NSObject

Adopted By: NSObject

Declared In: Foundation/NSObject.h

Protocol Description

The NSObject protocol groups methods that are fundamental to all Objective-C objects. If an object conforms to this protocol, it can be considered a first-class object in NEXTSTEP. Such an object can be asked about its:

- Class, and the place of its class in the inheritance hierarchy
- Conformance to protocols
- Ability to respond to a particular message

In addition, objects that conform to this protocol—with its retain, release, and autorelease methods—can also integrate with the object-management and deallocation scheme defined in the Foundation Kit. (See the introduction to the Foundation Kit for more information.) Thus, an object that conforms to the NSObject protocol can be managed by container objects like those defined by NSArray and NSDictionary.

NEXTSTEP's root class, NSObject, adopts this protocol, so virtually all objects in NEXTSTEP have the features described by this protocol.

Method Types

Identifying classes	– class – superclass
Identifying and comparing objects	– isEqual: – hash – self
Determining allocation zones	– zone
Managing reference counts	– retain – release – autorelease – retainCount
Testing class functionality	– respondsToSelector:
Testing inheritance relationships	– isKindOfClass: – isMemberOfClass:

Testing protocol conformance	– conformsToProtocol:
Describing objects	– description
Sending messages	– perform: – perform:withObject: – perform:withObject:withObject:
Identifying proxies	– isProxy

Instance Methods

autorelease

– (id)**autorelease**

Adds the receiver to the current autorelease pool and returns **self**. You add an object to an autorelease pool so that it will receive a **release** message—and thus might be deallocated—when the pool is destroyed. For more information on the autorelease mechanism, see the NSAutoreleasePool class specification.

See also: – **retain**, – **retainCount**

class

– (Class)**class**

Returns the class object for the receiver's class.

See also: + **class** (NSObject class)

conformsToProtocol:

– (BOOL)**conformsToProtocol:**(Protocol *)*aProtocol*

Returns YES if the receiving class conforms to *aProtocol*, NO otherwise. This method works identically to the **conformsToProtocol:** class method declared in NSObject. It's provided as a convenience so that you don't need to get the class object to find out whether an instance can respond to a given set of messages.

See also: + **conformsToProtocol:** (NSObject class)

description

– (NSString *)**description**

Returns an NSString object that describes the contents of the receiver. The debugger's **print-object** command indirectly invokes this method to produce a textual description of an object.

hash

– (unsigned)**hash**

Returns an integer that can be used as a table address in a hash table structure. If two objects are equal (as determined by the **isEqual:** method), they must have the same hash value. This last point is particularly important if you define **hash** in a subclass and intend to put instances of that subclass into a collection.

isEqual:

– (BOOL)**isEqual:**(id)*anObject*

Returns YES if the receiver and *anObject* are equal, NO otherwise. This method defines what it means for an instance to be equal. For example, a container object might define two containers as equal if their corresponding objects all respond YES to an **isEqual:** request. See the NSData, NSDictionary, NSArray, and NSString class specifications for examples of the use of this method.

isKindOfClass:

– (BOOL)**isKindOfClass:**(Class)*aClass*

Returns YES if the receiver is an instance of *aClass* or an instance of any class that inherits from *aClass*, NO otherwise. For example, in this code, **isKindOfClass:** would return YES because, in the Application Kit, the NSMenu class inherits from NSWindow:

```
id aMenu = [[NSMenu alloc] init];
if ( [aMenu isKindOfClass:[NSWindow class]] )
    . . .
```

When the receiver is a class object, this method returns YES if *aClass* is NSObject, NO otherwise.

See also: – **isMemberOfClass:**

isMemberOfClass:

– (BOOL)**isMemberOfClass:**(Class)*aClass*

Returns YES if the receiver is an instance of *aClass*, NO otherwise. For example, in this code, **isMemberOfClass:** would return NO:

```
id aMenu = [[NSMenu alloc] init];
if ([aMenu isMemberOfClass:[NSWindow class]])
    . . .
```

When the receiver is a class object, this method returns NO. Class objects are not “members of” any class.

See also: – **isKindOfClass:**

isProxy

– (BOOL)**isProxy**

Returns NO if the receiver really descends from NSObject, YES otherwise. This method is necessary because sending isKindOfClass: or isMemberOfClass: to an NSProxy object will test the object that the proxy stands-in for, not itself. Use this method to test if the receiver is a proxy (or a member of some other root class).

perform:

– (id)**perform:**(SEL)*aSelector*

Sends an *aSelector* message to the receiver and returns the result of the message. If *aSelector* is NULL, an NSInvalidArgumentException is raised.

perform: is equivalent to sending an *aSelector* message directly to the receiver. For example, all three of the following messages do the same thing:

```
id myClone = [anObject copy];
id myClone = [anObject perform:@selector(copy)];
id myClone = [anObject perform:sel_getUid("copy")];
```

However, the **perform:** method allows you to send messages that aren't determined until run time. A variable selector can be passed as the argument:

```
SEL myMethod = findTheAppropriateSelectorForTheCurrentSituation();
[anObject perform:myMethod];
```

aSelector should identify a method that takes no arguments. For methods that return anything other than an object, use NSInvocation.

See also: – **perform:withObject:**, – **perform:withObject:withObject:**

perform:withObject:

– (id)**perform:**(SEL)*aSelector* **withObject:**(id)*anObject*

Sends an *aSelector* message to the receiver with *anObject* as the argument. If *aSelector* is NULL, an NSInvalidArgumentException is raised.

This method is the same as **perform:** except that you can supply an argument for *aSelector*. *aSelector* should identify a method that takes a single argument of type **id**. For methods with other argument types and return values, use NSInvocation.

See also: – **perform:withObject:withObject:**, – **methodForSelector:** (NSObject class)

perform:withObject:withObject:

– (id)**perform:**(SEL)*aSelector*
withObject:(id)*anObject*
withObject:(id)*anotherObject*

Sends the receiver an *aSelector* message with *anObject* and *anotherObject* as arguments. If *aSelector* is NULL, an NSInvalidArgumentException is raised. This method is the same as **perform:** except that you can supply two arguments for *aSelector. aSelector* should identify a method that can take two arguments of type **id**. For methods with other argument types and return values use NSInvocation.

See also: – **perform:withObject:**, – **methodForSelector:** (NSObject class)

release

– (oneway void)**release**

Decrements the receivers's reference count, and sends it a dealloc message when its reference count reaches 0.

You send release messages only to objects that you "own." By definition, you own objects that you create using one of the alloc... or copy... methods. These methods return objects with an implicit reference count of one. You also own (or perhaps share ownership in) an object that you send a retain message to because retain increments the object's reference count. Each retain message you send an object should be balanced eventually with a release or autorelease message, so that the object can be deallocated. For more information on the automatic deallocation mechanism, see the introduction to the Foundation Kit.

You would only implement this method to define your own reference-counting scheme. Such implementations should not invoke the inherited method; that is, they should not include a release message to super.

See also: – **retainCount**

respondsToSelector:

– (BOOL)**respondsToSelector:**(SEL)*aSelector*

Returns YES if the receiver implements or inherits a method that can respond to *aSelector* messages, NO otherwise. The application is responsible for determining whether a NO response should be considered an error.

Note that if the receiver is able to forward *aSelector* messages to another object, it will be able to respond to the message, albeit indirectly, even though this method returns NO.

See also: – **forwardInvocation:** (NSObject class), + **instancesRespondToSelector:**(NSObject class)

retain

– (id)**retain**

Increments the receiver's reference count. You send an object a retain message when you want to prevent it from being deallocated without your express permission.

An object is deallocated automatically when its reference count reaches 0. retain messages increment the reference count, and release messages decrement it. For more information on this mechanism, see the introduction to the Foundation Kit.

As a convenience, retain returns self because it is often used in nested expressions:

```
NSString *headerDir = [[NSString
    stringWithCString:"/LocalLibrary/Headers"] retain];
```

You would only implement this method if you were defining your own reference-counting scheme. Such implementations must return self and should not invoke the inherited method by sending a retain message to super.

See also: – **autorelease,** – **release**, – **retainCount**

retainCount

– (unsigned)**retainCount**

Returns the receiver's reference count for debugging purposes. You rarely send a retainCount message; however, you might implement this method in a class to implement your own reference-counting scheme. For objects that never get released (that is, their release method does nothing), this method should return UINT_MAX, as defined in <limits.h>.

See also: – **autorelease**, – **retain**

self

– (id)**self**

Returns the receiver.

See also: – **class**

superclass

– (Class)**superclass**

Returns the class object for the receiver's superclass.

See also: + **superclass** (NSObject class)

zone

– (NSZone *)**zone**

Returns a pointer to the zone from which the receiver was allocated. Objects created without specifying a zone are allocated from the default zone.

See also: + **allocWithZone:** (NSObject class)