# NSObject Class Reference

**Cocoa > Objective-C Language**

**2009-02-04**

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



Simultaneously published in the United States and Canada.

# Contents


## NSObject Class Reference 7

Overview 7
- Selectors 7

Adopted Protocols 9

Tasks 9
- Initializing a Class 9
- Creating, Copying, and Deallocating Objects 9
- Identifying Classes 10
- Testing Class Functionality 10
- Testing Protocol Conformance 10
- Obtaining Information About Methods 11
- Describing Objects 11
- Posing 11
- Sending Messages 11
- Forwarding Messages 12
- Dynamically Resolving Methods 12
- Error Handling 12
- Archiving 12
- Working with Class Descriptions 13
- Scripting 13

Class Methods 14
- alloc 14
- allocWithZone: 14
- cancelPreviousPerformRequestsWithTarget: 15
- cancelPreviousPerformRequestsWithTarget:selector:object: 16
- class 17
- classFallbacksForKeyedArchiver 17
- classForKeyedUnarchiver 18
- conformsToProtocol: 18
- copyWithZone: 19
- description 19
- initialize 20
- instanceMethodForSelector: 21
- instanceMethodSignatureForSelector: 22
- instancesRespondToSelector: 23
- isSubclassOfClass: 23
- load 23
- mutableCopyWithZone: 24
- new 25
- resolveClassMethod: 26
- resolveInstanceMethod: 26

setVersion: 27
superclass 28
version 28
Instance Methods 29
attributeKeys 29
awakeAfterUsingCoder: 30
classCode 31
classDescription 31
classForArchiver 32
classForCoder 32
classForKeyedArchiver 32
classForPortCoder 33
className 33
copy 34
copyScriptingValue:forKey:withProperties: 34
dealloc 35
doesNotRecognizeSelector: 36
finalize 37
forwardInvocation: 38
init 40
inverseForRelationshipKey: 42
methodForSelector: 42
methodSignatureForSelector: 43
mutableCopy 44
newScriptingObjectOfClass:forValueForKey:withContentsValue:properties: 44
performSelector:onThread:withObject:waitUntilDone: 45
performSelector:onThread:withObject:waitUntilDone:modes: 46
performSelector:withObject:afterDelay: 47
performSelector:withObject:afterDelay:inModes: 48
performSelectorInBackground:withObject: 49
performSelectorOnMainThread:withObject:waitUntilDone: 50
performSelectorOnMainThread:withObject:waitUntilDone:modes: 51
replacementObjectForArchiver: 52
replacementObjectForCoder: 53
replacementObjectForKeyedArchiver: 53
replacementObjectForPortCoder: 54
scriptingProperties 54
scriptingValueForSpecifier: 55
setScriptingProperties: 55
toManyRelationshipKeys 56
toOneRelationshipKeys 56

Appendix A Deprecated NSObject Methods 59

Deprecated in Mac OS X v10.5 59
poseAsClass: 59

Document Revision History 61

Index 63

# NSObject Class Reference

| | |
|---|---|
| **Inherits from** | none (NSObject is a root class) |
| **Conforms to** | NSObject |
| **Framework** | /System/Library/Frameworks/Foundation.framework |
| **Availability** | Available in Mac OS X v10.0 and later. |
| **Companion guide** | Cocoa Fundamentals Guide |
| **Declared in** | NSArchiver.h<br>NSClassDescription.h<br>NSKeyedArchiver.h<br>NSObject.h<br>NSObjectScripting.h<br>NSPortCoder.h<br>NSRunLoop.h<br>NSScriptClassDescription.h<br>NSThread.h |
| **Related sample code** | CoreRecipes<br>Dicey<br>ImageClient<br>Quartz Composer WWDC 2005 TextEdit<br>StickiesExample |

## Overview

`NSObject` is the root class of most Objective-C class hierarchies. Through `NSObject`, objects inherit a basic interface to the runtime system and the ability to behave as Objective-C objects.

### Selectors

`NSObject` has some special methods that take advantage of the Objective-C runtime system. For example, you can ask a class or instance if it responds to a message before sending it a message. You can also ask for a method implementation and invoke it using one of the `perform...` methods, or as a function. The advantage of obtaining a method's implementation and calling it as a function is that you can invoke the implementation multiple times within a loop, or similar C construct, without the overhead of Objective-C messaging.

These and other `NSObject` methods take a selector of type `SEL` as an argument. For efficiency, full ASCII names are not used to represent methods in compiled code. Instead the compiler uses a unique identifier to represent a method at runtime called a **selector**. A selector for a method name is obtained using the `@selector()` directive:

```
SEL method = @selector(isEqual:);
```

The `instanceMethodForSelector:` (page 21) class method and the `methodForSelector:` (page 42) instance method return a method implementation of type `IMP`. `IMP` is defined as a pointer to a function that returns an `id` and takes a variable number of arguments (in addition to the two "hidden" arguments—`self` and `_cmd`—that are passed to every method implementation):

```
typedef id (*IMP)(id, SEL, ...);
```

This definition serves as a prototype for the function pointer returned by these methods. It's sufficient for methods that return an object and take object arguments. However, if the selector takes different argument types or returns anything but an `id`, its function counterpart will be inadequately prototyped. Lacking a prototype, the compiler will promote floats to doubles and chars to ints, which the implementation won't expect. It will therefore behave differently (and erroneously) when performed as a method.

To remedy this situation, it's necessary to provide your own prototype. In the example below, the declaration of the `test` variable serves to prototype the implementation of the `isEqual:` method. `test` is defined as a pointer to a function that returns a `BOOL` and takes an `id` argument (in addition to the two "hidden" arguments). The value returned by `methodForSelector:` (page 42) is then similarly cast to be a pointer to this same function type:

```
BOOL (*test)(id, SEL, id);
test = (BOOL (*)(id, SEL, id))[target methodForSelector:@selector(isEqual:)];

while ( !test(target, @selector(isEqual:), someObject) ) {
    ...
}
```

In some cases, it might be clearer to define a type (similar to `IMP`) that can be used both for declaring the variable and for casting the function pointer `methodForSelector:` (page 42) returns. The example below defines the `EqualIMP` type for just this purpose:

```
typedef BOOL (*EqualIMP)(id, SEL, id);
EqualIMP test;
test = (EqualIMP)[target methodForSelector:@selector(isEqual:)];

while ( !test(target, @selector(isEqual:), someObject) ) {
    ...
}
```

Either way, it's important to cast the return value of `methodForSelector:` (page 42) to the appropriate function type. It's not sufficient to simply call the function returned by `methodForSelector:` and cast the result of that call to the desired type. Doing so can result in errors.

See "How Messaging Works" in *The Objective-C 2.0 Programming Language* for more information.

# Adopted Protocols

NSObject
- autorelease
- class
- conformsToProtocol:
- description
- hash
- isEqual:
- isKindOfClass:
- isMemberOfClass:
- isProxy
- performSelector:
- performSelector:withObject:
- performSelector:withObject:withObject:
- release
- respondsToSelector:
- retain
- retainCount
- self
- superclass
- zone

# Tasks

## Initializing a Class

+ initialize (page 20)
Initializes the receiver before it's used (before it receives its first message).

+ load (page 23)
Invoked whenever a class or category is added to the Objective-C runtime; implement this method to perform class-specific behavior upon loading.

## Creating, Copying, and Deallocating Objects

+ new (page 25)
Allocates a new instance of the receiving class, sends it an init (page 40) message, and returns the initialized object.

+ alloc (page 14)
Returns a new instance of the receiving class.

+ `allocWithZone:` (page 14)
  Returns a new instance of the receiving class where memory for the new instance is allocated from a given zone.

- `init` (page 40)
  Implemented by subclasses to initialize a new object (the receiver) immediately after memory for it has been allocated.

- `copy` (page 34)
  Returns the object returned by `copyWithZone:`, where the zone is `nil`.

+ `copyWithZone:` (page 19)
  Returns the receiver.

- `mutableCopy` (page 44)
  Returns the object returned by `mutableCopyWithZone:` where the zone is `nil`.

+ `mutableCopyWithZone:` (page 24)
  Returns the receiver.

- `dealloc` (page 35)
  Deallocates the memory occupied by the receiver.

- `finalize` (page 37)
  The garbage collector invokes this method on the receiver before disposing of the memory it uses.

## Identifying Classes

+ `class` (page 17)
  Returns the class object.

+ `superclass` (page 28)
  Returns the class object for the receiver's superclass.

+ `isSubclassOfClass:` (page 23)
  Returns a Boolean value that indicates whether the receiving class is a subclass of, or identical to, a given class.

## Testing Class Functionality

+ `instancesRespondToSelector:` (page 23)
  Returns a Boolean value that indicates whether instances of the receiver are capable of responding to a given selector.

## Testing Protocol Conformance

+ `conformsToProtocol:` (page 18)
  Returns a Boolean value that indicates whether the receiver conforms to a given protocol.

## Obtaining Information About Methods

- `methodForSelector:` (page 42)
  Locates and returns the address of the receiver's implementation of a method so it can be called as a function.
+ `instanceMethodForSelector:` (page 21)
  Locates and returns the address of the implementation of the instance method identified by a given selector.
+ `instanceMethodSignatureForSelector:` (page 22)
  Returns an `NSMethodSignature` object that contains a description of the instance method identified by a given selector.
- `methodSignatureForSelector:` (page 43)
  Returns an `NSMethodSignature` object that contains a description of the method identified by a given selector.

## Describing Objects

+ `description` (page 19)
  Returns a string that represents the contents of the receiving class.

## Posing

+ `poseAsClass:` (page 59) Deprecated in Mac OS X v10.5
  Causes the receiving class to pose as a specified superclass.

## Sending Messages

- `performSelector:withObject:afterDelay:` (page 47)
  Invokes a method of the receiver on the current thread using the default mode after a delay.
- `performSelector:withObject:afterDelay:inModes:` (page 48)
  Invokes a method of the receiver on the current thread using the specified modes after a delay.
- `performSelectorOnMainThread:withObject:waitUntilDone:` (page 50)
  Invokes a method of the receiver on the main thread using the default mode.
- `performSelectorOnMainThread:withObject:waitUntilDone:modes:` (page 51)
  Invokes a method of the receiver on the main thread using the specified modes.
- `performSelector:onThread:withObject:waitUntilDone:` (page 45)
  Invokes a method of the receiver on the specified thread using the default mode.
- `performSelector:onThread:withObject:waitUntilDone:modes:` (page 46)
  Invokes a method of the receiver on the specified thread using the specified modes.
- `performSelectorInBackground:withObject:` (page 49)
  Invokes a method of the receiver on a new background thread.
+ `cancelPreviousPerformRequestsWithTarget:` (page 15)
  Cancels perform requests previously registered with the `performSelector:withObject:afterDelay:` (page 47) instance method.

`+ cancelPreviousPerformRequestsWithTarget:selector:object:` (page 16)
Cancels perform requests previously registered with `performSelector:withObject:afterDelay:` (page 47).

## Forwarding Messages

`- forwardInvocation:` (page 38)
Overridden by subclasses to forward messages to other objects.

## Dynamically Resolving Methods

`+ resolveClassMethod:` (page 26)
Dynamically provides an implementation for a given selector for a class method.

`+ resolveInstanceMethod:` (page 26)
Dynamically provides an implementation for a given selector for an instance method.

## Error Handling

`- doesNotRecognizeSelector:` (page 36)
Handles messages the receiver doesn't recognize.

## Archiving

`- awakeAfterUsingCoder:` (page 30)
Overridden by subclasses to substitute another object in place of the object that was decoded and subsequently received this message.

`- classForArchiver` (page 32)
Overridden by subclasses to substitute a class other than its own during archiving.

`- classForCoder` (page 32)
Overridden by subclasses to substitute a class other than its own during coding.

`- classForKeyedArchiver` (page 32)
Overridden by subclasses to substitute a new class for instances during keyed archiving.

`+ classFallbacksForKeyedArchiver` (page 17)
Overridden to return the names of classes that can be used to decode objects if their class is unavailable.

`+ classForKeyedUnarchiver` (page 18)
Overridden by subclasses to substitute a new class during keyed unarchiving.

`- classForPortCoder` (page 33)
Overridden by subclasses to substitute a class other than its own for distribution encoding.

`- replacementObjectForArchiver:` (page 52)
Overridden by subclasses to substitute another object for itself during archiving.

`- replacementObjectForCoder:` (page 53)
Overridden by subclasses to substitute another object for itself during encoding.

- `replacementObjectForKeyedArchiver:` (page 53)
  Overridden by subclasses to substitute another object for itself during keyed archiving.
- `replacementObjectForPortCoder:` (page 54)
  Overridden by subclasses to substitute another object or a copy for itself during distribution encoding.
+ `setVersion:` (page 27)
  Sets the receiver's version number.
+ `version` (page 28)
  Returns the version number assigned to the class.

## Working with Class Descriptions

- `attributeKeys` (page 29)
  Returns an array of `NSString` objects containing the names of immutable values that instances of the receiver's class contain.
- `classDescription` (page 31)
  Returns an object containing information about the attributes and relationships of the receiver's class.
- `inverseForRelationshipKey:` (page 42)
  For a given key that defines the name of the relationship from the receiver's class to another class, returns the name of the relationship from the other class to the receiver's class.
- `toManyRelationshipKeys` (page 56)
  Returns array containing the keys for the to-many relationship properties of the receiver.
- `toOneRelationshipKeys` (page 56)
  Returns the keys for the to-one relationship properties of the receiver, if any.

## Scripting

- `classCode` (page 31)
  Returns the receiver's Apple event type code, as stored in the `NSScriptClassDescription` object for the object's class.
- `className` (page 33)
  Returns a string containing the name of the class.
- `copyScriptingValue:forKey:withProperties:` (page 34)
  Creates and returns one or more scripting objects to be inserted into the specified relationship by copying the passed-in value and setting the properties in the copied object or objects.
- `newScriptingObjectOfClass:forValueForKey:withContentsValue:properties:` (page 44)
  Creates and returns an instance of a scriptable class, setting its contents and properties, for insertion into the relationship identified by the key.
- `scriptingProperties` (page 54)
  Returns an `NSString`-keyed dictionary of the receiver's scriptable properties.
- `setScriptingProperties:` (page 55)
  Given an `NSString`-keyed dictionary, sets one or more scriptable properties of the receiver.
- `scriptingValueForSpecifier:` (page 55)
  Given an object specifier, returns the specified object or objects in the receiving container.

# Class Methods

## alloc

Returns a new instance of the receiving class.

```
+ (id)alloc
```

**Return Value**
A new instance of the receiver.

**Discussion**
The `isa` instance variable of the new instance is initialized to a data structure that describes the class; memory for all other instance variables is set to `0`. The new instance is allocated from the default zone—use `allocWithZone:` (page 14) to specify a particular zone.

An `init...` method must be used to complete the initialization process. For example:

```
TheClass *newObject = [[TheClass alloc] init];
```

Subclasses shouldn't override `alloc` to include initialization code. Instead, class-specific versions of `init...` methods should be implemented for that purpose. Class methods can also be implemented to combine allocation and initialization, similar to the `new` class method.

**Special Considerations**

If you are using managed memory (not garbage collection), this method retains the object before returning it. The returned object has a retain count of `1` and is *not* autoreleased. The invoker of this method is responsible for releasing the returned object, using either `release` or `autorelease`.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`- init` (page 40)

**Related Sample Code**
CoreRecipes
GLSLShowpiece
ImageClient
iSpend
QTCoreVideo301

**Declared In**
`NSObject.h`

## allocWithZone:

Returns a new instance of the receiving class where memory for the new instance is allocated from a given zone.

```
+ (id)allocWithZone:(NSZone *)zone
```

**Parameters**

*zone*

The memory zone in which to create the new instance.

**Return Value**

A new instance of the receiver, where memory for the new instance is allocated from *zone*.

**Discussion**

The `isa` instance variable of the new instance is initialized to a data structure that describes the class; memory for its other instance variables is set to 0. If *zone* is `nil`, the new instance will be allocated from the default zone (as returned by `NSDefaultMallocZone`).

An `init...` method must be used to complete the initialization process. For example:

```
TheClass *newObject = [[TheClass allocWithZone:someZone] init];
```

Subclasses shouldn't override `allocWithZone:` to include any initialization code. Instead, class-specific versions of `init...` methods should be implemented for that purpose.

When one object creates another, it's sometimes a good idea to make sure they're both allocated from the same region of memory. The `zone` method (declared in the `NSObject` protocol) can be used for this purpose; it returns the zone where the receiver is located. For example:

```
id myCompanion = [[TheClass allocWithZone:[self zone]] init];
```

**Special Considerations**

If you are using managed memory (not garbage collection), this method retains the object before returning it. The returned object has a retain count of `1` and is *not* autoreleased. The invoker of this method is responsible for releasing the returned object, using either `release` or `autorelease`.

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**

`+ alloc` (page 14)

`- init` (page 40)

**Related Sample Code**

MenuItemView

QTCoreVideo201

Quartz Composer WWDC 2005 TextEdit

Sketch-112

TextEditPlus

**Declared In**

`NSObject.h`

## cancelPreviousPerformRequestsWithTarget:

Cancels perform requests previously registered with the `performSelector:withObject:afterDelay:` (page 47) instance method.

```
+ (void)cancelPreviousPerformRequestsWithTarget:(id)aTarget
```

**Parameters**

*aTarget*

The target for requests previously registered with the `performSelector:withObject:afterDelay:` (page 47) instance method.

**Discussion**

All perform requests having the same target *aTarget* are canceled. This method removes perform requests only in the current run loop, not all run loops.

**Availability**

Available in Mac OS X v10.2 and later.

**Declared In**

`NSRunLoop.h`

## cancelPreviousPerformRequestsWithTarget:selector:object:

Cancels perform requests previously registered with `performSelector:withObject:afterDelay:` (page 47).

```
+ (void)cancelPreviousPerformRequestsWithTarget:(id)aTarget selector:(SEL)aSelector
    object:(id)anArgument
```

**Parameters**

*aTarget*

The target for requests previously registered with the `performSelector:withObject:afterDelay:` (page 47) instance method

*aSelector*

The selector for requests previously registered with the `performSelector:withObject:afterDelay:` (page 47) instance method.

See "Selectors" (page 7) for a description of the `SEL` type.

*anArgument*

The argument for requests previously registered with the `performSelector:withObject:afterDelay:` (page 47) instance method. Argument equality is determined using `isEqual:`, so the value need not be the same object that was passed originally. Pass `nil` to match a request for `nil` that was originally passed as the argument.

**Discussion**

All perform requests are canceled that have the same target as *aTarget*, argument as *anArgument*, and selector as *aSelector*. This method removes perform requests only in the current run loop, not all run loops.

**Availability**

Available in Mac OS X v10.0 and later.

**Related Sample Code**

AttachAScript

**Declared In**

`NSRunLoop.h`

## class

Returns the class object.

```
+ (Class)class
```

**Return Value**
The class object.

**Discussion**
Refer to a class only by its name when it is the receiver of a message. In all other cases, the class object must be obtained through this or a similar method. For example, here `SomeClass` is passed as an argument to the `isKindOfClass:` method (declared in the `NSObject` protocol):

```
BOOL test = [self isKindOfClass:[SomeClass class]];
```

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`class` (`NSObject` protocol)

**Related Sample Code**
NewsReader
OpenGLCaptureToMovie
Quartz Composer WWDC 2005 TextEdit
Sketch-112
TextEditPlus

**Declared In**
`NSObject.h`

## classFallbacksForKeyedArchiver

Overridden to return the names of classes that can be used to decode objects if their class is unavailable.

```
+ (NSArray *)classFallbacksForKeyedArchiver
```

**Return Value**
An array of `NSString` objects that specify the names of classes in preferred order for unarchiving

**Discussion**
`NSKeyedArchiver` calls this method and stores the result inside the archive. If the actual class of an object doesn't exist at the time of unarchiving, `NSKeyedUnarchiver` goes through the stored list of classes and uses the first one that does exists as a substitute class for decoding the object. The default implementation of this method returns `nil`.

Developers who introduce a new class can use this method to provided some backwards compatibility in case the archive will be read on a system that does not have that class. Sometimes there may be another class which may work nearly as well as a substitute for the new class, and the archive keys and archived state for the new class can be carefully chosen (or compatibility written out) so that the object can be unarchived as the substitute class if necessary.

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`NSKeyedArchiver.h`

## classForKeyedUnarchiver

Overridden by subclasses to substitute a new class during keyed unarchiving.

```
+ (Class)classForKeyedUnarchiver
```

**Return Value**
The class to substitute for the receiver during keyed unarchiving.

**Discussion**
During keyed unarchiving, instances of the receiver will be decoded as members of the returned class. This method overrides the results of the decoder's class and instance name to class encoding tables.

**Availability**
Available in Mac OS X v10.2 and later.

**Declared In**
`NSKeyedArchiver.h`

## conformsToProtocol:

Returns a Boolean value that indicates whether the receiver conforms to a given protocol.

```
+ (BOOL)conformsToProtocol:(Protocol *)aProtocol
```

**Parameters**

*aProtocol*
A protocol.

**Return Value**
`YES` if the receiver conforms to *aProtocol*, otherwise `NO`.

**Discussion**
A class is said to "conform to" a protocol if it adopts the protocol or inherits from another class that adopts it. Protocols are adopted by listing them within angle brackets after the interface declaration. For example, here `MyClass` adopts the (fictitious) `AffiliationRequests` and `Normalization` protocols:

```
@interface MyClass : NSObject <AffiliationRequests, Normalization>
```

A class also conforms to any protocols that are incorporated in the protocols it adopts or inherits. Protocols incorporate other protocols in the same way classes adopt them. For example, here the `AffiliationRequests` protocol incorporates the `Joining` protocol:

```
@protocol AffiliationRequests <Joining>
```

If a class adopts a protocol that incorporates another protocol, it must also implement all the methods in the incorporated protocol or inherit those methods from a class that adopts it.

This method determines conformance solely on the basis of the formal declarations in header files, as illustrated above. It doesn't check to see whether the methods declared in the protocol are actually implemented—that's the programmer's responsibility.

The protocol required as this method's argument can be specified using the `@protocol()` directive:

```
BOOL canJoin = [MyClass conformsToProtocol:@protocol(Joining)];
```

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`+ conformsToProtocol:` (page 18)

**Declared In**
`NSObject.h`

## copyWithZone:

Returns the receiver.

```
+ (id)copyWithZone:(NSZone *)zone
```

**Return Value**
The receiver.

**Discussion**
This method exists so class objects can be used in situations where you need an object that conforms to the `NSCopying` protocol. For example, this method lets you use a class object as a key to an `NSDictionary` object. You should not override this method.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`- copy` (page 34)

**Related Sample Code**
AlbumToSlideshow
Quartz Composer WWDC 2005 TextEdit
TextEditPlus

**Declared In**
`NSObject.h`

## description

Returns a string that represents the contents of the receiving class.

```
+ (NSString *)description
```

**Return Value**
A string that represents the contents of the receiving class.

**Discussion**
The debugger's print-object command invokes this method to produce a textual description of an object.

`NSObject`'s implementation of this method simply prints the name of the class.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`description` (`NSObject` protocol)

**Related Sample Code**
iSpend
QTKitMovieShuffler
QTRecorder
SimpleCalendar
StickiesExample

**Declared In**
`NSObject.h`

## initialize

Initializes the receiver before it's used (before it receives its first message).

```
+ (void)initialize
```

**Discussion**
The runtime sends `initialize` to each class in a program exactly one time just before the class, or any class that inherits from it, is sent its first message from within the program. (Thus the method may never be invoked if the class is not used.) The runtime sends the `initialize` message to classes in a thread-safe manner. Superclasses receive this message before their subclasses.

For example, if the first message your program sends is this:

```
[NSApplication new]
```

the runtime system sends these three `initialize` messages:

```
[NSObject initialize];
[NSResponder initialize];
[NSApplication initialize];
```

because `NSApplication` is a subclass of `NSResponder` and `NSResponder` is a subclass of `NSObject`. All the `initialize` messages precede the `new` (page 25) message.

If your program later begins to use the `NSText` class,

```
[NSText instancesRespondToSelector:someSelector]
```

the runtime system invokes these additional `initialize` messages:

```
[NSView initialize];
[NSText initialize];
```

because `NSText` inherits from `NSObject`, `NSResponder`, and `NSView`. The `instancesRespondToSelector:` (page 23) message is sent only after all these classes are initialized. Note that the `initialize` messages to `NSObject` and `NSResponder` aren't repeated.

You implement `initialize` to provide class-specific initialization as needed. Since the runtime sends appropriate initialize messages automatically, you should typically not send `initialize` to `super` in your implementation.

If a particular class does not implement `initialize`, the `initialize` method of its superclass is invoked twice, once for the superclass and once for the non-implementing subclass. If you want to make sure that your class performs class-specific initializations only once, implement `initialize` as in the following example:

```
@implementation MyClass
+ (void)initialize
{
    if ( self == [MyClass class] ) {
        /* put initialization code here */
    }
}
```

Loading a subclasses of MyClass that does not implement its own `initialize` method will cause MyClass's implementation to be invoked. The test clause (`if ( self == [MyClass class] )`) ensures that the initialization code has no effect if `initialize` is invoked when a subclass is loaded.

**Special Considerations**

`initialize` it is invoked only once per class. If you want to perform independent initialization for the class and for categories of the class, you should implement `load` (page 23) methods.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`- init` (page 40)
`+ load` (page 23)
`class` (`NSObject` protocol)

**Related Sample Code**
CoreRecipes
Dicey
iSpend
NewsReader
Reducer

**Declared In**
`NSObject.h`

## instanceMethodForSelector:

Locates and returns the address of the implementation of the instance method identified by a given selector.

```
+ (IMP)instanceMethodForSelector:(SEL)aSelector
```

**Parameters**

*aSelector*

A selector that identifies the method for which to return the implementation address. The selector must be non-NULL and valid for the receiver. If in doubt, use the respondsToSelector: method to check before passing the selector to methodForSelector:.

See "Selectors" (page 7) for a description of the SEL type.

**Return Value**

The address of the implementation of the *aSelector* instance method.

**Discussion**

An error is generated if instances of the receiver can't respond to *aSelector* messages.

Use this method to ask the class object for the implementation of instance methods only. To ask the class for the implementation of a class method, send the methodForSelector: (page 42) instance method to the class instead.

See "Selectors" (page 7) for a description of the IMP type, and how to invoke the returned method implementation.

**Availability**

Available in Mac OS X v10.0 and later.

**Declared In**

NSObject.h

## instanceMethodSignatureForSelector:

Returns an NSMethodSignature object that contains a description of the instance method identified by a given selector.

```
+ (NSMethodSignature *)instanceMethodSignatureForSelector:(SEL)aSelector
```

**Parameters**

*aSelector*

A selector that identifies the method for which to return the implementation address.

See "Selectors" (page 7) for a description of the SEL type.

**Return Value**

An NSMethodSignature object that contains a description of the instance method identified by *aSelector*, or nil if the method can't be found.

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**

- methodSignatureForSelector: (page 43)

**Declared In**

NSObject.h

## instancesRespondToSelector:

Returns a Boolean value that indicates whether instances of the receiver are capable of responding to a given selector.

```
+ (BOOL)instancesRespondToSelector:(SEL)aSelector
```

**Parameters**

*aSelector*

A selector. See "Selectors" (page 7) for a description of the `SEL` type.

**Return Value**

`YES` if instances of the receiver are capable of responding to *aSelector* messages, otherwise `NO`.

**Discussion**

If *aSelector* messages are forwarded to other objects, instances of the class are able to receive those messages without error even though this method returns `NO`.

To ask the class whether it, rather than its instances, can respond to a particular message, send to the class instead the `NSObject` protocol instance method `respondsToSelector:`.

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**

`- forwardInvocation:` (page 38)

**Declared In**

`NSObject.h`

## isSubclassOfClass:

Returns a Boolean value that indicates whether the receiving class is a subclass of, or identical to, a given class.

```
+ (BOOL)isSubclassOfClass:(Class)aClass
```

**Parameters**

*aClass*

A class object.

**Return Value**

`YES` if the receiving class is a subclass of—or identical to—*aClass*, otherwise `NO`.

**Availability**

Available in Mac OS X v10.2 and later.

**Declared In**

`NSObject.h`

## load

Invoked whenever a class or category is added to the Objective-C runtime; implement this method to perform class-specific behavior upon loading.

```
+ (void)load
```

**Discussion**
The `load` message is sent to classes and categories that are both dynamically loaded and statically linked, but only if the newly loaded class or category implements a method that can respond.

On Mac OS X v10.5, the order of initialization is as follows:

1. All initializers in any framework you link to.
2. All `+load` methods in your image.
3. All C++ static initializers and C/C++ `__attribute__(constructor)` functions in your image.
4. All initializers in frameworks that link to you.

In addition:

- A class's `+load` method is called after all of its superclasses' `+load` methods.
- A category `+load` method is called after the class's own `+load` method.

In a `+load` method, you can therefore safely message other unrelated classes from the same image, but any `+load` methods on those classes may not have run yet.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`+ initialize` (page 20)

**Related Sample Code**
CIAnnotation
CustomAtomicStoreSubclass
LSMSmartCategorizer
MethodReplacement
TextLinks

**Declared In**
`NSObject.h`

## mutableCopyWithZone:

Returns the receiver.

```
+ (id)mutableCopyWithZone:(NSZone *)zone
```

**Parameters**

*zone*
The memory zone in which to create the copy of the receiver.

**Return Value**
The receiver.

**Discussion**
This method exists so class objects can be used in situations where you need an object that conforms to the `NSMutableCopying` protocol. For example, this method lets you use a class object as a key to an `NSDictionary` object. You should not override this method.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`NSObject.h`

## new

Allocates a new instance of the receiving class, sends it an `init` (page 40) message, and returns the initialized object.

```
+ (id)new
```

**Return Value**
A new instance of the receiver.

**Discussion**
This method is a combination of `alloc` (page 14) and `init` (page 40). Like `alloc` (page 14), it initializes the `isa` instance variable of the new object so it points to the class data structure. It then invokes the `init` (page 40) method to complete the initialization process.

Unlike `alloc` (page 14), `new` (page 25) is sometimes re-implemented in subclasses to invoke a class-specific initialization method. If the `init...` method includes arguments, they're typically reflected in a `new...` method as well. For example:

```
+ newMyClassWithTag:(int)tag data:(struct info *)data
{
    return [[self alloc] initWithTag:tag data:data];
}
```

However, there's little point in implementing a `new...` method if it's simply a shorthand for `alloc` (page 14) and `init...`, as shown above. Often `new...` methods will do more than just allocation and initialization. In some classes, they manage a set of instances, returning the one with the requested properties if it already exists, allocating and initializing a new instance only if necessary. For example:

```
+ newMyClassWithTag:(int)tag data:(struct info *)data
{
    MyClass *theInstance;

    if ( theInstance = findTheObjectWithTheTag(tag) )
        return [theInstance retain];
    return [[self alloc] initWithTag:tag data:data];
}
```

Although it's appropriate to define new `new...` methods in this way, the `alloc` (page 14) and `allocWithZone:` (page 14) methods should never be augmented to include initialization code.

**Special Considerations**

If you are using managed memory (not garbage collection), this method retains the object before returning it. The returned object has a retain count of `1` and is *not* autoreleased. The invoker of this method is responsible for releasing the returned object, using either `release` or `autorelease`.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
Fiendishthngs
LSMSmartCategorizer
NURBSSurfaceVertexProg
Quartz Composer QCTV
SurfaceVertexProgram

**Declared In**
`NSObject.h`

## resolveClassMethod:

Dynamically provides an implementation for a given selector for a class method.

```
+ (BOOL)resolveClassMethod:(SEL)name
```

**Parameters**

*name*

The name of a selector to resolve.

**Return Value**
`YES` if the method was found and added to the receiver, otherwise `NO`.

**Discussion**
This method allows you to dynamically provides an implementation for a given selector. See `resolveInstanceMethod:` (page 26) for further discussion.

**Availability**
Available in Mac OS X v10.5 and later.

**See Also**
`+ resolveInstanceMethod:` (page 26)

**Declared In**
`NSObject.h`

## resolveInstanceMethod:

Dynamically provides an implementation for a given selector for an instance method.

```
+ (BOOL)resolveInstanceMethod:(SEL)name
```

**Parameters**

*name*

The name of a selector to resolve.

**Return Value**

`YES` if the method was found and added to the receiver, otherwise `NO`.

**Discussion**

This method and `resolveClassMethod:` (page 26) allow you to dynamically provide an implementation for a given selector.

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

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

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

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

**Special Considerations**

This method is called before the Objective-C forwarding mechanism (see The Runtime System in *The Objective-C 2.0 Programming Language*) is invoked. If `respondsToSelector:` or `instancesRespondToSelector:` (page 23) is invoked, the dynamic method resolver is given the opportunity to provide an `IMP` for the given selector first.

**Availability**

Available in Mac OS X v10.5 and later.

**See Also**

`+ resolveClassMethod:` (page 26)

**Declared In**

`NSObject.h`

## setVersion:

Sets the receiver's version number.

```
+ (void)setVersion:(NSInteger)aVersion
```

**Parameters**

*aVersion*

The version number for the receiver.

**Discussion**

The version number is helpful when instances of the class are to be archived and reused later. The default version is `0`.

**Special Considerations**

The version number applies to `NSArchiver`/`NSUnarchiver`, but not to `NSKeyedArchiver`/`NSKeyedUnarchiver`. A keyed archiver does not encode class version numbers.

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**

`+ version` (page 28)

**Declared In**

`NSObject.h`

## superclass

Returns the class object for the receiver's superclass.

```
+ (Class)superclass
```

**Return Value**

The class object for the receiver's superclass.

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**

`+ class` (page 17)

`superclass` (`NSObject` protocol)

**Declared In**

`NSObject.h`

## version

Returns the version number assigned to the class.

```
+ (NSInteger)version
```

**Return Value**

The version number assigned to the class.

**Discussion**

If no version has been set, the default is `0`.

Version numbers are needed for decoding or unarchiving, so older versions of an object can be detected and decoded correctly.

Caution should be taken when obtaining the version from within an `NSCoding` protocol or other methods. Use the class name explicitly when getting a class version number:

```
version = [MyClass version];
```

Don't simply send `version` to the return value of class—a subclass version number may be returned instead.

**Special Considerations**

The version number applies to `NSArchiver`/`NSUnarchiver`, but not to `NSKeyedArchiver`/`NSKeyedUnarchiver`. A keyed archiver does not encode class version numbers.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`+ setVersion:` (page 27)
`versionForClassName:` (`NSCoder`)

**Related Sample Code**
CoreRecipes
Fiendishthngs
PrefsPane

**Declared In**
`NSObject.h`

# Instance Methods

## attributeKeys

Returns an array of `NSString` objects containing the names of immutable values that instances of the receiver's class contain.

```
- (NSArray *)attributeKeys
```

**Return Value**
An array of `NSString` objects containing the names of immutable values that instances of the receiver's class contain.

**Discussion**
`NSObject`'s implementation of `attributeKeys` simply calls `[[self classDescription] attributeKeys]`. To make use of the default implementation, you must therefore implement and register a suitable class description—see `NSClassDescription`. A class description that describes Movie objects could, for example, return the attribute keys `title`, `dateReleased`, and `rating`.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
- `classDescription` (page 31)
- `inverseForRelationshipKey:` (page 42)
- `toManyRelationshipKeys` (page 56)
- `toOneRelationshipKeys` (page 56)

**Related Sample Code**
Core Data HTML Store
CoreRecipes
StickiesExample

**Declared In**
`NSClassDescription.h`

## awakeAfterUsingCoder:

Overridden by subclasses to substitute another object in place of the object that was decoded and subsequently received this message.

```
- (id)awakeAfterUsingCoder:(NSCoder *)aDecoder
```

**Parameters**
*aDecoder*
The decoder used to decode the receiver.

**Return Value**
The receiver, or another object to take the place of the object that was decoded and subsequently received this message.

**Discussion**
This method can be used to eliminate redundant objects created by the coder. For example, if after decoding an object you discover that an equivalent object already exists, you can return the existing object. If a replacement is returned, your overriding method is responsible for releasing the receiver. To prevent the accidental use of the receiver after its replacement has been returned, you should invoke the receiver's `release` method to release the object immediately.

This method is invoked by `NSCoder`. `NSObject`'s implementation simply returns `self`.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
- `classForCoder` (page 32)
- `replacementObjectForCoder:` (page 53)
`initWithCoder:` (`NSCoding` protocol)

**Declared In**
`NSObject.h`

## classCode

Returns the receiver's Apple event type code, as stored in the `NSScriptClassDescription` object for the object's class.

```
- (FourCharCode)classCode
```

**Return Value**
The receiver's Apple event type code, as stored in the `NSScriptClassDescription` object for the object's class.

**Discussion**
This method is invoked by Cocoa's scripting support classes.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
Video Hardware Info

**Declared In**
`NSScriptClassDescription.h`

## classDescription

Returns an object containing information about the attributes and relationships of the receiver's class.

```
- (NSClassDescription *)classDescription
```

**Return Value**
An object containing information about the attributes and relationships of the receiver's class.

**Discussion**
`NSObject`'s implementation simply calls `[NSClassDescription classDescriptionForClass:[self class]]`. See `NSClassDescription` for more information.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
- `attributeKeys` (page 29)
- `inverseForRelationshipKey:` (page 42)
- `toManyRelationshipKeys` (page 56)
- `toOneRelationshipKeys` (page 56)

**Related Sample Code**
SimpleScriptingObjects

**Declared In**
`NSClassDescription.h`

## classForArchiver

Overridden by subclasses to substitute a class other than its own during archiving.

```
- (Class)classForArchiver
```

**Return Value**
The class to substitute for the receiver's own class during archiving.

**Discussion**
This method is invoked by `NSArchiver`. It allows specialized behavior for archiving—for example, the private subclasses of a class cluster substitute the name of their public superclass when being archived.

`NSObject`'s implementation returns the object returned by `classForCoder` (page 32). Override `classForCoder` (page 32) to add general coding behavior.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`- replacementObjectForArchiver:` (page 52)

**Declared In**
`NSArchiver.h`

## classForCoder

Overridden by subclasses to substitute a class other than its own during coding.

```
- (Class)classForCoder
```

**Return Value**
The class to substitute for the receiver's own class during coding.

**Discussion**
This method is invoked by `NSCoder`. `NSObject`'s implementation returns the receiver's class. The private subclasses of a class cluster substitute the name of their public superclass when being archived.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`- awakeAfterUsingCoder:` (page 30)
`- replacementObjectForCoder:` (page 53)

**Declared In**
`NSObject.h`

## classForKeyedArchiver

Overridden by subclasses to substitute a new class for instances during keyed archiving.

```
- (Class)classForKeyedArchiver
```

**Discussion**
The object will be encoded as if it were a member of the returned class. The results of this method are overridden by the encoder class and instance name to class encoding tables. If `nil` is returned, the result of this method is ignored.

**Availability**
Available in Mac OS X v10.2 and later.

**See Also**
`- replacementObjectForKeyedArchiver:` (page 53)

**Declared In**
`NSKeyedArchiver.h`

## classForPortCoder

Overridden by subclasses to substitute a class other than its own for distribution encoding.

```
- (Class)classForPortCoder
```

**Return Value**
The class to substitute for the receiver in distribution encoding.

**Discussion**
This method allows specialized behavior for distributed objects—override `classForCoder` (page 32) to add general coding behavior. This method is invoked by `NSPortCoder`. `NSObject`'s implementation returns the class returned by `classForCoder` (page 32).

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`- replacementObjectForPortCoder:` (page 54)

**Declared In**
`NSPortCoder.h`

## className

Returns a string containing the name of the class.

```
- (NSString *)className
```

**Return Value**
A string containing the name of the class.

**Discussion**
This method is invoked by Cocoa's scripting support classes.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
Sketch-112

StickiesExample

**Declared In**
NSScriptClassDescription.h

## copy

Returns the object returned by `copyWithZone:`, where the zone is `nil`.

```
- (id)copy
```

**Return Value**
The object returned by the `NSCopying` protocol method `copyWithZone:`, where the zone is `nil`.

**Discussion**
This is a convenience method for classes that adopt the `NSCopying` protocol. An exception is raised if there is no implementation for `copyWithZone:`.

`NSObject` does not itself support the `NSCopying` protocol. Subclasses must support the protocol and implement the `copyWithZone:` method. A subclass version of the `copyWithZone:` method should send the message to `super` first, to incorporate its implementation, unless the subclass descends directly from `NSObject`.

**Special Considerations**

If you are using managed memory (not garbage collection), this method retains the new object before returning it. The invoker of the method, however, is responsible for releasing the returned object.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
AttachAScript
Dicey
iSpend
Quartz Composer WWDC 2005 TextEdit
TextEditPlus

**Declared In**
NSObject.h

## copyScriptingValue:forKey:withProperties:

Creates and returns one or more scripting objects to be inserted into the specified relationship by copying the passed-in value and setting the properties in the copied object or objects.

```
- (id)copyScriptingValue:(id)value forKey:(NSString *)key
    withProperties:(NSDictionary *)properties;
```

**Parameters**

*value*

An object or objects to be copied. The type must match the type of the property identified by *key*. (See also the Discussion section.)

For example, if the property is a to-many relationship, `value` will always be an array of objects to be copied, and this method must therefore return an array of objects.

*key*

A key that identifies the relationship into which to insert the copied object or objects.

*properties*

The properties to be set in the copied object or objects. Derived from the "with properties" parameter of a `duplicate` command. (See also the Discussion section.)

**Return Value**
The copied object or objects. Returns `nil` if an error occurs.

**Discussion**
You can override the `copyScriptingValue` method to take more control when your application is sent a `duplicate` command. This method is invoked on the prospective container of the copied object or objects. The `properties` are derived from the `with properties` parameter of the `duplicate` command. The returned objects or objects are then inserted into the container using key-value coding.

When this method is invoked by Cocoa, neither the value nor the properties will have yet been coerced using the `NSScriptKeyValueCoding` method `coerceValue:forKey:`. For sdef-declared scriptability, however, the types of the passed-in objects reliably match the relevant sdef declarations.

The default implementation of this method copies scripting objects by sending `copyWithZone:` to the object or objects specified by *value*. You override this method for situations where this is not sufficient, such as in Core Data applications, in which new objects must be initialized with `[NSManagedObject initWithEntity:insertIntoManagedObjectContext:]`.

**Availability**
Available in Mac OS X v10.5 and later.

**Declared In**
`NSObjectScripting.h`

## dealloc

Deallocates the memory occupied by the receiver.

```
- (void)dealloc
```

**Discussion**
Subsequent messages to the receiver may generate an error indicating that a message was sent to a deallocated object (provided the deallocated memory hasn't been reused yet).

You never send a `dealloc` message directly. Instead, an object's `dealloc` method is invoked indirectly through the `release` `NSObject` protocol method (if the `release` message results in the receiver's retain count becoming `0`). See *Memory Management Programming Guide for Cocoa* for more details on the use of these methods.

Subclasses must implement their own versions of `dealloc` to allow the release of any additional memory consumed by the object—such as dynamically allocated storage for data or object instance variables owned by the deallocated object. After performing the class-specific deallocation, the subclass method should incorporate superclass versions of `dealloc` through a message to `super`:

```
- (void)dealloc {
    [companion release];
    NSZoneFree(private, [self zone])
    [super dealloc];
}
```

**Important:** Note that when an application terminates, objects may not be sent a `dealloc` message since the process's memory is automatically cleared on exit—it is more efficient simply to allow the operating system to clean up resources than to invoke all the memory management methods. For this and other reasons, you should not manage scarce resources in `dealloc`—see Object Ownership and Disposal in *Memory Management Programming Guide for Cocoa* for more details.

#### Special Considerations

When garbage collection is enabled, the garbage collector sends `finalize` (page 37) to the receiver instead of `dealloc`.

When garbage collection is enabled, this method is a no-op.

#### Availability

Available in Mac OS X v10.0 and later.

#### See Also

`autorelease` (`NSObject` protocol)

`release` (`NSObject` protocol)

`- finalize` (page 37)

#### Related Sample Code

ImageClient

iSpend

QTCoreVideo301

Sketch-112

StickiesExample

#### Declared In

`NSObject.h`

## doesNotRecognizeSelector:

Handles messages the receiver doesn't recognize.

```
- (void)doesNotRecognizeSelector:(SEL)aSelector
```

#### Parameters

*aSelector*

A selector that identifies a method not implemented or recognized by the receiver.

See "Selectors" (page 7) for a description of the `SEL` type.

**Discussion**
The runtime system invokes this method whenever an object receives an *aSelector* message it can't respond to or forward. This method, in turn, raises an `NSInvalidArgumentException`, and generates an error message.

Any `doesNotRecognizeSelector:` messages are generally sent only by the runtime system. However, they can be used in program code to prevent a method from being inherited. For example, an `NSObject` subclass might renounce the `copy` (page 34) or `init` (page 40) method by re-implementing it to include a `doesNotRecognizeSelector:` message as follows:

```
- (id)copy
{
    [self doesNotRecognizeSelector:_cmd];
}
```

The `_cmd` variable is a hidden argument passed to every method that is the current selector; in this example, it identifies the selector for the `copy` method. This code prevents instances of the subclass from responding to `copy` messages or superclasses from forwarding `copy` messages—although `respondsToSelector:` will still report that the receiver has access to a `copy` method.

If you override this method, you must call `super` or raise an `NSInvalidArgumentException` exception at the end of your implementation. In other words, this method must not return normally; it must always result in an exception being thrown.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
`- forwardInvocation:` (page 38)

**Declared In**
`NSObject.h`

## finalize

The garbage collector invokes this method on the receiver before disposing of the memory it uses.

```
- (void)finalize
```

**Discussion**
The garbage collector invokes this method on the receiver before disposing of the memory it uses. When garbage collection is enabled, this method is invoked instead of `dealloc`.

**Note:** Garbage collection is not available for use in Mac OS X before version 10.5.

You can override this method to relinquish resources the receiver has obtained, as shown in the following example:

```
- (void)finalize {
    if (log_file != NULL) {
        fclose(log_file);
        log_file = NULL;
    }
```

```
    [super finalize];
}
```

Typically, however, you are encouraged to relinquish resources prior to finalization if at all possible. For more details, see Implementing a finalize Method.

**Special Considerations**

It is an error to store `self` into a new or existing live object (colloquially known as "resurrection"), which implies that this method will be called only once. However, the receiver may be messaged after finalization by other objects also being finalized at this time, so your override should guard against future use of resources that have been reclaimed, as shown by the `log_file = NULL` statement in the example. The `finalize` method itself will never be invoked more than once for a given object.

**Important:** `finalize` methods must be thread-safe.

**Availability**
Available in Mac OS X v10.4 and later.

**See Also**
`- dealloc` (page 35)

**Declared In**
`NSObject.h`

## forwardInvocation:

Overridden by subclasses to forward messages to other objects.

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
```

**Parameters**

*anInvocation*

The invocation to forward.

**Discussion**
When an object is sent a message for which it has no corresponding method, the runtime system gives the receiver an opportunity to delegate the message to another receiver. It delegates the message by creating an `NSInvocation` object representing the message and sending the receiver a `forwardInvocation:` message containing this `NSInvocation` object as the argument. The receiver's `forwardInvocation:` method can then choose to forward the message to another object. (If that object can't respond to the message either, it too will be given a chance to forward it.)

The `forwardInvocation:` message thus allows an object to establish relationships with other objects that will, for certain messages, act on its behalf. The forwarding object is, in a sense, able to "inherit" some of the characteristics of the object it forwards the message to.

**Important:** To respond to methods that your object does not itself recognize, you must override `methodSignatureForSelector:` (page 43) in addition to `forwardInvocation:`. The mechanism for forwarding messages uses information obtained from `methodSignatureForSelector:` (page 43) to create the `NSInvocation` object to be forwarded. Your overriding method must provide an appropriate method signature for the given selector, either by preformulating one or by asking another object for one.

An implementation of the `forwardInvocation:` method has two tasks:

- To locate an object that can respond to the message encoded in *`anInvocation`*. This object need not be the same for all messages.
- To send the message to that object using *`anInvocation`*. *`anInvocation`* will hold the result, and the runtime system will extract and deliver this result to the original sender.

In the simple case, in which an object forwards messages to just one destination (such as the hypothetical `friend` instance variable in the example below), a `forwardInvocation:` method could be as simple as this:

```
- (void)forwardInvocation:(NSInvocation *)invocation
{
    SEL aSelector = [invocation selector];

    if ([friend respondsToSelector:aSelector])
        [invocation invokeWithTarget:friend];
    else
        [self doesNotRecognizeSelector:aSelector];
}
```

The message that's forwarded must have a fixed number of arguments; variable numbers of arguments (in the style of `printf()`) are not supported.

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

Implementations of the `forwardInvocation:` method can do more than just forward messages. `forwardInvocation:` can, for example, be used to consolidate code that responds to a variety of different messages, thus avoiding the necessity of having to write a separate method for each selector. A `forwardInvocation:` method might also involve several other objects in the response to a given message, rather than forward it to just one.

`NSObject`'s implementation of `forwardInvocation:` simply invokes the `doesNotRecognizeSelector:` (page 36) method; it doesn't forward any messages. Thus, if you choose not to implement `forwardInvocation:`, sending unrecognized messages to objects will raise exceptions.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`NSObject.h`

## init

Implemented by subclasses to initialize a new object (the receiver) immediately after memory for it has been allocated.

```
- (id)init
```

**Return Value**
The initialized receiver.

**Discussion**
An `init` message is generally coupled with an `alloc` (page 14) or `allocWithZone:` (page 14) message in the same line of code:

```
TheClass *newObject = [[TheClass alloc] init];
```

An object isn't ready to be used until it has been initialized. The `init` method defined in the `NSObject` class does no initialization; it simply returns `self`.

Subclass implementations of this method should initialize and return the new object. If it can't be initialized, they should release the object and return `nil`. In some cases, an `init` method might release the new object and return a substitute. Programs should therefore always use the object returned by `init`, and not necessarily the one returned by `alloc` (page 14) or `allocWithZone:` (page 14), in subsequent code.

Every class must guarantee that the `init` method either returns a fully functional instance of the class or raises an exception. Subclasses should override the `init` method to add class-specific initialization code. Subclass versions of `init` need to incorporate the initialization code for the classes they inherit from, through a message to `super`:

```
- (id)init
{
    if ((self = [super init])) {
        /* class-specific initialization goes here */
    }
    return self;
}
```

Note that the message to `super` precedes the initialization code added in the method. This sequencing ensures that initialization proceeds in the order of inheritance.

Subclasses often define `init...` methods with additional arguments to allow specific values to be set. The more arguments a method has, the more freedom it gives you to determine the character of initialized objects. Classes often have a set of `init...` methods, each with a different number of arguments. For example:

```
- (id)init;
- (id)initWithTag:(int)tag;
- (id)initWithTag:(int)tag data:(struct info *)data;
```

The convention is that at least one of these methods, usually the one with the most arguments, includes a message to `super` to incorporate the initialization of classes higher up the hierarchy. This method is called the *designated initializer* for the class. The other `init...` methods defined in the class directly or indirectly invoke the designated initializer through messages to `self`. In this way, all `init...` methods are chained together. For example:

```
- (id)init
{
```

```
    return [self initWithTag:-1];
}

- (id)initWithTag:(int)tag
{
    return [self initWithTag:tag data:NULL];
}

- (id)initWithTag:(int)tag data:(struct info *)data
{
    if ((self = [super init. . .])) {
        /* class-specific initialization goes here */
    }
    return self;
}
```

In this example, the `initWithTag:data:` method is the designated initializer for the class.

If a subclass does any initialization of its own, it must define its own designated initializer. This method should begin by sending a message to `super` to invoke the designated initializer of its superclass. Suppose, for example, that the three methods illustrated above are defined in the B class. The C class, a subclass of B, might have this designated initializer:

```
- (id)initWithTag:(int)tag data:(struct info *)data object:anObject
{
    if ((self = [super initWithTag:tag data:data])) {
        /* class-specific initialization goes here */
    }
    return self;
}
```

If inherited `init...` methods are to successfully initialize instances of the subclass, they must all be made to (directly or indirectly) invoke the new designated initializer. To accomplish this, the subclass is obliged to cover (override) only the designated initializer of the superclass. For example, in addition to its designated initializer, the C class would also implement this method:

```
- (id)initWithTag:(int)tag data:(struct info *)data
{
    return [self initWithTag:tag data:data object:nil];
}
```

This code ensures that all three methods inherited from the B class also work for instances of the C class.

Often the designated initializer of the subclass overrides the designated initializer of the superclass. If so, the subclass need only implement the one `init...` method.

These conventions maintain a direct chain of `init...` links and ensure that the `new` method and all inherited `init...` methods return usable, initialized objects. They also prevent the possibility of an infinite loop wherein a subclass method sends a message (to `super`) to perform a superclass method, which in turn sends a message (to `self`) to perform the subclass method.

This `init` method is the designated initializer for the `NSObject` class. Subclasses that do their own initialization should override it, as described above.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
ImageClient
Quartz Composer WWDC 2005 TextEdit
Quartz EB
StickiesExample
TextEditPlus

**Declared In**
`NSObject.h`

## inverseForRelationshipKey:

For a given key that defines the name of the relationship from the receiver's class to another class, returns the name of the relationship from the other class to the receiver's class.

```
- (NSString *)inverseForRelationshipKey:(NSString *)relationshipKey
```

**Parameters**
*`relationshipKey`*
The name of the relationship from the receiver's class to another class.

**Return Value**
The name of the relationship that is the inverse of the receiver's relationship named *`relationshipKey`*.

**Discussion**
`NSObject`'s implementation of `inverseForRelationshipKey:` simply invokes `[[self classDescription] inverseForRelationshipKey:relationshipKey]`. To make use of the default implementation, you must therefore implement and register a suitable class description—see `NSClassDescription`.

For example, suppose an Employee class has a relationship named `department` to a Department class, and that Department has a relationship called `employees` to Employee. The statement:

```
employee inverseForRelationshipKey:@"department"];
```

returns the string `employees`.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
- `attributeKeys` (page 29)
- `classDescription` (page 31)
- `toManyRelationshipKeys` (page 56)
- `toOneRelationshipKeys` (page 56)

**Declared In**
`NSClassDescription.h`

## methodForSelector:

Locates and returns the address of the receiver's implementation of a method so it can be called as a function.

```
- (IMP)methodForSelector:(SEL)aSelector
```

**Parameters**

*aSelector*

A selector that identifies the method for which to return the implementation address. The selector must be a valid and non-NULL. If in doubt, use the respondsToSelector: method to check before passing the selector to methodForSelector:.

**Return Value**

The address of the receiver's implementation of the *aSelector*.

**Discussion**

If the receiver is an instance, *aSelector* should refer to an instance method; if the receiver is a class, it should refer to a class method.

See "Selectors" (page 7) for a description of the IMP and SEL types, and how to invoke the returned method implementation.

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**

+ instanceMethodForSelector: (page 21)

**Declared In**

NSObject.h

## methodSignatureForSelector:

Returns an NSMethodSignature object that contains a description of the method identified by a given selector.

```
- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector
```

**Parameters**

*aSelector*

A selector that identifies the method for which to return the implementation address. When the receiver is an instance, *aSelector* should identify an instance method; when the receiver is a class, it should identify a class method.

See "Selectors" (page 7) for a description of the SEL type.

**Return Value**

An NSMethodSignature object that contains a description of the method identified by *aSelector*, or nil if the method can't be found.

**Discussion**

This method is used in the implementation of protocols. This method is also used in situations where an NSInvocation object must be created, such as during message forwarding. If your object maintains a delegate or is capable of handling messages that it does not directly implement, you should override this method to return an appropriate method signature.

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**
+ instanceMethodSignatureForSelector: (page 22)
- forwardInvocation: (page 38)

**Declared In**
NSObject.h

## mutableCopy

Returns the object returned by mutableCopyWithZone: where the zone is nil.

```
- (id)mutableCopy
```

**Return Value**
The object returned by the NSMutableCopying protocol method mutableCopyWithZone:, where the zone is nil.

**Discussion**
This is a convenience method for classes that adopt the NSMutableCopying protocol. An exception is raised if there is no implementation for mutableCopyWithZone:.

**Special Considerations**

If you are using managed memory (not garbage collection), this method retains the new object before returning it. The invoker of the method, however, is responsible for releasing the returned object.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
CoreRecipes
EnhancedAudioBurn
iSpend
Quartz Composer WWDC 2005 TextEdit
TextEditPlus

**Declared In**
NSObject.h

## newScriptingObjectOfClass:forValueForKey:withContentsValue:properties:

Creates and returns an instance of a scriptable class, setting its contents and properties, for insertion into the relationship identified by the key.

```
- (id)newScriptingObjectOfClass:(Class)class forValueForKey:(NSString *)key
    withContentsValue:(id)contentsValue properties:(NSDictionary *)properties;
```

**Parameters**

*class*
The class of the scriptable object to be created.

*key*
A key that identifies the relationship into which the new class object will be inserted.

*contentsValue*

Specifies the contents of the object to be created. This may be `nil`. (See also the Discussion section.)

*properties*

The properties to be set in the new object. (See also the Discussion section.)

**Return Value**

The new object. Returns `nil` if an error occurs.

**Discussion**

You can override the `newScriptingObjectOfClass` method to take more control when your application is sent a `make` command. This method is invoked on the prospective container of the new object. The `contentsValue` and `properties` are derived from the `with contents` and `with properties` parameters of the `make` command. The returned objects or objects are then inserted into the container using key-value coding.

When this method is invoked by Cocoa, neither the contents value nor the properties will have yet been coerced using the `NSScriptKeyValueCoding` method `coerceValue:forKey:`. For sdef-declared scriptability, however, the types of the passed-in objects reliably match the relevant sdef declarations.

The default implementation of this method creates new scripting objects by sending `alloc` to a class and `init` to the resulting object. You override this method for situations where this is not sufficient, such as in Core Data applications, in which new objects must be initialized with `[NSManagedObject initWithEntity:insertIntoManagedObjectContext:]`.

**Availability**

Available in Mac OS X v10.5 and later.

**Declared In**

`NSObjectScripting.h`

## performSelector:onThread:withObject:waitUntilDone:

Invokes a method of the receiver on the specified thread using the default mode.

```
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(id)arg
    waitUntilDone:(BOOL)wait
```

**Parameters**

*aSelector*

A selector that identifies the method to invoke. The method should not have a significant return value and should take a single argument of type `id`, or no arguments.

See "Selectors" (page 7) for a description of the `SEL` type.

*thr*

The thread on which to execute *aSelector*.

*arg*

The argument to pass to the method when it is invoked. Pass `nil` if the method does not take an argument.

*wait*

A Boolean that specifies whether the current thread blocks until after the specified selector is performed on the receiver on the specified thread. Specify `YES` to block this thread; otherwise, specify `NO` to have this method return immediately.

If the current thread and target thread are the same, and you specify `YES` for this parameter, the selector is performed immediately on the current thread. If you specify `NO`, this method queues the message on the thread's run loop and returns, just like it does for other threads. The current thread must then dequeue and process the message when it has an opportunity to do so.

**Discussion**

You can use this method to deliver messages to other threads in your application. The message in this case is a method of the current object that you want to execute on the target thread.

This method queues the message on the run loop of the target thread using the default run loop modes—that is, the modes associated with the `NSRunLoopCommonModes` constant. As part of its normal run loop processing, the target thread dequeues the message (assuming it is running in one of the default run loop modes) and invokes the desired method.

You cannot cancel messages queued using this method. If you want the option of canceling a message on the current thread, you must use either the `performSelector:withObject:afterDelay:` (page 47) or `performSelector:withObject:afterDelay:inModes:` (page 48) method.

This method retains the receiver and the *arg* parameter until after the selector is performed.

**Availability**

Available in Mac OS X v10.5 and later.

**See Also**

- `performSelector:onThread:withObject:waitUntilDone:modes:` (page 46)
- `performSelectorInBackground:withObject:` (page 49)

**Declared In**

`NSThread.h`

## performSelector:onThread:withObject:waitUntilDone:modes:

Invokes a method of the receiver on the specified thread using the specified modes.

```
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(id)arg
    waitUntilDone:(BOOL)wait modes:(NSArray *)array
```

**Parameters**

*aSelector*

A selector that identifies the method to invoke. It should not have a significant return value and should take a single argument of type `id`, or no arguments.

See "Selectors" (page 7) for a description of the `SEL` type.

*thr*

The thread on which to execute *aSelector*. This thread represents the target thread.

*arg*

The argument to pass to the method when it is invoked. Pass `nil` if the method does not take an argument.

*wait*

A Boolean that specifies whether the current thread blocks until after the specified selector is performed on the receiver on the specified thread. Specify `YES` to block this thread; otherwise, specify `NO` to have this method return immediately.

If the current thread and target thread are the same, and you specify `YES` for this parameter, the selector is performed immediately. If you specify `NO`, this method queues the message and returns immediately, regardless of whether the threads are the same or different.

*array*

An array of strings that identifies the modes in which it is permissible to perform the specified selector. This array must contain at least one string. If you specify `nil` or an empty array for this parameter, this method returns without performing the specified selector.

**Discussion**

You can use this method to deliver messages to other threads in your application. The message in this case is a method of the current object that you want to execute on the target thread.

This method queues the message on the run loop of the target thread using the run loop modes specified in the *array* parameter. As part of its normal run loop processing, the target thread dequeues the message (assuming it is running in one of the specified modes) and invokes the desired method.

You cannot cancel messages queued using this method. If you want the option of canceling a message on the current thread, you must use either the `performSelector:withObject:afterDelay:` (page 47) or `performSelector:withObject:afterDelay:inModes:` (page 48) method instead.

This method retains the receiver and the *arg* parameter until after the selector is performed.

**Availability**

Available in Mac OS X v10.5 and later.

**See Also**

- `performSelector:onThread:withObject:waitUntilDone:` (page 45)
- `performSelectorInBackground:withObject:` (page 49)

**Declared In**

`NSThread.h`

## performSelector:withObject:afterDelay:

Invokes a method of the receiver on the current thread using the default mode after a delay.

```
- (void)performSelector:(SEL)aSelector withObject:(id)anArgument
    afterDelay:(NSTimeInterval)delay
```

**Parameters**

*aSelector*

A selector that identifies the method to invoke. The method should not have a significant return value and should take a single argument of type `id`, or no arguments.

See "Selectors" (page 7) for a description of the `SEL` type.

*anArgument*

The argument to pass to the method when it is invoked. Pass `nil` if the method does not take an argument.

*delay*

The minimum time before which the message is sent. Specifying a delay of 0 does not necessarily cause the selector to be performed immediately. The selector is still queued on the thread's run loop and performed as soon as possible.

**Discussion**

This method sets up a timer to perform the *aSelector* message on the current thread's run loop. The timer is configured to run in the default mode (`NSDefaultRunLoopMode`). When the timer fires, the thread attempts to dequeue the message from the run loop and perform the selector. It succeeds if the run loop is running and in the default mode; otherwise, the timer waits until the run loop is in the default mode.

If you want the message to be dequeued when the run loop is in a mode other than the default mode, use the `performSelector:withObject:afterDelay:inModes:` (page 48) method instead. To ensure that the selector is performed on the main thread, use the `performSelectorOnMainThread:withObject:waitUntilDone:` (page 50) or `performSelectorOnMainThread:withObject:waitUntilDone:modes:` (page 51) method instead. To cancel a queued message, use the cancelPreviousPerformRequestsWithTarget: (page 15) or cancelPreviousPerformRequestsWithTarget:selector:object: (page 16) method.

This method retains the receiver and the *anArgument* parameter until after the selector is performed.

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**

+ `cancelPreviousPerformRequestsWithTarget:selector:object:` (page 16)
- `performSelectorOnMainThread:withObject:waitUntilDone:` (page 50)
- `performSelectorOnMainThread:withObject:waitUntilDone:modes:` (page 51)
- `performSelector:onThread:withObject:waitUntilDone:modes:` (page 46)

**Related Sample Code**

IdentitySample

**Declared In**

`NSRunLoop.h`

## performSelector:withObject:afterDelay:inModes:

Invokes a method of the receiver on the current thread using the specified modes after a delay.

```
- (void)performSelector:(SEL)aSelector withObject:(id)anArgument
    afterDelay:(NSTimeInterval)delay inModes:(NSArray *)modes
```

**Parameters**

*aSelector*

A selector that identifies the method to invoke. The method should not have a significant return value and should take a single argument of type `id`, or no arguments.

See "Selectors" (page 7) for a description of the `SEL` type.

*anArgument*

The argument to pass to the method when it is invoked. Pass `nil` if the method does not take an argument.

*delay*

The minimum time before which the message is sent. Specifying a delay of 0 does not necessarily cause the selector to be performed immediately. The selector is still queued on the thread's run loop and performed as soon as possible.

*modes*

An array of strings that identify the modes to associate with the timer that performs the selector. This array must contain at least one string. If you specify `nil` or an empty array for this parameter, this method returns without performing the specified selector.

**Discussion**

This method sets up a timer to perform the *aSelector* message on the current thread's run loop. The timer is configured to run in the modes specified by the *modes* parameter. When the timer fires, the thread attempts to dequeue the message from the run loop and perform the selector. It succeeds if the run loop is running and in one of the specified modes; otherwise, the timer waits until the run loop is in one of those modes.

If you want the message to be dequeued when the run loop is in a mode other than the default mode, use the `performSelector:withObject:afterDelay:inModes:` (page 48) method instead. To ensure that the selector is performed on the main thread, use the `performSelectorOnMainThread:withObject:waitUntilDone:` (page 50) or `performSelectorOnMainThread:withObject:waitUntilDone:modes:` (page 51) method instead. To cancel a queued message, use the cancelPreviousPerformRequestsWithTarget: (page 15) or cancelPreviousPerformRequestsWithTarget:selector:object: (page 16) method.

This method retains the receiver and the *anArgument* parameter until after the selector is performed.

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**

- `performSelector:withObject:afterDelay:` (page 47)
- `performSelectorOnMainThread:withObject:waitUntilDone:` (page 50)
- `performSelectorOnMainThread:withObject:waitUntilDone:modes:` (page 51)
- `performSelector:onThread:withObject:waitUntilDone:modes:` (page 46)

`addTimer:forMode:` (`NSRunLoop`)

`invalidate` (`NSTimer`)

**Declared In**

`NSRunLoop.h`

## performSelectorInBackground:withObject:

Invokes a method of the receiver on a new background thread.

```
- (void)performSelectorInBackground:(SEL)aSelector withObject:(id)arg
```

**Parameters**

*aSelector*

A selector that identifies the method to invoke. The method should not have a significant return value and should take a single argument of type `id`, or no arguments.

See "Selectors" (page 7) for a description of the `SEL` type.

*arg*

The argument to pass to the method when it is invoked. Pass `nil` if the method does not take an argument.

**Discussion**
This method creates a new thread in your application, putting your application into multithreaded mode if it was not already. The method represented by *aSelector* must set up the thread environment just as you would for any other new thread in your program. For more information about how to configure and run threads, see *Threading Programming Guide*.

This method retains the receiver and the *arg* parameter until after the selector is performed.

**Availability**
Available in Mac OS X v10.5 and later.

**See Also**
- `performSelector:onThread:withObject:waitUntilDone:modes:` (page 46)

**Declared In**
`NSThread.h`

## performSelectorOnMainThread:withObject:waitUntilDone:

Invokes a method of the receiver on the main thread using the default mode.

```
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg
    waitUntilDone:(BOOL)wait
```

**Parameters**

*aSelector*

A selector that identifies the method to invoke. The method should not have a significant return value and should take a single argument of type `id`, or no arguments.

See "Selectors" (page 7) for a description of the `SEL` type.

*arg*

The argument to pass to the method when it is invoked. Pass `nil` if the method does not take an argument.

*wait*

A Boolean that specifies whether the current thread blocks until after the specified selector is performed on the receiver on the main thread. Specify `YES` to block this thread; otherwise, specify `NO` to have this method return immediately.

If the current thread is also the main thread, and you specify `YES` for this parameter, the message is delivered and processed immediately.

**Discussion**
You can use this method to deliver messages to the main thread of your application. The main thread encompasses the application's main run loop, and is where the `NSApplication` object receives events. The message in this case is a method of the current object that you want to execute on the thread.

This method queues the message on the run loop of the main thread using the default run loop modes—that is, the modes associated with the `NSRunLoopCommonModes` constant. As part of its normal run loop processing, the main thread dequeues the message (assuming it is running in one of the default run loop modes) and invokes the desired method.

You cannot cancel messages queued using this method. If you want the option of canceling a message on the current thread, you must use either the `performSelector:withObject:afterDelay:` (page 47) or `performSelector:withObject:afterDelay:inModes:` (page 48) method.

This method retains the receiver and the *arg* parameter until after the selector is performed.

**Availability**
Available in Mac OS X v10.2 and later.

**See Also**
- `performSelector:withObject:afterDelay:` (page 47)
- `performSelector:withObject:afterDelay:inModes:` (page 48)
- `performSelectorOnMainThread:withObject:waitUntilDone:modes:` (page 51)
- `performSelector:onThread:withObject:waitUntilDone:modes:` (page 46)

**Related Sample Code**
AudioDeviceNotify
CocoaDVDPlayer
ExtractMovieAudioToAIFF
HelpHook
JSheets

**Declared In**
`NSThread.h`

## performSelectorOnMainThread:withObject:waitUntilDone:modes:

Invokes a method of the receiver on the main thread using the specified modes.

```
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg
    waitUntilDone:(BOOL)wait modes:(NSArray *)array
```

**Parameters**

*aSelector*

A selector that identifies the method to invoke. The method should not have a significant return value and should take a single argument of type `id`, or no arguments.

See "Selectors" (page 7) for a description of the `SEL` type.

*arg*

The argument to pass to the method when it is invoked. Pass `nil` if the method does not take an argument.

*wait*

A Boolean that specifies whether the current thread blocks until after the specified selector is performed on the receiver on the main thread. Specify `YES` to block this thread; otherwise, specify `NO` to have this method return immediately.

If the current thread is also the main thread, and you pass `YES`, the message is performed immediately, otherwise the perform is queued to run the next time through the run loop.

*array*

An array of strings that identifies the modes in which it is permissible to perform the specified selector. This array must contain at least one string. If you specify `nil` or an empty array for this parameter, this method returns without performing the specified selector.

**Discussion**
You can use this method to deliver messages to the main thread of your application. The main thread encompasses the application's main run loop, and is where the `NSApplication` object receives events. The message in this case is a method of the current object that you want to execute on the thread.

This method queues the message on the run loop of the main thread using the run loop modes specified in the *array* parameter. As part of its normal run loop processing, the main thread dequeues the message (assuming it is running in one of the specified modes) and invokes the desired method.

You cannot cancel messages queued using this method. If you want the option of canceling a message on the current thread, you must use either the `performSelector:withObject:afterDelay:` (page 47) or `performSelector:withObject:afterDelay:inModes:` (page 48) method.

This method retains the receiver and the *arg* parameter until after the selector is performed.

**Availability**
Available in Mac OS X v10.2 and later.

**See Also**
- `performSelector:withObject:afterDelay:` (page 47)
- `performSelector:withObject:afterDelay:inModes:` (page 48)
- `performSelectorOnMainThread:withObject:waitUntilDone:` (page 50)
- `performSelector:onThread:withObject:waitUntilDone:modes:` (page 46)

**Declared In**
`NSThread.h`

## replacementObjectForArchiver:

Overridden by subclasses to substitute another object for itself during archiving.

```
- (id)replacementObjectForArchiver:(NSArchiver *)anArchiver
```

**Parameters**

*anArchiver*
The archiver creating an archive.

**Return Value**
The object to substitute for the receiver during archiving.

**Discussion**
This method is invoked by `NSArchiver`. `NSObject`'s implementation returns the object returned by `replacementObjectForCoder:` (page 53).

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
- `classForArchiver` (page 32)

**Declared In**
`NSArchiver.h`

## replacementObjectForCoder:

Overridden by subclasses to substitute another object for itself during encoding.

```
- (id)replacementObjectForCoder:(NSCoder *)aCoder
```

**Parameters**

*aCoder*

The coder encoding the receiver.

**Return Value**

The object encode instead of the receiver (if different).

**Discussion**

An object might encode itself into an archive, but encode a proxy for itself if it's being encoded for distribution. This method is invoked by `NSCoder`. `NSObject`'s implementation returns `self`.

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**

`- classForCoder` (page 32)

`- awakeAfterUsingCoder:` (page 30)

**Declared In**

`NSObject.h`

## replacementObjectForKeyedArchiver:

Overridden by subclasses to substitute another object for itself during keyed archiving.

```
- (id)replacementObjectForKeyedArchiver:(NSKeyedArchiver *)archiver
```

**Parameters**

*archiver*

A keyed archiver creating an archive.

**Return Value**

The object encode instead of the receiver (if different).

**Discussion**

This method is called only if no replacement mapping for the object has been set up in the encoder (for example, due to a previous call of `replacementObjectForKeyedArchiver:` to that object).

**Availability**

Available in Mac OS X v10.2 and later.

**See Also**

`- classForKeyedArchiver` (page 32)

**Declared In**

`NSKeyedArchiver.h`

## replacementObjectForPortCoder:

Overridden by subclasses to substitute another object or a copy for itself during distribution encoding.

```
- (id)replacementObjectForPortCoder:(NSPortCoder *)aCoder
```

**Parameters**

*aCoder*

The port coder encoding the receiver.

**Return Value**

The object encode instead of the receiver (if different).

**Discussion**

This method is invoked by `NSPortCoder`. `NSObject`'s implementation returns an `NSDistantObject` object for the object returned by `replacementObjectForCoder:` (page 53), enabling all objects to be distributed by proxy as the default. However, if `replacementObjectForCoder:` (page 53) returns `nil`, `NSObject`'s implementation will also return `nil`.

Subclasses that want to be passed by copy instead of by reference must override this method and return `self`. The following example shows how to support object replacement both by copy and by reference:

```
- (id)replacementObjectForPortCoder:(NSPortCoder *)encoder {
    if ([encoder isByref])
        return [NSDistantObject proxyWithLocal:self
connection:[encoder connection]];
    else
        return self;
}
```

**Availability**

Available in Mac OS X v10.0 and later.

**See Also**

`- classForPortCoder` (page 33)

**Declared In**

`NSPortCoder.h`

## scriptingProperties

Returns an `NSString`-keyed dictionary of the receiver's scriptable properties.

```
- (NSDictionary *)scriptingProperties
```

**Return Value**

An `NSString`-keyed dictionary of the receiver's scriptable properties, including all of those that are declared as Attributes and ToOneRelationships in the `.scriptSuite` property list entries for the class and its scripting superclasses, with the exception of ones keyed by "scriptingProperties." Each key in the dictionary must be identical to the key for an Attribute or ToOneRelationship. The values of the dictionary must be Objective-C objects that are convertible to `NSAppleEventDescriptor` objects.

**Availability**

Available in Mac OS X v10.2 and later.

**See Also**
- `setScriptingProperties:` (page 55)

**Declared In**
`NSObjectScripting.h`

## scriptingValueForSpecifier:

Given an object specifier, returns the specified object or objects in the receiving container.

```
- (id)scriptingValueForSpecifier:(NSScriptObjectSpecifier *)objectSpecifier;
```

**Parameters**

*objectSpecifier*

An object specifier to be evaluated.

**Return Value**
The specified object or objects in the receiving container.

This method might successfully return an object, an array of objects, or `nil`, depending on the kind of object specifier. Because `nil` is a valid return value, failure is signaled by invoking the object specifier's `setEvaluationError:` method before returning.

**Discussion**
You can override this method to customize the evaluation of object specifiers without requiring that the scripting container make up indexes for contained objects that don't naturally have indexes (as can be the case if you implement `indicesOfObjectsByEvaluatingObjectSpecifier:` instead).

Your override of this method doesn't need to also invoke any of the `NSScriptCommand` error signaling methods, though it can, to record very specific information. The `NSUnknownKeySpecifierError` and `NSInvalidIndexSpecifierError` numbers are special, in that Cocoa may continue evaluating an outer specifier if they're encountered, for the convenience of scripters.

**Availability**
Available in Mac OS X v10.5 and later.

**Declared In**
`NSObjectScripting.h`

## setScriptingProperties:

Given an `NSString`-keyed dictionary, sets one or more scriptable properties of the receiver.

```
- (void)setScriptingProperties:(NSDictionary *)properties
```

**Parameters**

*properties*

A dictionary containing one or more scriptable properties of the receiver. The valid keys for the dictionary include the keys for non-ReadOnly Attributes and ToOneRelationships in the `.scriptSuite` property list entries for the object's class and its scripting superclasses, and no others. The values of the dictionary are Objective-C objects.

**Discussion**
Invokers of this method must have already done any necessary validation to ensure that the properties dictionary includes nothing but entries for declared, settable, Attributes and ToOneRelationships. Implementations of this method are not expected to check the validity of keys in the passed-in dictionary, but must be able to accept dictionaries that do not contain entries for every scriptable property. Implementations of this method must perform type checking on the dictionary values.

**Availability**
Available in Mac OS X v10.2 and later.

**See Also**
- `scriptingProperties` (page 54)

**Declared In**
`NSObjectScripting.h`

## toManyRelationshipKeys

Returns array containing the keys for the to-many relationship properties of the receiver.

```
- (NSArray *)toManyRelationshipKeys
```

**Return Value**
An array containing the keys for the to-many relationship properties of the receiver (if any).

**Discussion**
`NSObject`'s implementation simply invokes `[[self classDescription] toManyRelationshipKeys]`. To make use of the default implementation, you must therefore implement and register a suitable class description—see `NSClassDescription`.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
- `attributeKeys` (page 29)
- `classDescription` (page 31)
- `inverseForRelationshipKey:` (page 42)
- `toOneRelationshipKeys` (page 56)

**Declared In**
`NSClassDescription.h`

## toOneRelationshipKeys

Returns the keys for the to-one relationship properties of the receiver, if any.

```
- (NSArray *)toOneRelationshipKeys
```

**Return Value**
An array containing the keys for the to-one relationship properties of the receiver.

**Discussion**
`NSObject`'s implementation of `toOneRelationshipKeys` simply invokes `[[self classDescription] toOneRelationshipKeys]`. To make use of the default implementation, you must therefore implement and register a suitable class description—see `NSClassDescription`.

**Availability**
Available in Mac OS X v10.0 and later.

**See Also**
- `attributeKeys` (page 29)
- `classDescription` (page 31)
- `toManyRelationshipKeys` (page 56)
- `inverseForRelationshipKey:` (page 42)

**Declared In**
`NSClassDescription.h`

# Deprecated NSObject Methods

A method identified as deprecated has been superseded and may become unsupported in the future.

## Deprecated in Mac OS X v10.5

### poseAsClass:

Causes the receiving class to pose as a specified superclass. (Deprecated in Mac OS X v10.5.)

```
+ (void)poseAsClass:(Class)aClass
```

**Parameters**

*aClass*

A superclass of the receiver.

**Discussion**

The receiver takes the place of *aClass* in the inheritance hierarchy; all messages sent to *aClass* will actually be delivered to the receiver. The receiver must be defined as a subclass of *aClass*. It can't declare any new instance variables of its own, but it can define new methods and override methods defined in *aClass*. The `poseAsClass:` message should be sent before any messages are sent to *aClass* and before any instances of *aClass* are created.

This facility allows you to add methods to an existing class by defining them in a subclass and having the subclass substitute for the existing class. The new method definitions will be inherited by all subclasses of the superclass. Care should be taken to ensure that the inherited methods do not generate errors.

A subclass that poses as its superclass still inherits from the superclass. Therefore, none of the functionality of the superclass is lost in the substitution. Posing doesn't alter the definition of either class.

Posing is useful as a debugging tool, but category definitions are a less complicated and more efficient way of augmenting existing classes. Posing admits only two possibilities that are absent from categories:

- A method defined by a posing class can override any method defined by its superclass. Methods defined in categories can replace methods defined in the class proper, but they cannot reliably replace methods defined in other categories. If two categories define the same method, one of the definitions will prevail, but there's no guarantee which one.
- A method defined by a posing class can, through a message to `super`, incorporate the superclass method it overrides. A method defined in a category can replace a method defined elsewhere by the class, but it can't incorporate the method it replaces.

**Special Considerations**

Posing is deprecated in Mac OS X v10.5. The `poseAsClass:` method is not available in 64-bit applications on Mac OS X v10.5.

**Availability**
Available in Mac OS X v10.0.
Deprecated in Mac OS X v10.5.

**Declared In**
`NSObject.h`

# Document Revision History

This table describes the changes to *NSObject Class Reference*.

| Date | Notes |
| --- | --- |
| 2009-02-04 | Added note about managing scarce resources in dealloc. |
| 2008-10-15 | Added special consideration for +initialize method. |
| 2008-06-09 | Updated sample code in +resolveInstanceMethod: to properly call super. |
| 2008-03-11 | Updated the descriptions of the methods copyScriptingValue:forKey:withProperties: and newScriptingObjectOfClass:forValueForKey: withContentsValue:properties:. |
| 2008-02-08 | Updated the description of the load method. |
| 2007-12-11 | Updated descriptions for the performSelector methods. |
| 2007-07-07 | Included new API for Mac OS X v10.5. |
| | The following new methods have been added for use with Cocoa scripting: `copyScriptingValue:forKey:withProperties:` (page 34), `newScriptingObjectOfClass:forValueForKey:withContentsValue:properties:` (page 44), and `scriptingValueForSpecifier:` (page 55). |
| 2006-06-28 | Augmented the description of dealloc and version. |
| 2006-05-23 | First publication of this content as a separate document. |

Document Revision History

# Index


## A

alloc class method 14
allocWithZone: class method 14
attributeKeys instance method 29
awakeAfterUsingCoder: instance method 30

## C

cancelPreviousPerformRequestsWithTarget: class method 15
cancelPreviousPerformRequestsWithTarget:selector:object: class method 16
class class method 17
classCode instance method 31
classDescription instance method 31
classFallbacksForKeyedArchiver class method 17
classForArchiver instance method 32
classForCoder instance method 32
classForKeyedArchiver instance method 32
classForKeyedUnarchiver class method 18
classForPortCoder instance method 33
className instance method 33
conformsToProtocol: class method 18
copy instance method 34
copyScriptingValue:forKey:withProperties: instance method 34
copyWithZone: class method 19

## D

dealloc instance method 35
description class method 19
doesNotRecognizeSelector: instance method 36

## F

finalize instance method 37
forwardInvocation: instance method 38

## I

init instance method 40
initialize class method 20
instanceMethodForSelector: class method 21
instanceMethodSignatureForSelector: class method 22
instancesRespondToSelector: class method 23
inverseForRelationshipKey: instance method 42
isSubclassOfClass: class method 23

## L

load class method 23

## M

methodForSelector: instance method 42
methodSignatureForSelector: instance method 43
mutableCopy instance method 44
mutableCopyWithZone: class method 24

## N

new class method 25
newScriptingObjectOfClass:forValueForKey:withContentsValue:properties: instance method 44

## P

performSelector:onThread:withObject:waitUntilDone: instance method 45
performSelector:onThread:withObject:waitUntilDone:modes: instance method 46
performSelector:withObject:afterDelay: instance method 47
performSelector:withObject:afterDelay:inModes: instance method 48
performSelectorInBackground:withObject: instance method 49
performSelectorOnMainThread:withObject:waitUntilDone: instance method 50
performSelectorOnMainThread:withObject:waitUntilDone:modes: instance method 51
poseAsClass: class method 59

## R

replacementObjectForArchiver: instance method 52
replacementObjectForCoder: instance method 53
replacementObjectForKeyedArchiver: instance method 53
replacementObjectForPortCoder: instance method 54
resolveClassMethod: class method 26
resolveInstanceMethod: class method 26

## S

scriptingProperties instance method 54
scriptingValueForSpecifier: instance method 55
setScriptingProperties: instance method 55
setVersion: class method 27
superclass class method 28

## T

toManyRelationshipKeys instance method 56
toOneRelationshipKeys instance method 56

## V

version class method 28