Message Forwarding

Sending a message to an object that does not handle that message is an error. However, before announcing the error, the runtime system gives the receiving object a second chance to handle the message.

Forwarding

If you send a message to an object that does not handle that message, before announcing an error the runtime sends the object a `forwardInvocation:` message with an `NSInvocation` object as its sole argument—the `NSInvocation` object encapsulates the original message and the arguments that were passed with it.

You can implement a `forwardInvocation:` method to give a default response to the message, or to avoid the error in some other way. As its name implies, `forwardInvocation:` is commonly used to forward the message to another object.

To see the scope and intent of forwarding, imagine the following scenarios: Suppose, first, that you're designing an object that can respond to a message called `negotiate`, and you want its response to include the response of another kind of object. You could accomplish this easily by passing a `negotiate` message to the other object somewhere in the body of the `negotiate` method you implement.

Take this a step further, and suppose that you want your object's response to a `negotiate` message to be exactly the response implemented in another class. One way to accomplish this would be to make your class inherit the method from the other class. However, it might not be possible to arrange things this way. There may be good reasons why your class and the class that implements `negotiate` are in different branches of the inheritance hierarchy.

Even if your class can't inherit the `negotiate` method, you can still "borrow" it by implementing a version of the method that simply passes the message on to an instance of the other class:

```
- (id)negotiate
{
    if ( [someOtherObject respondsTo:@selector(negotiate)] )
        return [someOtherObject negotiate];
    return self;
}
```

This way of doing things could get a little cumbersome, especially if there were a number of messages you wanted your object to pass on to the other object. You'd have to implement one method to cover each method you wanted to borrow from the other class. Moreover, it would be impossible to handle cases where you didn't know, at the time you wrote the code, the full set of messages you might want to forward. That set might depend on events at runtime, and it might change as new methods and classes are implemented in the future.

The second chance offered by a `forwardInvocation:` message provides a less ad hoc solution to this problem, and one that's dynamic rather than static. It works like this: When an object can't respond to a message because it doesn't have a method matching the selector in the message, the runtime system informs the object by sending it a `forwardInvocation:` message. Every object inherits a `forwardInvocation:` method from the `NSObject` class. However, `NSObject`'s version of the method simply invokes `doesNotRecognizeSelector:`. By overriding `NSObject`'s version and implementing your own, you can take advantage of the opportunity that the `forwardInvocation:` message provides to forward messages to other objects.

To forward a message, all a `forwardInvocation:` method needs to do is:

- Determine where the message should go, and
- Send it there with its original arguments.

The message can be sent with the `invokeWithTarget:` method:

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    if ([someOtherObject respondsToSelector:
            [anInvocation selector]])
        [anInvocation invokeWithTarget:someOtherObject];
    else
        [super forwardInvocation:anInvocation];
}
```

The return value of the message that's forwarded is returned to the original sender. All types of return values can be delivered to the sender, including `id`s, structures, and double-precision floating-point numbers.

A `forwardInvocation:` method can act as a distribution center for unrecognized messages, parceling them out to different receivers. Or it can be a transfer station, sending all messages to the same destination. It can translate one message into another, or simply "swallow" some messages so there's no response and no error. A `forwardInvocation:` method can also consolidate several messages into a single response. What `forwardInvocation:` does is up to the implementor. However, the opportunity it provides for linking objects in a forwarding chain opens up possibilities for program design.

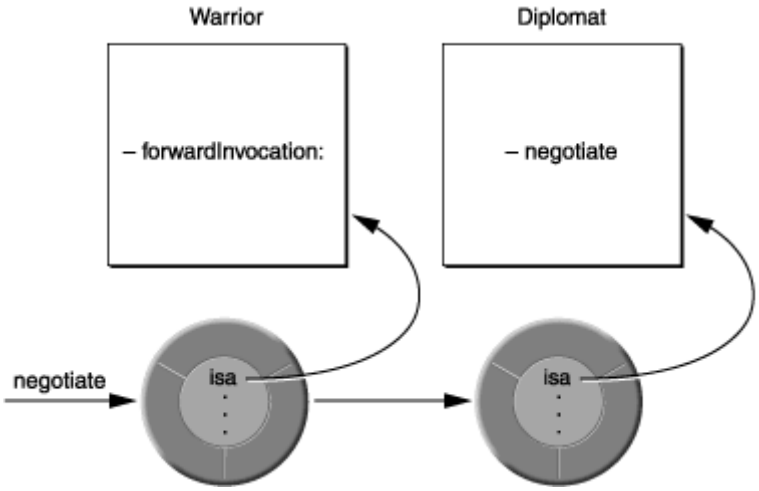
Note: The `forwardInvocation:` method gets to handle messages only if they don't invoke an existing method in the nominal receiver. If, for example, you want your object to forward `negotiate` messages to another object, it can't have a `negotiate` method of its own. If it does, the message will never reach `forwardInvocation:`.

For more information on forwarding and invocations, see the `NSInvocation` class specification in the Foundation framework reference.

Forwarding and Multiple Inheritance

Forwarding mimics inheritance, and can be used to lend some of the effects of multiple inheritance to Objective-C programs. As shown in Figure 5-1, an object that responds to a message by forwarding it appears to borrow or "inherit" a method implementation defined in another class.

Figure 5-1 Forwarding



In this illustration, an instance of the Warrior class forwards a `negotiate` message to an instance of the Diplomat class. The Warrior will appear to negotiate like a Diplomat. It will seem to respond to the `negotiate` message, and for all practical purposes it does respond (although it's really a Diplomat that's doing the work).

The object that forwards a message thus "inherits" methods from two branches of the inheritance hierarchy—its own branch and that of the object that responds to the message. In the example above, it appears as if the Warrior class inherits from Diplomat as well as its own superclass.

Forwarding provides most of the features that you typically want from multiple inheritance. However, there's an important difference between the two: Multiple inheritance combines different capabilities in a single object. It tends toward large, multifaceted objects. Forwarding, on the other hand, assigns separate responsibilities to disparate objects. It decomposes problems into smaller objects, but associates those objects in a way that's transparent to the message sender.

Surrogate Objects

Forwarding not only mimics multiple inheritance, it also makes it possible to develop lightweight objects that represent or "cover" more substantial objects. The surrogate stands in for the other object and funnels messages to it.

The proxy discussed in "Remote Messaging" in *The Objective-C Programming Language* is such a surrogate. A proxy takes care of the administrative details of forwarding messages to a remote receiver, making sure argument values are copied and retrieved across the connection, and so on. But it doesn't attempt to do much else; it doesn't duplicate the functionality of the remote object but simply gives the remote object a local address, a place where it can receive messages in another application.

Other kinds of surrogate objects are also possible. Suppose, for example, that you have an object that manipulates a lot of data—perhaps it creates a complicated image or reads the contents of a file on disk. Setting this object up could be time-consuming, so you prefer to do it lazily—when it's really needed or when system resources are temporarily idle. At the same time, you need at least a placeholder for this object in order for the other objects in the application to function properly.

In this circumstance, you could initially create, not the full-fledged object, but a lightweight surrogate for it. This object could do some things on its own, such as answer questions about the data, but mostly it would just hold a place for the larger object and, when the time came, forward messages to it. When the surrogate's `forwardInvocation:` method first receives a message destined for the other object, it would ensure that the object existed and would create it if it didn't. All messages for the larger object go through the surrogate, so, as far as the rest of the program is concerned, the surrogate and the larger object would be the same.

Forwarding and Inheritance

Although forwarding mimics inheritance, the `NSObject` class never confuses the two. Methods like `respondsToSelector:` and `isKindOfClass:` look only at the inheritance hierarchy, never at the forwarding chain. If, for example, a Warrior object is asked whether it responds to a `negotiate` message,

```
if ( [aWarrior respondsToSelector:@selector(negotiate)] )
    ...
```

the answer is `NO`, even though it can receive `negotiate` messages without error and respond to them, in a sense, by forwarding them to a Diplomat. (See Figure 5-1.)

In many cases, `NO` is the right answer. But it may not be. If you use forwarding to set up a surrogate object or to extend the capabilities of a class, the forwarding mechanism should probably be as transparent as inheritance. If you want your objects to act as if they truly inherited the behavior of the objects they forward messages to, you'll need to re-implement the `respondsToSelector:` and `isKindOfClass:` methods to include your forwarding algorithm:

```
- (BOOL)respondsToSelector:(SEL)aSelector
{
    if ( [super respondsToSelector:aSelector] )
        return YES;
    else {
        /* Here, test whether the aSelector message can     *
         * be forwarded to another object and whether that  *
         * object can respond to it. Return YES if it can.  */
    }
    return NO;
}
```

In addition to `respondsToSelector:` and `isKindOfClass:`, the `instancesRespondToSelector:` method should also mirror the forwarding algorithm. If protocols are used, the `conformsToProtocol:` method should likewise be added to the list. Similarly, if an object forwards any remote messages it receives, it should have a version of `methodSignatureForSelector:` that can return accurate descriptions of the methods that ultimately respond to the forwarded messages; for example, if an object is able to forward a message to its surrogate, you would implement `methodSignatureForSelector:` as follows:

```
- (NSMethodSignature*)methodSignatureForSelector:(SEL)selector
{
    NSMethodSignature* signature = [super methodSignatureForSelector:selector];
    if (!signature) {
       signature = [surrogate methodSignatureForSelector:selector];
    }
    return signature;
}
```

You might consider putting the forwarding algorithm somewhere in private code and have all these methods, `forwardInvocation:` included, call it.

Note: This is an advanced technique, suitable only for situations where no other solution is possible. It is not intended as a replacement for inheritance. If you must make use of this

technique, make sure you fully understand the behavior of the class doing the forwarding and the class you're forwarding to.

The methods mentioned in this section are described in the `NSObject` class specification in the Foundation framework reference. For information on `invokeWithTarget:`, see the `NSInvocation` class specification in the Foundation framework reference.
