Messaging

This chapter describes how the message expressions are converted into `objc_msgSend` function calls, and how you can refer to methods by name. It then explains how you can take advantage of `objc_msgSend`, and how—if you need to—you can circumvent dynamic binding.

The objc_msgSend Function

In Objective-C, messages aren't bound to method implementations until runtime. The compiler converts a message expression,

```
[receiver message]
```

into a call on a messaging function, `objc_msgSend`. This function takes the receiver and the name of the method mentioned in the message—that is, the method selector—as its two principal parameters:

```
objc_msgSend(receiver, selector)
```

Any arguments passed in the message are also handed to `objc_msgSend`:

```
objc_msgSend(receiver, selector, arg1, arg2, ...)
```

The messaging function does everything necessary for dynamic binding:

- It first finds the procedure (method implementation) that the selector refers to. Since the same method can be implemented differently by separate classes, the precise procedure that it finds depends on the class of the receiver.
- It then calls the procedure, passing it the receiving object (a pointer to its data), along with any arguments that were specified for the method.
- Finally, it passes on the return value of the procedure as its own return value.

> **Note:** The compiler generates calls to the messaging function. You should never call it directly in the code you write.

The key to messaging lies in the structures that the compiler builds for each class and object. Every class structure includes these two essential elements:

- A pointer to the superclass.
- A class *dispatch table*. This table has entries that associate method selectors with the class-specific addresses of the methods they identify. The selector for the `setOrigin::` method is associated with the address of (the procedure that implements) `setOrigin::`, the selector for the `display` method is associated with `display`'s address, and so on.

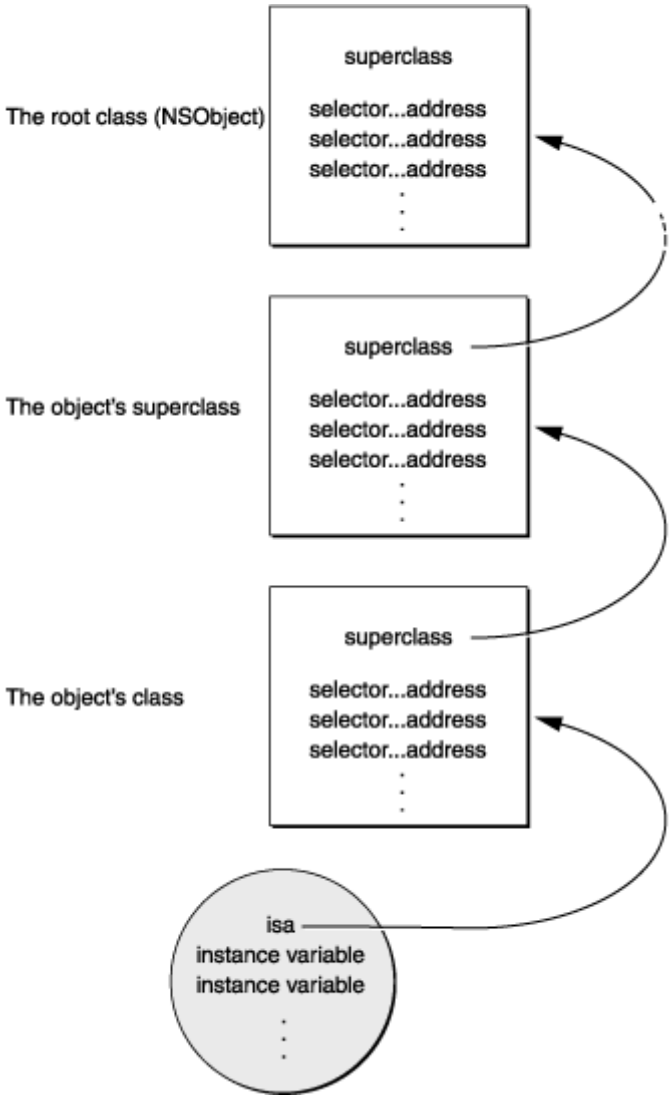
When a new object is created, memory for it is allocated, and its instance variables are initialized. First among the object's variables is a pointer to its class structure. This pointer, called `isa`, gives the object access to its class and, through the class, to all the classes it inherits from.

> **Note:** While not strictly a part of the language, the `isa` pointer is required for an object to work with the Objective-C runtime system. An object needs to be "equivalent" to a `struct objc_object` (defined in `objc/objc.h`) in whatever fields the structure defines. However, you

rarely, if ever, need to create your own root object, and objects that inherit from `NSObject` or `NSProxy` automatically have the `isa` variable.

These elements of class and object structure are illustrated in Figure 3-1.

Figure 3-1 Messaging Framework



When a message is sent to an object, the messaging function follows the object's `isa` pointer to the class structure where it looks up the method selector in the dispatch table. If it can't find the selector there, `objc_msgSend` follows the pointer to the superclass and tries to find the selector in its dispatch table. Successive failures cause `objc_msgSend` to climb the class hierarchy until it reaches the `NSObject` class. Once it locates the selector, the function calls the method entered in the table and passes it the receiving object's data structure.

This is the way that method implementations are chosen at runtime—or, in the jargon of object-oriented programming, that methods are dynamically bound to messages.

To speed the messaging process, the runtime system caches the selectors and addresses of methods as they are used. There's a separate cache for each class, and it can contain selectors for inherited methods as well as for methods defined in the class. Before searching the dispatch tables, the messaging routine first checks the cache of the receiving object's class (on the theory that a method that was used once may likely be used again). If the method selector is in the cache, messaging is only slightly slower than a function call. Once a program has been running long enough to "warm up" its caches, almost all the messages it sends find a cached method. Caches grow dynamically to accommodate new messages as the program runs.

Using Hidden Arguments

When `objc_msgSend` finds the procedure that implements a method, it calls the procedure and passes it all the arguments in the message. It also passes the procedure two hidden arguments:

- The receiving object
- The selector for the method

These arguments give every method implementation explicit information about the two halves of the message expression that invoked it. They're said to be "hidden" because they aren't declared in the source code that defines the method. They're inserted into the implementation when the code is compiled.

Although these arguments aren't explicitly declared, source code can still refer to them (just as it can refer to the receiving object's instance variables). A method refers to the receiving object as `self`, and to its own selector as `_cmd`. In the example below, `_cmd` refers to the selector for the `strange` method and `self` to the object that receives a `strange` message.

```
- strange
{
    id  target = getTheReceiver();
    SEL method = getTheMethod();

    if ( target == self || method == _cmd )
        return nil;
    return [target performSelector:method];
}
```

`self` is the more useful of the two arguments. It is, in fact, the way the receiving object's instance variables are made available to the method definition.

Getting a Method Address

The only way to circumvent dynamic binding is to get the address of a method and call it directly as if it were a function. This might be appropriate on the rare occasions when a particular method will be performed many times in succession and you want to avoid the overhead of messaging each time the method is performed.

With a method defined in the `NSObject` class, `methodForSelector:`, you can ask for a pointer to the procedure that implements a method, then use the pointer to call the procedure. The pointer that `methodForSelector:` returns must be carefully cast to the proper function type. Both return and argument types should be included in the cast.

The example below shows how the procedure that implements the `setFilled:` method might be called:

```
void (*setter)(id, SEL, BOOL);
int i;

setter = (void (*)(id, SEL, BOOL))[target
    methodForSelector:@selector(setFilled:)];
for ( i = 0 ; i < 1000 ; i++ )
    setter(targetList[i], @selector(setFilled:), YES);
```

The first two arguments passed to the procedure are the receiving object (`self`) and the method selector (`_cmd`). These arguments are hidden in method syntax but must be made explicit when the method is called as a function.

Using `methodForSelector:` to circumvent dynamic binding saves most of the time required by messaging. However, the savings will be significant only where a particular message is repeated many times, as in the `for` loop shown above.

Note that `methodForSelector:` is provided by the Cocoa runtime system; it's not a feature of the Objective-C language itself.
