntax to declare a *category* uses the `@interface` keyword, just like a standard Objective-C class description, but does not indicate any inheritance from a subclass. Instead, it specifies the name of the category in parentheses, like this:

```
@interface ClassName (CategoryName)


@end
```

A category can be declared for any class, even if you don't have the original implementation source code (such as for standard Cocoa or Cocoa Touch classes). Any methods that you declare in a category will be available to all instances of the original class, as well as any subclasses of the original class. At runtime, there's no difference between a method added by a category and one that is implemented by the original class.

Consider the `XYZPerson` class from the previous chapters, which has properties for a person's first and last name. If you're writing a record-keeping application, you may find that you frequently need to display a list of people by last name, like this:

```
Appleseed, John
Doe, Jane
Smith, Bob
Warwick, Kate
```

Rather than having to write code to generate a suitable `lastName, firstName` string each time you wanted to display it, you could add a category to the `XYZPerson` class, like this:

```
#import "XYZPerson.h"


@interface XYZPerson (XYZPersonNameDisplayAdditions)
- (NSString *)lastNameFirstNameString;
@end
```

In this example, the `XYZPersonNameDisplayAdditions` category declares one additional method to return the necessary string.

A category is usually declared in a separate header file and implemented in a separate source code file. In the case of `XYZPerson`, you might declare the category in a header file called `XYZPerson+XYZPersonNameDisplayAdditions.h`.

Even though any methods added by a category are available to all instances of the class and its subclasses, you'll need to import the category header file in any source code file where you wish to use the additional methods, otherwise you'll run into compiler warnings and errors.

The category implementation might look like this:

```
#import "XYZPerson+XYZPersonNameDisplayAdditions.h"


@implementation XYZPerson (XYZPersonNameDisplayAdditions)
- (NSString *)lastNameFirstNameString {
    return [NSString stringWithFormat:@"%@, %@", self.lastName, self.firstName];
}
@end
```

Once you've declared a category and implemented the methods, you can use those methods from any instance of the class, as if they were part of the original class interface:

```
#import "XYZPerson+XYZPersonNameDisplayAdditions.h"
@implementation SomeObject
- (void)someMethod {
    XYZPerson *person = [[XYZPerson alloc] initWithFirstName:@"John"
                                                    lastName:@"Doe"];
    XYZShoutingPerson *shoutingPerson =
                        [[XYZShoutingPerson alloc] initWithFirstName:@"Monica"
                                                            lastName:@"Robinson"];


    NSLog(@"The two people are %@ and %@",
         [person lastNameFirstNameString], [shoutingPerson
lastNameFirstNameString]);
}
@end
```

As well as just adding methods to existing classes, you can also use categories to split the implementation of a complex class across multiple source code files. You might, for example, put the drawing code for a custom user interface element in a separate file to the rest of the implementation if the geometrical calculations, colors, and gradients, etc, are particularly complicated. Alternatively, you could provide different implementations for the category methods, depending on whether you were writing an app for OS X or iOS.

Categories can be used to declare either instance methods or class methods but are not usually suitable for declaring additional properties. It's valid syntax to include a property declaration in a category interface, but it's not possible to declare an additional instance variable in a category. This means the compiler won't synthesize any instance variable, nor will it synthesize any property accessor methods. You can write your own accessor methods in the category implementation, but you won't be able to keep track of a value for that property unless it's already stored by the original class.

The only way to add a traditional property—backed by a new instance variable—to an existing class is to use a class extension, as described in Class Extensions Extend the Internal Implementation.

**Note:** Cocoa and Cocoa Touch include a variety of categories for some of the primary framework classes.

The string-drawing functionality mentioned in the introduction to this chapter is in fact already provided for `NSString` by the `NSStringDrawing` category for OS X, which includes the `drawAtPoint:withAttributes:` and `drawInRect:withAttributes:` methods. For iOS, the `UIStringDrawing` category includes methods such as `drawAtPoint:withFont:` and `drawInRect:withFont:`.

## Avoid Category Method Name Clashes

Because the methods declared in a category are added to an existing class, you need to be very careful about method names.

If the name of a method declared in a category is the same as a method in the original class, or a method in another category on the same class (or even a superclass), the behavior is undefined as to which method implementation is used at runtime. This is less likely to be an issue if you're using categories with your own classes, but can cause problems when using categories to add methods to standard Cocoa or Cocoa Touch classes.

An application that works with a remote web service, for example, might need an easy way to encode a string of characters using Base64 encoding. It would make sense to define a category on `NSString` to add an instance method to return a Base64-encoded version of a string, so you might add a convenience method called `base64EncodedString`.

A problem arises if you link to another framework that also happens to define its own category on `NSString`, including its own method called `base64EncodedString`. At runtime, only one of the method implementations will "win" and be added to `NSString`, but which one is undefined.

Another problem can arise if you add convenience methods to Cocoa or Cocoa Touch classes that are then added to the original classes in later releases. The `NSSortDescriptor` class, for example, which describes how a collection of objects should be ordered, has always had an `initWithKey:ascending:` initialization method, but didn't offer a corresponding class factory method under early OS X and iOS versions.

By convention, the class factory method should be called `sortDescriptorWithKey:ascending:`, so you might have chosen to add a category on `NSSortDescriptor` to provide this method for convenience. This would have worked as you'd expect under older versions of OS X and iOS, but with the release of Mac OS X version 10.6 and iOS 4.0, a `sortDescriptorWithKey:ascending:` method was added to the original `NSSortDescriptor` class, meaning you'd now end up with a naming clash when your application was run on these or later platforms.

In order to avoid undefined behavior, it's best practice to add a prefix to method names in categories on framework classes, just like you should add a prefix to the names of your own *classes*. You might choose to use the same three letters you use for your class prefixes, but lowercase to follow the usual convention for method names, then an underscore, before the rest of the method name. For the `NSSortDescriptor` example, your own category might look like this:

```
@interface NSSortDescriptor (XYZAdditions)
+ (id)xyz_sortDescriptorWithKey:(NSString *)key ascending:(BOOL)ascending;
@end
```

This means you can be sure that your method will be used at runtime. The ambiguity is removed because your code now looks like this:

```
    NSSortDescriptor *descriptor =
                 [NSSortDescriptor xyz_sortDescriptorWithKey:@"name" ascending:YES];
```

## Class Extensions Extend the Internal Implementation

A class extension bears some similarity to a category, but it can only be added to a class for which you have the source code at compile time (the class is compiled at the same time as the class extension). The methods declared by a class extension are implemented in the `@implementation` block for the original class so you can't, for example, declare a class extension on a framework class, such as a Cocoa or Cocoa Touch class like `NSString`.

The syntax to declare a class extension is similar to the syntax for a category, and looks like this:

```
@interface ClassName ()


@end
```

Because no name is given in the parentheses, class extensions are often referred to as *anonymous categories*.

Unlike regular categories, a class extension can add its own properties and instance variables to a class. If you declare a property in a class extension, like this:

```
@interface XYZPerson ()
@property NSObject *extraProperty;
@end
```

the compiler will automatically synthesize the relevant accessor methods, as well as an instance variable, inside the primary class implementation.

If you add any *methods* in a class extension, these must be implemented in the primary implementation for the class.

It's also possible to use a class extension to add custom instance variables. These are declared inside braces in the class extension interface:

```
@interface XYZPerson () {
    id _someCustomInstanceVariable;
}
...
@end
```

### Use Class Extensions to Hide Private Information

The primary interface for a class is used to define the way that other classes are expected to interact with it. In other words, it's the *public* interface to the class.

Class extensions are often used to extend the public interface with additional *private* methods or properties for use within the implementation of the class itself. It's common, for example, to define a property as `readonly` in the interface, but as `readwrite` in a class extension declared above the implementation, in order that the internal methods of the class can change the property value directly.

As an example, the `XYZPerson` class might add a property called `uniqueIdentifier`, designed to keep track of information like a Social Security Number in the US.

It usually requires a large amount of paperwork to have a unique identifier assigned to an individual in the real world, so the `XYZPerson` class interface might declare this property as `readonly`, and provide some method that requests an identifier be assigned, like this:

```
@interface XYZPerson : NSObject
...
@property (readonly) NSString *uniqueIdentifier;
- (void)assignUniqueIdentifier;
@end
```

This means that it's not possible for the `uniqueIdentifier` to be set directly by another object. If a person doesn't already have one, a request must be made to assign an identifier by calling the `assignUniqueIdentifier` method.

In order for the `XYZPerson` class to be able to change the property internally, it makes sense to redeclare the property in a class extension that's defined at the top of the implementation file for the class:

```
@interface XYZPerson ()
@property (readwrite) NSString *uniqueIdentifier;
@end


@implementation XYZPerson
...
@end
```

**Note:** The `readwrite` attribute is optional, because it's the default. You may like to use it when redeclaring a property, for clarity.

This means that the compiler will now also synthesize a setter method, so any method inside the `XYZPerson` implementation will be able to set the property directly using either the setter or dot syntax.

By declaring the class extension inside the source code file for the `XYZPerson` implementation, the information stays private to the `XYZPerson` class. If another type of object tries to set the property, the compiler will generate an error.

**Note:** By adding the class extension shown above, redeclaring the `uniqueIdentifier` property as a `readwrite` property, a `setUniqueIdentifier:` method will exist at runtime on every `XYZPerson` object, regardless of whether other source code files were aware of the class extension or not.

The compiler will complain if code in one of those other source code files attempts to call a private method or set a `readonly` property, but it's possible to avoid compiler errors and leverage dynamic runtime features to call those methods in other ways, such as by using one of the `performSelector:...` methods offered by `NSObject`. You should avoid a class hierarchy or design where this is necessary; instead, the primary class interface should always define the correct "public" interactions.

If you intend to make "private" methods or properties available to select other classes, such as related classes within a framework, you can declare the class extension in a separate header file and import it in the source files that need it. It's not uncommon to have two header files for a class, for example, such as `XYZPerson.h` and `XYZPersonPrivate.h`. When you release the framework, you only release the public `XYZPerson.h` header file.

## Consider Other Alternatives for Class Customization

Categories and class extensions make it easy to add behavior directly to an existing class, but sometimes this isn't the best option.

One of the primary goals of object-oriented programming is to write reusable code, which means that classes should be reusable in a variety of situations, wherever possible. If you're creating a view class to describe an object that displays information on screen, for example, it's a good idea to think whether the class could be usable in multiple situations.

Rather than hard-coding decisions about layout or content, one alternative is to leverage inheritance and leave those decisions in methods specifically designed to be overridden by subclasses. Although this does make it relatively easy to reuse the class, you still need to create a new subclass every time you want to make use of that original class.

Another alternative is for a class to use a *delegate* object. Any decisions that might limit reusability can be delegated to another object, which is left to make those decisions at runtime. One common example is a standard table view class (`NSTableView` for OS X and `UITableView` for iOS). In order for a generic table view (an object that displays information using one or more columns and rows) to be useful, it leaves decisions about its content to be decided by another object at runtime. Delegation is covered in detail in the next chapter, Working with Protocols.

### Interact Directly with the Objective-C Runtime

Objective-C offers its dynamic behavior through the Objective-C runtime system.

Many decisions, such as which methods are called when messages are sent, aren't made at compile-time but are instead determined when the application is run. Objective-C is more than just a language that is compiled down to machine code. Instead, it requires a runtime system in place to execute that code.

It's possible to interact directly with this runtime system, such as by adding *associative references* to an object. Unlike class extensions, associated references do not affect the original class declaration and implementation, which means you can use them with framework classes for which you don't have access to the original source code.

An associative reference links one object with another, in a similar way to a property or instance variable. For more information, see Associative References. To learn more about the Objective-C Runtime in general, see *Objective-C Runtime Programming Guide*.

## Exercises

1. Add a category to the `XYZPerson` class to declare and implement additional behavior, such as displaying a person's name in different ways.
2. Add a category to `NSString` in order to add a method to draw the uppercase version of a string at a given point, calling through to one of the existing `NSStringDrawing` category methods to perform the actual drawing. These methods are documented in *NSString UIKit Additions Reference* for iOS and *NSString Application Kit Additions Reference* for OS X.
3. Add two `readonly` properties to the original `XYZPerson` class implementation to represent a person's height and weight, along with methods to `measureWeight` and `measureHeight`.

   Use a class extension to redeclare the properties as `readwrite`, and implement the methods to set the properties to suitable values.