Previous Next

Protocols

Protocols declare methods that can be implemented by any class. Protocols are useful in at least three situations:

- To declare methods that others are expected to implement
- To declare the *interface* to an object while concealing its class
- To capture similarities among classes that are not hierarchically related

Declaring Interfaces for Others to Implement

Class and category interfaces declare methods that are associated with a particular class—mainly methods that the class implements. Informal and formal *protocols*, on the other hand, declare methods that are independent of any specific class, but which any class, and perhaps many classes, might implement.

A protocol is simply a list of method declarations, unattached to a class definition. For example, these methods that report user actions on the mouse could be gathered into a protocol:

```
- (void)mouseDown:(NSEvent *)theEvent;
- (void)mouseDragged:(NSEvent *)theEvent;
- (void)mouseUp:(NSEvent *)theEvent;
```

Any class that wanted to respond to mouse *events* could adopt the protocol and implement its methods.

Protocols free method declarations from dependency on the class hierarchy, so they can be used in ways that classes and categories cannot. Protocols list methods that are (or may be) implemented somewhere, but the identity of the class that implements them is not of interest. What is of interest is whether or not a particular class *conforms* to the protocol—whether it has implementations of the methods the protocol declares. Thus objects can be grouped into types not just on the basis of similarities resulting from inheriting from the same class, but also on the basis of their similarity in conforming to the same protocol. Classes in unrelated branches of the inheritance hierarchy might be typed alike because they conform to the same protocol.

Protocols can play a significant role in object-oriented design, especially when a project is divided among many implementors or it incorporates objects developed in other projects. Cocoa software uses protocols heavily to support interprocess communication through Objective-C messages.

However, an Objective-C program doesn't need to use protocols. Unlike class definitions and message expressions, they're optional. Some Cocoa frameworks use them; some don't. It all depends on the task at hand.

Methods for Others to Implement

If you know the class of an object, you can look at its interface declaration (and the interface declarations of the classes it inherits from) to find what messages it responds to. These declarations advertise the messages it can receive. Protocols provide a way for it to also advertise the messages it sends.

Communication works both ways; objects send messages as well as receive them. For example, an object might delegate responsibility for a certain operation to another object, or it may on occasion simply need to ask another object for information. In some cases, an object might be willing to notify other objects of its actions so that they can take whatever collateral measures might be required.

If you develop the class of the sender and the class of the receiver as part of the same project (or if someone else has supplied you with the receiver and its interface file), this communication is easily coordinated. The sender simply imports the interface file of the receiver. The imported file declares the method selectors the sender uses in the messages it sends.

However, if you develop an object that sends messages to objects that aren't yet defined—objects that you're leaving for others to implement—you won't have the receiver's interface file. You need another way to declare the methods you use in messages but don't implement. A protocol serves this purpose. It informs the compiler about methods the class uses and also informs other implementors of the methods they need to define to have their objects work with yours.

Suppose, for example, that you develop an object that asks for the assistance of another object by sending it `helpOut:` and other messages. You provide an `assistant` instance variable to record the *outlet* for these messages and define a companion method to set the instance variable. This method lets other objects register themselves as potential recipients of your object's messages:

```
- setAssistant:anObject
{
    assistant = anObject;
}
```

Then, whenever a message is to be sent to the `assistant`, a check is made to be sure that the receiver implements a method that can respond:

```
- (BOOL)doWork
{
    ...
    if ( [assistant respondsToSelector:@selector(helpOut:)] ) {
        [assistant helpOut:self];
        return YES;
    }
    return NO;
}
```

Because, at the time you write this code, you can't know what kind of object might register itself as the `assistant`, you can only declare a protocol for the `helpOut:` method; you can't import the interface file of the class that implements it.

Declaring Interfaces for Anonymous Objects

A protocol can be used to declare the methods of an *anonymous object*, an object of unknown class. An anonymous object may represent a service or handle a limited set of functions, especially when only one object of its kind is needed. (Objects that play a fundamental role in defining an application's architecture and objects that you must initialize before using are not good candidates for anonymity.)

Objects are not anonymous to their developers, of course, but they are anonymous when the developer supplies them to someone else. For example, consider the following situations:

- Someone who supplies a framework or a suite of objects for others to use can include objects that are not identified by a class name or an interface file. Lacking the name and class interface, users have no way of creating instances of the class. Instead, the supplier must provide a ready-made instance. Typically, a method in another class returns a usable object:

  ```
  id formatter = [receiver formattingService];
  ```

  The object returned by the method is an object without a class identity, at least not one the supplier is willing to reveal. For it to be of any use at all, the supplier must be willing to identify at least some of the messages that it can respond to. The messages are identified by associating the object with a list of methods declared in a protocol.
- You can send Objective-C messages to *remote objects*—objects in other applications.

  Each application has its own structure, classes, and internal logic. But you don't need to know how another application works or what its components are to communicate with it. As an outsider, all you need to know is what messages you can send (the protocol) and where to send them (the receiver).

  An application that publishes one of its objects as a potential receiver of *remote messages* must also publish a protocol declaring the methods the object will use to respond to those messages. It doesn't have to disclose anything else about the object. The sending application doesn't need to know the class of the object or use the class in its own design. All it needs is the protocol.

Protocols make anonymous objects possible. Without a protocol, there would be no way to declare an interface to an object without identifying its class.

> **Note:** Even though the supplier of an anonymous object doesn't reveal its class, the object itself reveals it at runtime. A class message returns the anonymous object's class. However, there's usually little point in discovering this extra information; the information in the protocol is sufficient.

Nonhierarchical Similarities

If more than one class implements a set of methods, those classes are often grouped under an abstract class that declares the methods they have in common. Each subclass can reimplement the methods in its own way, but the inheritance hierarchy and the common declaration in the abstract class capture the essential similarity between the subclasses.

However, sometimes it's not possible to group common methods in an abstract class. Classes that are unrelated in most respects might nevertheless need to implement some similar methods. This limited similarity may not justify a hierarchical relationship. For example, you might want to add support for creating XML representations of objects in your application and for initializing objects from an XML representation:

```
- (NSXMLElement *)XMLRepresentation;
- initFromXMLRepresentation:(NSXMLElement *)xmlString;
```

These methods could be grouped into a protocol and the similarity between implementing classes accounted for by noting that they all conform to the same protocol.

Objects can be typed by this similarity (the protocols they conform to), rather than by their class. For example, an `NSMatrix` instance must communicate with the objects that represent its cells. The matrix could require each of these objects to be a kind of `NSCell` (a type based on class) and rely on the fact that all objects that inherit from the `NSCell` class have the methods needed to respond to `NSMatrix` messages. Alternatively, the `NSMatrix` object could require objects representing cells to have methods that can respond to a particular set of messages (a type based on protocol). In this case, the `NSMatrix` object wouldn't care what class a cell object belonged to, just that it implemented the methods.

Formal Protocols

The Objective-C language provides a way to formally declare a list of methods (including declared properties) as a protocol. *Formal protocols* are supported by the language and the runtime system. For example, the compiler can check for types based on protocols, and objects can introspect at runtime to report whether or not they conform to a protocol.

Declaring a Protocol

You declare formal protocols with the `@protocol` directive:

```
@protocol ProtocolName
method declarations
@end
```

For example, you could declare an XML representation protocol like this:

```
@protocol MyXMLSupport
- initFromXMLRepresentation:(NSXMLElement *)XMLElement;
- (NSXMLElement *)XMLRepresentation;
@end
```

Unlike class names, protocol names don't have global visibility. They live in their own namespace.

Optional Protocol Methods

Protocol methods can be marked as optional using the `@optional` keyword. Corresponding to the `@optional` modal keyword, there is a `@required` keyword to formally denote the semantics of the default behavior. You can use `@optional` and `@required` to partition your protocol into sections as you see fit. If you do not specify any keyword, the default is `@required`.

```
@protocol MyProtocol

- (void)requiredMethod;

@optional
- (void)anOptionalMethod;
- (void)anotherOptionalMethod;

@required
- (void)anotherRequiredMethod;

@end
```

> **Note:** In OS X v10.5, protocols cannot include optional declared properties. This constraint is removed in OS X v10.6 and later.

Informal Protocols

In addition to formal protocols, you can also define an *informal protocol* by grouping the methods in a category declaration:

```
@interface NSObject ( MyXMLSupport )
- initFromXMLRepresentation:(NSXMLElement *)XMLElement;
- (NSXMLElement *)XMLRepresentation;
@end
```

Informal protocols are typically declared as categories of the `NSObject` class, because that broadly associates the method names with any class that inherits from `NSObject`. Because all classes inherit from the root class, the methods aren't restricted to any part of the *inheritance hierarchy*. (It is also possible to declare an informal protocol as a category of another class to limit it to a certain branch of the inheritance hierarchy, but there is little reason to do so.)

When used to declare a protocol, a category interface doesn't have a corresponding implementation. Instead, classes that implement the protocol declare the methods again in their own interface files and define them along with other methods in their *implementation* files.

An informal protocol bends the rules of category declarations to list a group of methods but not associate them with any particular class or implementation.

Being informal, protocols declared in categories don't receive much language support. There's no type checking at compile time nor a check at runtime to see whether an object conforms to the protocol. To get these benefits, you must use a formal protocol. An informal protocol may be useful when all the methods are optional, such as for a *delegate*, but (in OS X v10.5 and later) it is typically better to use a formal protocol with optional methods.

Protocol Objects

Just as classes are represented at runtime by class objects and methods by selector codes, formal protocols are represented by a special data type—instances of the `Protocol` class. Source code that deals with a protocol (other than to use it in a type specification) must refer to the corresponding protocol object.

In many ways, protocols are similar to class definitions. They both declare methods, and at runtime they're both represented by objects—classes by instances of `Class` and protocols by instances of `Protocol`. Like class objects, protocol objects are created automatically from the definitions and declarations found in source code and are used by the runtime system. They're not allocated and initialized in program source code.

Source code can refer to a protocol object using the `@protocol()` directive—the same directive that declares a protocol, except that here it has a set of trailing parentheses. The parentheses enclose the protocol name:

```
Protocol *myXMLSupportProtocol = @protocol(MyXMLSupport);
```

This is the only way that source code can conjure up a protocol object. Unlike a class name, a protocol name doesn't designate the object—except inside `@protocol()`.

The compiler creates a protocol object for each protocol declaration it encounters, but only if the protocol is also:

- Adopted by a class, or
- Referred to somewhere in source code (using `@protocol()`)

Protocols that are declared but not used (except for type checking as described below) aren't represented by protocol objects at runtime.

Adopting a Protocol

Adopting a protocol is similar in some ways to declaring a superclass. Both assign methods to the class. The superclass declaration assigns it inherited methods; the protocol assigns it methods declared in the protocol list. A class is said to *adopt* a formal protocol if in its declaration it lists the protocol within angle brackets after the superclass name:

```
@interface ClassName : ItsSuperclass < protocol list >
```

Categories adopt protocols in much the same way:

```
@interface ClassName ( CategoryName ) < protocol list >
```

A class can adopt more than one protocol; names in the protocol list are separated by commas.

```
@interface Formatter : NSObject < Formatting, Prettifying >
```

A class or category that adopts a protocol must implement all the required methods the protocol declares, otherwise the compiler issues a warning. The Formatter class above would define all the required methods declared in the two protocols it adopts, in addition to any it might have declared itself.

A class or category that adopts a protocol must import the header file where the protocol is declared. The methods declared in the adopted protocol are not declared elsewhere in the class or category interface.

It's possible for a class to simply adopt protocols and declare no other methods. For example, the following class declaration adopts the `Formatting` and `Prettifying` protocols, but declares no instance variables or methods of its own:

```
@interface Formatter : NSObject < Formatting, Prettifying >
@end
```

Conforming to a Protocol

A class is said to *conform* to a formal protocol if it adopts the protocol or inherits from another class that adopts it. An instance of a class is said to conform to the same set of protocols its class conforms to.

Because a class must implement all the required methods declared in the protocols it adopts, saying that a class or an instance conforms to a protocol is equivalent to saying that it has in its repertoire all the methods the protocol declares.

It's possible to check whether an object conforms to a protocol by sending it a `conformsToProtocol:` message.

```
if ( ! [receiver conformsToProtocol:@protocol(MyXMLSupport)]  ) {
    // Object does not conform to MyXMLSupport protocol
    // If you are expecting receiver to implement methods declared in the
    //  MyXMLSupport protocol, this is probably an error
}
```

(Note that there is also a class method with the same name—`conformsToProtocol:`.)

The `conformsToProtocol:` test is like the `respondsToSelector:` test for a single method, except that it tests whether a protocol has been adopted (and presumably all the methods it declares implemented) rather than just whether one particular method has been implemented. Because it checks for all the methods in the protocol, `conformsToProtocol:` can be more efficient than `respondsToSelector:`.

The `conformsToProtocol:` test is also like the `isKindOfClass:` test, except that it tests for a type based on a protocol rather than a type based on the inheritance hierarchy.

Type Checking

Type declarations for objects can be extended to include formal protocols. Protocols thus offer the possibility of another level of type checking by the compiler, one that's more abstract since it's not tied to particular implementations.

In a type declaration, protocol names are listed between angle brackets after the type name:

```
- (id <Formatting>)formattingService;
id <MyXMLSupport> anObject;
```

Just as static typing permits the compiler to test for a type based on the class hierarchy, this syntax permits the compiler to test for a type based on conformance to a protocol.

For example, if `Formatter` is an abstract class, the declaration

```
Formatter *anObject;
```

groups all objects that inherit from Formatter into a type and permits the compiler to check assignments against that type.

Similarly, the declaration

```
id <Formatting> anObject;
```

groups all objects that conform to the Formatting protocol into a type, regardless of their positions in the class hierarchy. The compiler can make sure only objects that conform to the protocol are assigned to the type.

In each case, the type groups similar objects—either because they share a common inheritance, or because they converge on a common set of methods.

The two types can be combined in a single declaration:

```
Formatter <Formatting> *anObject;
```

Protocols can't be used to type class objects. Only instances can be statically typed to a protocol, just as only instances can be statically typed to a class. (However, at runtime, both classes and instances respond to a `conformsToProtocol:` message.)

Protocols Within Protocols

One protocol can incorporate other protocols using the same syntax that classes use to adopt a protocol:

```
@protocol ProtocolName < protocol list >
```

All the protocols listed between angle brackets are considered part of the *ProtocolName* protocol. For example, if the `Paging` protocol incorporates the `Formatting` protocol

```
@protocol Paging < Formatting >
```

any object that conforms to the `Paging` protocol also conforms to `Formatting`. Type declarations such as

```
id <Paging> someObject;
```

and `conformsToProtocol:` messages such as

```
if ( [anotherObject conformsToProtocol:@protocol(Paging)] )
    ...
```

need to mention only the `Paging` protocol to test for conformance to `Formatting` as well.

When a class adopts a protocol, it must implement the required methods the protocol declares, as mentioned earlier. In addition, it must conform to any protocols the adopted protocol incorporates. If an incorporated protocol incorporates still other protocols, the class must also conform to them. A class can conform to an incorporated protocol using either of these techniques:

- Implementing the methods the protocol declares
- Inheriting from a class that adopts the protocol and implements the methods

Suppose, for example, that the `Pager` class adopts the `Paging` protocol. If `Pager` is a subclass of `NSObject` as shown here:

```
@interface Pager : NSObject < Paging >
```

it must implement all the `Paging` methods, including those declared in the incorporated `Formatting` protocol. It adopts the `Formatting` protocol along with `Paging`.

On the other hand, if `Pager` is a subclass of `Formatter` (a class that independently adopts the `Formatting` protocol) as shown here:

```
@interface Pager : Formatter < Paging >
```

it must implement all the methods declared in the `Paging` protocol proper, but not those declared in `Formatting`. `Pager` inherits conformance to the `Formatting` protocol from `Formatter`.

Note that a class can conform to a protocol without formally adopting it, simply by implementing the methods declared in the protocol.

Referring to Other Protocols

When working on complex applications, you occasionally find yourself writing code that looks like this:

```
#import "B.h"

@protocol A
- foo:(id <B>)anObject;
@end
```

where protocol `B` is declared like this:

```
#import "A.h"

@protocol B
- bar:(id <A>)anObject;
@end
```

In such a situation, circularity results and neither file will compile correctly. To break this recursive cycle, you must use the `@protocol` directive to make a forward reference to the needed protocol instead of importing the interface file where the protocol is defined:

```
@protocol B;

@protocol A
- foo:(id <B>)anObject;
@end
```

Note that using the `@protocol` directive in this manner simply informs the compiler that `B` is a protocol to be defined later. It doesn't import the interface file where protocol `B` is defined.

Previous Next

Copyright © 2013 Apple Inc. All Rights Reserved. Terms of Use | Privacy Policy | Updated: 2013-04-23