Objective-C Runtime Programming Guide

Developer

Contents

Figures and Tables

SwiftObjective-C

Introduction

The Objective-C language defers as many decisions as it can from compile time and link time to runtime. Whenever possible, it does things dynamically. This means that the language requires not just a compiler, but also a runtime system to execute the compiled code. The runtime system acts as a kind of operating system for the Objective-C language; it's what makes the language work.

This document looks at the `NSObject` class and how Objective-C programs interact with the runtime system. In particular, it examines the paradigms for dynamically loading new classes at runtime, and forwarding messages to other objects. It also provides information about how you can find information about objects while your program is running.

You should read this document to gain an understanding of how the Objective-C runtime system works and how you can take advantage of it. Typically, though, there should be little reason for you to need to know and understand this material to write a Cocoa application.

Organization of This Document

This document has the following chapters:

See Also

Objective-C Runtime Reference describes the data structures and functions of the Objective-C runtime support library. Your programs can use these interfaces to interact with the Objective-C runtime system. For example, you can add classes or methods, or obtain a list of all class definitions for loaded classes.

Programming with Objective-C describes the Objective-C language.

Objective-C Release Notes describes some of the changes in the Objective-C runtime in recent releases of OS X.

Runtime Versions and Platforms

There are different versions of the Objective-C runtime on different platforms.

Legacy and Modern Versions

There are two versions of the Objective-C runtime—"modern" and "legacy". The modern version was introduced with Objective-C 2.0 and includes a number of new features. The programming interface for the legacy version of the runtime is described in *Objective-C 1 Runtime Reference*; the programming interface for the modern version of the runtime is described in *Objective-C Runtime Reference*.

The most notable new feature is that instance variables in the modern runtime are "non-fragile":

- In the legacy runtime, if you change the layout of instance variables in a class, you must recompile classes that inherit from it.
- In the modern runtime, if you change the layout of instance variables in a class, you do not have to recompile classes that inherit from it.

In addition, the modern runtime supports instance variable synthesis for declared properties (see Declared Properties in *The Objective-C Programming Language*).

Platforms

iPhone applications and 64-bit programs on OS X v10.5 and later use the modern version of the runtime.

Other programs (32-bit programs on OS X desktop) use the legacy version of the runtime.

Interacting with the Runtime

Objective-C programs interact with the runtime system at three distinct levels: through Objective-C source code; through methods defined in the `NSObject` class of the Foundation framework; and through direct calls to runtime functions.

Objective-C Source Code

For the most part, the runtime system works automatically and behind the scenes. You use it just by writing and compiling Objective-C source code.

When you compile code containing Objective-C classes and methods, the compiler creates the data structures and function calls that implement the dynamic characteristics of the language. The data structures capture information found in class and category definitions and in protocol declarations; they include the class and protocol objects discussed in Defining a Class and Protocols in *The Objective-C Programming Language*, as well as method selectors, instance variable templates, and other information distilled from source code. The principal runtime function is the one that sends messages, as described in Messaging (page 11). It's invoked by source-code message expressions.

NSObject Methods

Most objects in Cocoa are subclasses of the `NSObject` class, so most objects inherit the methods it defines. (The notable exception is the `NSProxy` class; see Message Forwarding (page 19) for more information.) Its methods therefore establish behaviors that are inherent to every instance and every class object. However, in a few cases, the `NSObject` class merely defines a template for how something should be done; it doesn't provide all the necessary code itself.

For example, the `NSObject` class defines a `description` instance method that returns a string describing the contents of the class. This is primarily used for debugging—the GDB `print-object` command prints the string returned from this method. `NSObject`'s implementation of this method doesn't know what the class contains, so it returns a string with the name and address of the object. Subclasses of `NSObject` can implement this method to return more details. For example, the Foundation class `NSArray` returns a list of descriptions of the objects it contains.

Some of the `NSObject` methods simply query the runtime system for information. These methods allow objects to perform introspection. Examples of such methods are the `class` method, which asks an object to identify its class; `isKindOfClass:` and `isMemberOfClass:`, which test an object's position in the inheritance hierarchy; `respondsToSelector:`, which indicates whether an object can accept a particular message; `conformsToProtocol:`, which indicates whether an object claims to implement the methods defined in a specific protocol; and `methodForSelector:`, which provides the address of a method's implementation. Methods like these give an object the ability to introspect about itself.

Runtime Functions

The runtime system is a dynamic shared library with a public interface consisting of a set of functions and data structures in the header files located within the directory `/usr/include/objc`. Many of these functions allow you to use plain C to replicate what the compiler does when you write Objective-C code. Others form the basis for functionality exported through the methods of the `NSObject` class. These functions make it possible to develop other interfaces to the runtime system and produce tools that augment the development environment; they're not needed when programming in Objective-C. However, a few of the runtime functions might on occasion be useful when writing an Objective-C program. All of these functions are documented in *Objective-C Runtime Reference*.

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:

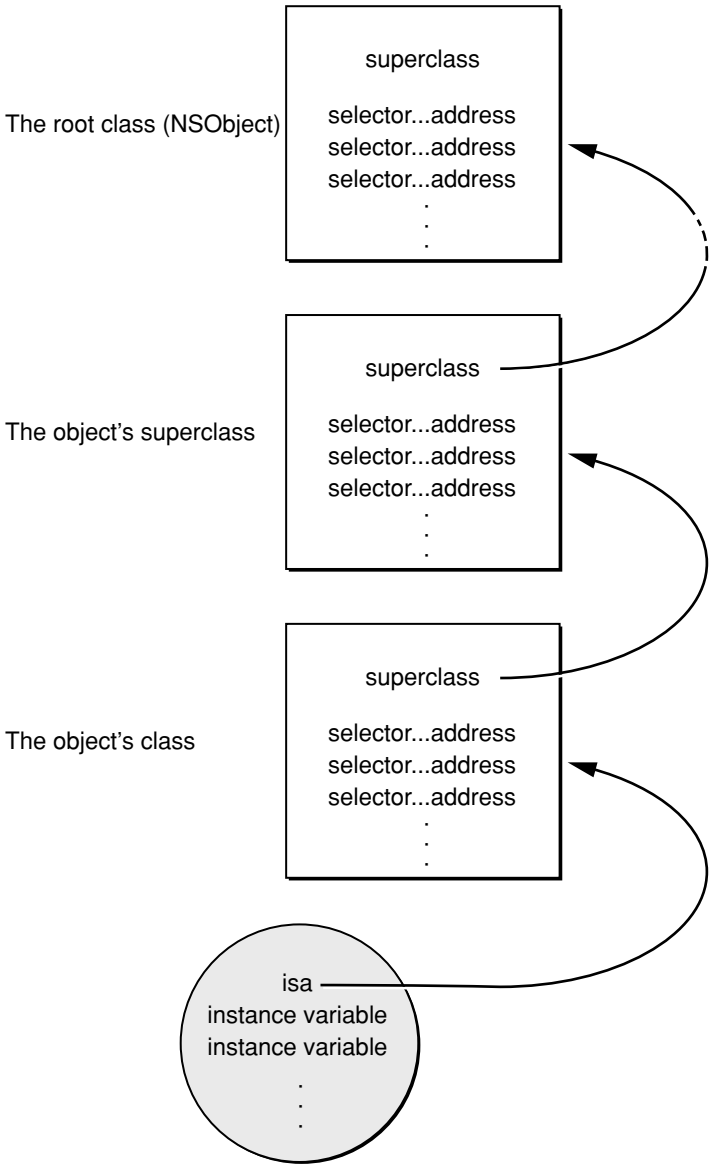
- A pointer to the superclass.
- A class *dispatch table*. This table has entries that associate method selectors with the class-specific addresses of the methods they identify. The selector for the `setOrigin::` method is associated with the address of (the procedure that implements) `setOrigin::`,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

Dynamic Method Resolution

Objective-CSwift
This chapter describes how you can provide an implementation of a method dynamically.

Dynamic Method Resolution

There are situations where you might want to provide an implementation of a method dynamically. For example, the Objective-C declared properties feature (see Declared Properties in *The Objective-C Programming Language*) includes the `@dynamic` directive:

```
@dynamic propertyName;
```

which tells the compiler that the methods associated with the property will be provided dynamically.

You can implement the methods `resolveInstanceMethod:` and `resolveClassMethod:` to dynamically provide an implementation for a given selector for an instance and class method respectively.

An Objective-C method is simply a C function that take at least two arguments—`self` and `_cmd`. You can add a function to a class as a method using the function `class_addMethod`. Therefore, given the following function:

```
void dynamicMethodIMP(id self, SEL _cmd) {
    // implementation ....
}
```

you can dynamically add it to a class as a method (called `resolveThisMethodDynamically`) using `resolveInstanceMethod:` like this:

```
@implementation MyClass
+ (BOOL)resolveInstanceMethod:(SEL)aSEL
{
    if (aSEL == @selector(resolveThisMethodDynamically)) {
          class_addMethod([self class], aSEL, (IMP) dynamicMethodIMP, "v@:");
          return YES;
```

```
    }
    return [super resolveInstanceMethod:aSEL];
}
@end
```

Forwarding methods (as described in Message Forwarding (page 19)) and dynamic method resolution are, largely, orthogonal. A class has the opportunity to dynamically resolve a method before the forwarding mechanism kicks in. If `respondsToSelector:` or `instancesRespondToSelector:` is invoked, the dynamic method resolver is given the opportunity to provide an `IMP` for the selector first. If you implement `resolveInstanceMethod:` but want particular selectors to actually be forwarded via the forwarding mechanism, you return `NO` for those selectors.

Dynamic Loading

An Objective-C program can load and link new classes and categories while it's running. The new code is incorporated into the program and treated identically to classes and categories loaded at the start.

Dynamic loading can be used to do a lot of different things. For example, the various modules in the System Preferences application are dynamically loaded.

In the Cocoa environment, dynamic loading is commonly used to allow applications to be customized. Others can write modules that your program loads at runtime—much as Interface Builder loads custom palettes and the OS X System Preferences application loads custom preference modules. The loadable modules extend what your application can do. They contribute to it in ways that you permit but could not have anticipated or defined yourself. You provide the framework, but others provide the code.

Although there is a runtime function that performs dynamic loading of Objective-C modules in Mach-O files (`objc_loadModules`, defined in `objc/objc-load.h`), Cocoa's `NSBundle` class provides a significantly more convenient interface for dynamic loading—one that's object-oriented and integrated with related services. See the `NSBundle` class specification in the Foundation framework reference for information on the `NSBundle` class and its use. See *OS X ABI Mach-O File Format Reference* for information on Mach-O files.

Message Forwarding

SwiftObjective-C

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.

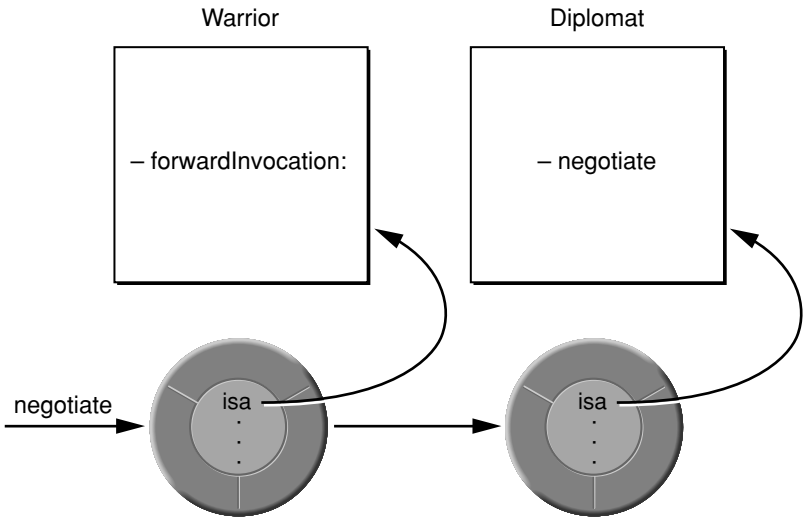
> **Note:** The `forwardInvocation:` method gets to handle messages only if they don't invoke an existing method in the nominal receiver. If, for example, you want your object to forward `negotiate` messages to another object, it can't have a `negotiate` method of its own. If it does, the message will 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 (page 21), 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 (page 21).)

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.

Type Encodings

To assist the runtime system, the compiler encodes the return and argument types for each method in a character string and associates the string with the method selector. The coding scheme it uses is also useful in other contexts and so is made publicly available with the `@encode()` compiler directive. When given a type specification, `@encode()` returns a string encoding that type. The type can be a basic type such as an `int`, a pointer, a tagged structure or union, or a class name—any type, in fact, that can be used as an argument to the C `sizeof()` operator.

```
char *buf1 = @encode(int **);
char *buf2 = @encode(struct key);
char *buf3 = @encode(Rectangle);
```

The table below lists the type codes. Note that many of them overlap with the codes you use when encoding an object for purposes of archiving or distribution. However, there are codes listed here that you can't use when writing a coder, and there are codes that you may want to use when writing a coder that aren't generated by `@encode()`. (See the `NSCoder` class specification in the Foundation Framework reference for more information on encoding objects for archiving or distribution.)

Table 6-1 Objective-C type encodings

Code	Meaning
`c`	A `char`
`i`	An `int`
`s`	A `short`
`l`	A `long` `l` is treated as a 32-bit quantity on 64-bit programs.
`q`	A `long long`
`C`	An `unsigned char`
`I`	An `unsigned int`
`S`	An `unsigned short`

Code	Meaning
`L`	An `unsigned long`
`Q`	An `unsigned long long`
`f`	A `float`
`d`	A `double`
`B`	A C++ `bool` or a C99 `_Bool`
`v`	A `void`
`*`	A character string (`char *`)
`@`	An object (whether statically typed or typed `id`)
`#`	A class object (`Class`)
`:`	A method selector (`SEL`)
[*array type*]	An array
{*name*=*type*...}	A structure
(*name*=*type*...)	A union
`b`num	A bit field of *num* bits
^type	A pointer to *type*
`?`	An unknown type (among other things, this code is used for function pointers)

Important: Objective-C does not support the `long double` type. `@encode(long double)` returns `d`, which is the same encoding as for `double`.

The type code for an array is enclosed within square brackets; the number of elements in the array is specified immediately after the open bracket, before the array type. For example, an array of 12 pointers to `float`s would be encoded as:

```
[12^f]
```

Structures are specified within braces, and unions within parentheses. The structure tag is listed first, followed by an equal sign and the codes for the fields of the structure listed in sequence. For example, the structure

```
typedef struct example {
    id   anObject;
    char *aString;
    int  anInt;
} Example;
```

would be encoded like this:

```
{example=@*i}
```

The same encoding results whether the defined type name (`Example`) or the structure tag (`example`) is passed to `@encode()`. The encoding for a structure pointer carries the same amount of information about the structure's fields:

```
^{example=@*i}
```

However, another level of indirection removes the internal type specification:

```
^^{example}
```

Objects are treated like structures. For example, passing the `NSObject` class name to `@encode()` yields this encoding:

```
{NSObject=#}
```

The `NSObject` class declares just one instance variable, `isa`, of type Class.

Note that although the `@encode()` directive doesn't return them, the runtime system uses the additional encodings listed in Table 6-2 for type qualifiers when they're used to declare methods in a protocol.

Table 6-2 Objective-C method encodings

Code	Meaning
`r`	`const`

Code	Meaning
n	in
N	inout
o	out
O	bycopy
R	byref
V	oneway

Declared Properties

When the compiler encounters property declarations (see Declared Properties in *The Objective-C Programming Language*), it generates descriptive metadata that is associated with the enclosing class, category or protocol. You can access this metadata using functions that support looking up a property by name on a class or protocol, obtaining the type of a property as an `@encode` string, and copying a list of a property's attributes as an array of C strings. A list of declared properties is available for each class and protocol.

Property Type and Functions

The `Property` structure defines an opaque handle to a property descriptor.

```
typedef struct objc_property *Property;
```

You can use the functions `class_copyPropertyList` and `protocol_copyPropertyList` to retrieve an array of the properties associated with a class (including loaded categories) and a protocol respectively:

```
objc_property_t *class_copyPropertyList(Class cls, unsigned int *outCount)
objc_property_t *protocol_copyPropertyList(Protocol *proto, unsigned int *outCount)
```

For example, given the following class declaration:

```
@interface Lender : NSObject {
    float alone;
}
@property float alone;
@end
```

you can get the list of properties using:

```
id LenderClass = objc_getClass("Lender");
unsigned int outCount;
```

```
objc_property_t *properties = class_copyPropertyList(LenderClass, &outCount);
```

You can use the `property_getName` function to discover the name of a property:

```
const char *property_getName(objc_property_t property)
```

You can use the functions `class_getProperty` and `protocol_getProperty` to get a reference to a property with a given name in a class and protocol respectively:

```
objc_property_t class_getProperty(Class cls, const char *name)
objc_property_t protocol_getProperty(Protocol *proto, const char *name, BOOL
isRequiredProperty, BOOL isInstanceProperty)
```

You can use the `property_getAttributes` function to discover the name and the `@encode` type string of a property. For details of the encoding type strings, see Type Encodings (page 25); for details of this string, see Property Type String (page 30) and Property Attribute Description Examples (page 31).

```
const char *property_getAttributes(objc_property_t property)
```

Putting these together, you can print a list of all the properties associated with a class using the following code:

```
id LenderClass = objc_getClass("Lender");
unsigned int outCount, i;
objc_property_t *properties = class_copyPropertyList(LenderClass, &outCount);
for (i = 0; i < outCount; i++) {
    objc_property_t property = properties[i];
    fprintf(stdout, "%s %s\n", property_getName(property),
property_getAttributes(property));
}
```

Property Type String

You can use the `property_getAttributes` function to discover the name, the `@encode` type string of a property, and other attributes of the property.

The string starts with a `T` followed by the `@encode` type and a comma, and finishes with a `V` followed by the name of the backing instance variable. Between these, the attributes are specified by the following descriptors, separated by commas:

Table 7-1 Declared property type encodings

Code	Meaning
`R`	The property is read-only (`readonly`).
`C`	The property is a copy of the value last assigned (`copy`).
`&`	The property is a reference to the value last assigned (`retain`).
`N`	The property is non-atomic (`nonatomic`).
`G<name>`	The property defines a custom getter selector name. The name follows the `G` (for example, `GcustomGetter,`).
`S<name>`	The property defines a custom setter selector name. The name follows the `S` (for example, `ScustomSetter:,`).
`D`	The property is dynamic (`@dynamic`).
`W`	The property is a weak reference (`__weak`).
`P`	The property is eligible for garbage collection.
`t<encoding>`	Specifies the type using old-style encoding.

For examples, see Property Attribute Description Examples (page 31).

Property Attribute Description Examples

Given these definitions:

```
enum FooManChu { FOO, MAN, CHU };
struct YorkshireTeaStruct { int pot; char lady; };
typedef struct YorkshireTeaStruct YorkshireTeaStructType;
union MoneyUnion { float alone; double down; };
```

the following table shows sample property declarations and the corresponding string returned by `property_getAttributes`:

Property declaration	Property description
`@property char charDefault;`	`Tc,VcharDefault`
`@property double doubleDefault;`	`Td,VdoubleDefault`
`@property enum FooManChu enumDefault;`	`Ti,VenumDefault`
`@property float floatDefault;`	`Tf,VfloatDefault`
`@property int intDefault;`	`Ti,VintDefault`
`@property long longDefault;`	`Tl,VlongDefault`
`@property short shortDefault;`	`Ts,VshortDefault`
`@property signed signedDefault;`	`Ti,VsignedDefault`
`@property struct YorkshireTeaStruct structDefault;`	`T{YorkshireTea-Struct="pot"i"lady"c},Vstruct-Default`
`@property YorkshireTeaStructType typedefDefault;`	`T{YorkshireTea-Struct="pot"i"lady"c},Vtypedef-Default`
`@property union MoneyUnion unionDefault;`	`T(Money-Union="alone"f"down"d),Vunion-Default`
`@property unsigned unsignedDefault;`	`TI,VunsignedDefault`
`@property int (*functionPointerDefault)(char *);`	`T^?,VfunctionPointerDefault`
`@property id idDefault;` Note: the compiler warns: `"no 'assign', 'retain', or 'copy' attribute is specified - 'assign' is assumed"`	`T@,VidDefault`
`@property int *intPointer;`	`T^i,VintPointer`
`@property void *voidPointerDefault;`	`T^v,VvoidPointerDefault`

Property declaration	Property description

`@property int intSynthEquals;`
In the implementation block:
`@synthesize intSynthEquals=_intSynthEquals;`
	`Ti,V_intSynthEquals`

`@property(getter=intGetFoo, setter=intSetFoo:) int intSetterGetter;`	`Ti,GintGetFoo,SintSetFoo: ,VintSetterGetter`
`@property(readonly) int intReadonly;`	`Ti,R,VintReadonly`
`@property(getter=isIntReadOnlyGetter, readonly) int intReadonlyGetter;`	`Ti,R,GisIntReadOnlyGetter`
`@property(readwrite) int intReadwrite;`	`Ti,VintReadwrite`
`@property(assign) int intAssign;`	`Ti,VintAssign`
`@property(retain)ididRetain;`	`T@,&,VidRetain`
`@property(copy)ididCopy;`	`T@,C,VidCopy`
`@property(nonatomic) int intNonatomic;`	`Ti,VintNonatomic`
`@property(nonatomic, readonly, copy) id idReadonlyCopyNonatomic;`	`T@,R,C,VidReadonlyCopyNonatomic`
`@property(nonatomic, readonly, retain) id idReadonlyRetainNonatomic;`	`T@,R,&,VidReadonlyRetain-Nonatomic`

Document Revision History

This table describes the changes to *Objective-C Runtime Programming Guide*.

Date	Notes
2009-10-19	Made minor editorial changes.
2009-07-14	Completed list of types described by property_getAttributes.
2009-02-04	Corrected typographical errors.
2008-11-19	New document that describes the Objective-C 2.0 runtime support library.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010