Working with Protocols

In the real world, people on official business are often required to follow strict procedures when dealing with certain situations. Law enforcement officials, for example, are required to "follow protocol" when making enquiries or collecting evidence.

In the world of object-oriented programming, it's important to be able to define a set of behavior that is expected of an object in a given situation. As an example, a table view expects to be able to communicate with a data source object in order to find out what it is required to display. This means that the data source must respond to a specific set of messages that the table view might send.

The data source could be an instance of any class, such as a view controller (a subclass of `NSViewController` on OS X or `UIViewController` on iOS) or a dedicated data source class that perhaps just inherits from `NSObject`. In order for the table view to know whether an object is suitable as a data source, it's important to be able to declare that the object implements the necessary methods.

Objective-C allows you to define *protocols*, which declare the methods expected to be used for a particular situation. This chapter describes the syntax to define a formal protocol, and explains how to mark a class interface as *conforming* to a protocol, which means that the class must implement the required methods.

Protocols Define Messaging Contracts

A class interface declares the methods and properties associated with that class. A protocol, by contrast, is used to declare methods and properties that are independent of any specific class.

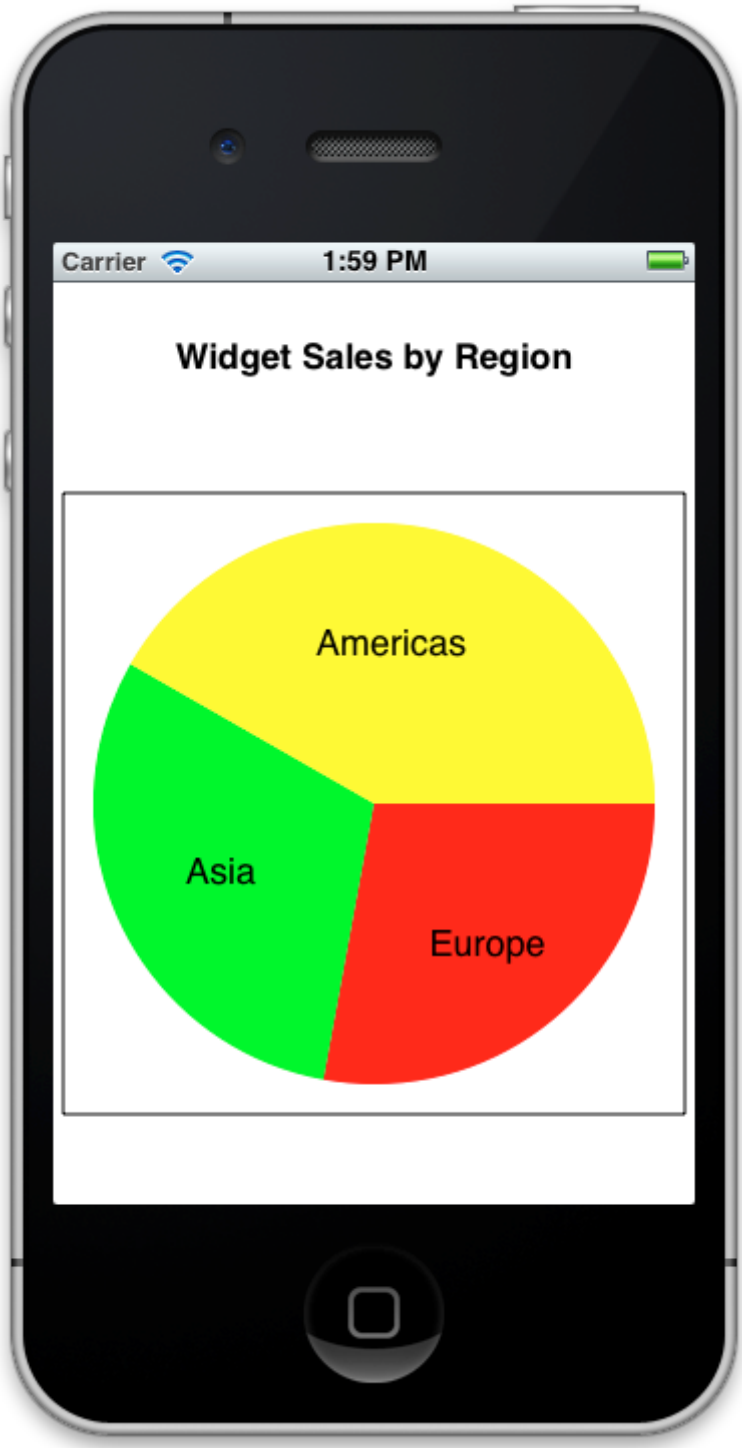
The basic syntax to define a protocol looks like this:

```
@protocol ProtocolName
// list of methods and properties
@end
```

Protocols can include declarations for both instance methods and class methods, as well as properties.

As an example, consider a custom view class that is used to display a pie chart, as shown in Figure 5-1.

Figure 5-1 A Custom Pie Chart View



To make the view as reusable as possible, all decisions about the information should be left to another object, a data source. This means that multiple instances of the same view class could display different information just by communicating with different sources.

The minimum information needed by the pie chart view includes the number of segments, the relative size of each segment, and the title of each segment. The pie chart's data source protocol, therefore, might look like this:

```
@protocol XYZPieChartViewDataSource
- (NSUInteger)numberOfSegments;
- (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
- (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
@end
```

Note: This protocol uses the `NSUInteger` value for unsigned integer scalar values. This type is discussed in more detail in the next chapter.

The pie chart view class interface would need a property to keep track of the data source object. This object could be of any class, so the basic property type will be `id`. The only thing that is known about the object is that it conforms to the relevant protocol.

The syntax to declare the data source property for the view would look like this:

```
@interface XYZPieChartView : UIView
@property (weak) id <XYZPieChartViewDataSource> dataSource;
...
@end
```

Objective-C uses angle brackets to indicate conformance to a protocol. This example declares a weak property for a generic object pointer that conforms to the `XYZPieChartViewDataSource` protocol.

Note: Delegate and data source properties are usually marked as `weak` for the object graph management reasons described earlier, in Avoid Strong Reference Cycles.

By specifying the required protocol conformance on the property, you'll get a compiler warning if you attempt to set the property to an object that doesn't conform to the protocol, even though the basic property class type is generic. It doesn't matter whether the object is an instance of `UIViewController` or `NSObject`. All that matters is that it conforms to the protocol, which means the pie chart view knows it can request the information it needs.

Protocols Can Have Optional Methods

By default, all methods declared in a protocol are required methods. This means that any class that conforms to the protocol must implement those methods.

It's also possible to specify *optional* methods in a protocol. These are methods that a class can implement only if it needs to.

As an example, you might decide that the titles on the pie chart should be optional. If the data source object doesn't implement the `titleForSegmentAtIndex:`, no titles should be shown in the view.

You can mark protocol methods as optional using the `@optional` directive, like this:

```
@protocol XYZPieChartViewDataSource
- (NSUInteger)numberOfSegments;
- (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
@optional
- (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
@end
```

In this case, only the `titleForSegmentAtIndex:` method is marked optional. The previous methods have no directive, so are assumed to be required.

The `@optional` directive applies to any methods that follow it, either until the end of the protocol definition, or until another directive is encountered, such as `@required`. You might add further methods to the protocol like this:

```
@protocol XYZPieChartViewDataSource
- (NSUInteger)numberOfSegments;
- (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
@optional
```

```
- (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
- (BOOL)shouldExplodeSegmentAtIndex:(NSUInteger)segmentIndex;
@required
- (UIColor *)colorForSegmentAtIndex:(NSUInteger)segmentIndex;
@end
```

This example defines a protocol with three required methods and two optional methods.

Check that Optional Methods Are Implemented at Runtime

If a method in a protocol is marked as optional, you must check whether an object implements that method before attempting to call it.

As an example, the pie chart view might test for the segment title method like this:

```
    NSString *thisSegmentTitle;
    if ([self.dataSource respondsToSelector:@selector(titleForSegmentAtIndex:)]) {
        thisSegmentTitle = [self.dataSource titleForSegmentAtIndex:index];
    }
```

The `respondsToSelector:` method uses a selector, which refers to the identifier for a method after compilation. You can provide the correct identifier by using the `@selector()` directive and specifying the name of the method.

If the data source in this example implements the method, the title is used; otherwise, the title remains `nil`.

> **Remember:** Local *object* variables are automatically initialized to `nil`.

If you attempt to call the `respondsToSelector:` method on an `id` conforming to the protocol as it's defined above, you'll get a compiler error that there's no known instance method for it. Once you qualify an `id` with a protocol, all static type-checking comes back; you'll get an error if you try to call any method that isn't defined in the specified protocol. One way to avoid the compiler error is to set the custom protocol to adopt the `NSObject` protocol.

Protocols Inherit from Other Protocols

In the same way that an Objective-C class can inherit from a superclass, you can also specify that one protocol conforms to another.

As an example, it's best practice to define your protocols to conform to the `NSObject` protocol (some of the `NSObject` behavior is split from its class interface into a separate protocol; the `NSObject` class adopts the `NSObject` protocol).

By indicating that your own protocol conforms to the `NSObject` protocol, you're indicating that any object that adopts the custom protocol will also provide implementations for each of the `NSObject` protocol methods. Because you're presumably using some subclass of `NSObject`, you don't need to worry about providing your own implementations for these `NSObject` methods. The protocol adoption is useful, however, for situations like that described above.

To specify that one protocol conforms to another, you provide the name of the other protocol in angle brackets, like this:

```
@protocol MyProtocol <NSObject>
...
@end
```

In this example, any object that adopts `MyProtocol` also effectively adopts all the methods declared in the `NSObject` protocol.

Conforming to Protocols

The syntax to indicate that a class adopts a protocol again uses angle brackets, like this

```
@interface MyClass : NSObject <MyProtocol>
...
@end
```

This means that any instance of `MyClass` will respond not only to the methods declared specifically in the interface, but that `MyClass` also provides implementations for the required methods in `MyProtocol`. There's no need to redeclare the protocol methods in the class interface—the adoption of the protocol is sufficient.

Note: The compiler does not automatically synthesize properties declared in adopted protocols.

If you need a class to adopt multiple protocols, you can specify them as a comma-separated list, like this:

```
@interface MyClass : NSObject <MyProtocol, AnotherProtocol, YetAnotherProtocol>
...
@end
```

Tip: If you find yourself adopting a large number of protocols in a class, it may be a sign that you need to refactor an overly-complex class by splitting the necessary behavior across multiple smaller classes, each with clearly-defined responsibilities.

One relatively common pitfall for new OS X and iOS developers is to use a single application delegate class to contain the majority of an application's functionality (managing underlying data structures, serving the data to multiple user interface elements, as well as responding to gestures and other user interaction). As complexity increases, the class becomes more difficult to maintain.

Once you've indicated conformance to a protocol, the class must at least provide method implementations for each of the required protocol methods, as well as any optional methods you choose. The compiler will warn you if you fail to implement any of the required methods.

Note: The method declaration in a protocol is just like any other declaration. The method name and argument types in the implementation must match the declaration in the protocol.

Cocoa and Cocoa Touch Define a Large Number of Protocols

Protocols are used by Cocoa and Cocoa Touch objects for a variety of different situations. For example, the table view classes (`NSTableView` for OS X and `UITableView` for iOS) both use a data source object to supply them with the necessary information. Both define their own data source protocol, which is used in much the same way as the `XYZPieChartViewDataSource` protocol example above. Both table view classes also allow you to set a delegate object, which again must conform to the relevant `NSTableViewDelegate` or `UITableViewDelegate` protocol. The delegate is responsible for dealing with user interactions, or customizing the display of certain entries.

Some protocols are used to indicate *non-hierarchical similarities* between classes. Rather than being

linked to specific class requirements, some protocols instead relate to more general Cocoa or Cocoa Touch communication mechanisms that may be adopted by multiple, unrelated classes.

For example, many framework model objects (such as the collection classes like `NSArray` and `NSDictionary`) support the `NSCoding` protocol, which means they can encode and decode their properties for archival or distribution as raw data. `NSCoding` makes it relatively easy to write entire object graphs to disk, provided every object within the graph adopts the protocol.

A few Objective-C language-level features also rely on protocols. In order to use fast enumeration, for example, a collection must adopt the `NSFastEnumeration` protocol, as described in Fast Enumeration Makes It Easy to Enumerate a Collection. Additionally, some objects can be copied, such as when using a property with a `copy` attribute as described in Copy Properties Maintain Their Own Copies. Any object you try to copy must adopt the `NSCopying` protocol, otherwise you'll get a runtime exception.

Protocols Are Used for Anonymity

Protocols are also useful in situations where the class of an object isn't known, or needs to stay hidden.

As an example, the developer of a framework may choose not to publish the interface for one of the classes within the framework. Because the class name isn't known, it's not possible for a *user* of the framework to create an instance of that class directly. Instead, some other object in the framework would typically be designated to return a ready-made instance, like this:

```
id utility = [frameworkObject anonymousUtility];
```

In order for this `anonymousUtility` object to be useful, the developer of the framework can publish a protocol that reveals some of its methods. Even though the original class interface isn't provided, which means the class stays anonymous, the object can still be used in a limited way:

```
id <XYZFrameworkUtility> utility = [frameworkObject anonymousUtility];
```

If you're writing an iOS app that uses the Core Data framework, for example, you'll likely run into the `NSFetchedResultsController` class. This class is designed to help a data source object supply stored data to an iOS `UITableView`, making it easy to provide information like the number of rows.

If you're working with a table view whose content is split into multiple sections, you can also ask a fetched results controller for the relevant section information. Rather than returning a specific class containing this section information, the `NSFetchedResultsController` class instead returns an anonymous object, which conforms to the `NSFetchedResultsSectionInfo` protocol. This means it's still possible to query the object for the information you need, such as the number of rows in a section:

```
NSInteger sectionNumber = ...
id <NSFetchedResultsSectionInfo> sectionInfo =
        [self.fetchedResultsController.sections objectAtIndex:sectionNumber];
NSInteger numberOfRowsInSection = [sectionInfo numberOfObjects];
```

Even though you don't know the class of the `sectionInfo` object, the `NSFetchedResultsSectionInfo` protocol dictates that it can respond to the `numberOfObjects` message.