NSObject

Inherits From: none *(NSObject is a root class)*

Conforms To: NSObject

Declared In: Foundation/NSObject.h

Class at a Glance

Purpose

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

Principal Attributes

- **isa** pointer

Creation

+ alloc
+ allocWithZone:
– init (designated initializer)

Class Description

NSObject is the root class of most Objective-C class hierarchies; it has no superclass. From NSObject, other classes inherit a basic interface to the run-time system for the Objective-C language, and its instances obtain their ability to behave as objects.

NSObject adopts the NSObject protocol. The NSObject protocol allows for multiple root objects. For example, NSProxy is another root class—it does not inherit from NSObject but adopts the NSObject protocol so that it shares a common interface with other Objective-C objects. Some of the methods discussed below are declared by NSObject protocol, not this class.

Among other things, the NSObject class provides inheriting classes with a framework for creating, initializing, deallocating, copying, comparing, archiving and distributing objects, for performing methods selected at run-time, for querying an object about its methods and its position in the inheritance hierarchy,

and for forwarding messages to other objects. For example, to ask an object what class it belongs to, you'd send it a **class** message. To find out whether it implements a particular method, you'd send it a **respondsToSelector:** message.

The NSObject class is mostly an abstract class; programs use instances of classes that inherit from NSObject, but rarely instances of NSObject itself.

Initializing an Object to Its Class

Every object that inherits directly or indirectly from NSObject is connected to the run-time system through its **isa** instance variable. **isa** identifies the object's class; it references a structure that's compiled from the class definition. Through **isa**, an object can find whatever information it needs at run-time—such as its place in the inheritance hierarchy, the size and structure of its instance variables, and the location of the method implementations it can perform in response to messages.

The installation of the class structure—the initialization of **isa**—is one of the responsibilities of class methods that create (allocate memory for) new instances: **alloc**, **allocWithZone:** and **new**. In other words, a small part of instance initialization is taken care of by these creation methods; it's not left to the methods, such as **init**, that initialize individual objects with their particular characteristics.

Instance and Class Methods

The run-time system treats methods defined in the root class in a special way:

> Instance methods defined in a root class can be performed both by instances and by class objects.

Therefore, all class objects have access to the instance methods defined in the root class. Any class object can perform any root instance method, provided it doesn't have a class method with the same name.

For example, a class object could be sent messages to perform NSObject's **respondsToSelector:** and **performSelector:withObject:** instance methods:

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

if ( [MyClass respondsToSelector:method] )
    [MyClass performSelector:method withObject:self];
```

Note that the only instance methods available to a class object are those defined in its root class. In the example above, if MyClass had reimplemented either **respondsToSelector:** or **performSelector:withObject:**, those new versions would be available only to instances. The class object for MyClass could perform only the versions defined in the NSObject class. (Of course, if MyClass had implemented **respondsToSelector:** or **performSelector:withObject:** as class methods rather than instance methods, the class would perform those new versions.)

Interface Conventions

The NSObject class defines a number of methods that subclasses are expected to override. Often, NSObject's default implementation simply returns **self**. Putting these "empty" methods in the NSObject class serves two purposes:

- It means that every object can readily respond to certain standard messages, such as **init**, even if the response is to do nothing. It's not necessary to check (using **respondsToSelector:**) before sending the message.
- It establishes conventions that, when followed by all classes, make object interactions more reliable. These conventions are explained in full under the method descriptions.

Instance Variables

Class **isa**;

isa	A pointer to the instance's class structure.

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

Method Types

Initializing the class	+ initialize + load
Creating, copying, and deallocating objects	+ new + alloc + allocWithZone: – init – copy + copyWithZone: – mutableCopy + mutableCopyWithZone: – dealloc
Identifying classes	+ class + superclass
Comparing objects	– isEqual: – hash
Testing class functionality	+ instancesRespondToSelector:
Testing protocol conformance	+ conformsToProtocol:
Obtaining method information	– methodForSelector: + instanceMethodForSelector: + instanceMethodSignatureForSelector: – methodSignatureForSelector:
Describing objects	+ description – description
Posing	+ poseAsClass:
Sending messages	– performSelector:withObject:afterDelay: – performSelector:withObject:afterDelay:inModes: + cancelPreviousPerformRequestsWithTarget: selector:object:
Forwarding messages	– forwardInvocation:
Error handling	– doesNotRecognizeSelector:

Archiving	– awakeAfterUsingCoder: – classForArchiver – classForCoder – classForPortCoder – replacementObjectForArchiver: – replacementObjectForCoder: – replacementObjectForPortCoder: + setVersion: + version

Class Methods

alloc

+ (id)**alloc**

Returns a new instance of the receiving class. 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 will be allocated from the default zone—use **allocWithZone:** to specify a particular zone.

An **init...** method should 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.

Note that it's your responsibility to release objects (with either **release** or **autorelease**) returned by **alloc...** methods.

See also: – **init**

allocWithZone:

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

Returns a new instance of the receiving class where memory for the new instance is allocated from *zone*. 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 NULL*, the new instance will be allocated from the default zone (as returned by **NSDefaultMallocZone()**).

An **init...** method should 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 often 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];
```

Note that it's your responsibility to release objects (with either **release** or **autorelease**) returned by **alloc...** methods.

See also: + **alloc**, – **init**

cancelPreviousPerformRequestsWithTarget:selector:object:

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

Cancels perform requests previously registered with the **performSelector:withObject:afterDelay:** instance method. All perform requests having the same target *aTarget*, and argument *anArgument*, (determined using **isEqual:**), and the same selector *aSelector*, will be canceled. This method removes perform requests only in the current run loop, not all run loops.

class

+ (Class)**class**

Returns the class object.

Only refer to a class by 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]];
```

See also: – **class** (NSObject protocol)

conformsToProtocol:

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

Returns YES if the receiving class conforms to *aProtocol*, NO otherwise.

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 that 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)];
```

See also: – **conformsToProtocol:**

copyWithZone:

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

Returns self. This method exists so that 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. You should not override this method.

See also: –

description

+ (NSString *)**description**

Returns an NSString that represents the contents of the receiving class. 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.

See also: – **description**

initialize

+ (void)**initialize**

Initializes the class before it's used (before it receives its first message). The run-time system sends an **initialize** message to each class just before the class, or any class that inherits from it, is sent its first message from within the program. Each class receives the **initialize** message just once from the run-time system. Superclasses will receive this message before subclasses.

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

```
[NSApplication new]
```

the run-time system invokes 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** message.

If your program later begins to use the NSText class,

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

the run-time system invokes these additional **initialize** messages:

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

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

You can implement your own versions of **initialize** to provide class-specific initialization as needed.

Normally the run-time system sends a class just one **initialize** message. However, if for some reason an application or the run-time system generates additional **initialize** messages, it is a good idea to prevent code from being invoked more than once:

```
+ (void)initialize
{
    static BOOL tooLate = NO;
    if ( !tooLate ) {
        /* put initialization code here */
        tooLate = YES;
    }
}
```

See also: – **init**, – **class** (NSObject protocol)

instanceMethodForSelector:

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

Locates and returns the address of the implementation for the *aSelector* instance method. 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 methods, send the **methodForSelector:** instance method to the class instead.

instanceMethodForSelector:, and the function pointer it returns, are subject to the same constraints as those described for **methodForSelector:**. See **methodForSelector:** for description of the IMP return value.

instanceMethodSignatureForSelector:

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

Returns an NSMethodSignature object that contains a description of the *aSelector* class method, or **nil** if the *aSelector* class method can't be found.

See also: – **methodSignatureForSelector:**

instancesRespondToSelector:

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

Returns YES if instances of the class are capable of responding to *aSelector* messages, NO otherwise. To ask the class whether it, rather than its instances, can respond to a particular message, send the **respondsToSelector:** NSObject protocol instance method to the class instead.

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

See also: – **forwardInvocation:**

load

+(void)**load**

This method is invoked whenever a class or category is added to the Objective-C runtime; implement this method to perform class-specific behavior upon loading. It 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. As an example, when Interface Builder loads a palette, the **load** method is sent to each class and category in the palette.

load is usually invoked before **initialize**. It is usually the very first method sent to the class, although this isn't guaranteed. The order in which classes are loaded is also not guaranteed, to the point that superclasses aren't even guaranteed to be loaded before all of their subclasses. Because you can't rely on other classes being loaded at the point when your class is sent a load message, you should be extremely careful when messaging other classes from within your **load** method.

Due to the amount of uncertainty about the environment at the point that **load** is invoked, you should avoid using **load** whenever possible. All class-specific initialization should be done in the class's **initialize** method.

Note that although **load** essentially replaces NEXTSTEP's **finishLoading:** method, the circumstances surrounding their invocation is slightly different. Consult your NEXTSTEP Developer documentation if you are porting code that uses **finishLoading:**.

See also: + **load** (NSProxy)

mutableCopyWithZone:

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

Returns self. This method exists so that 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. You should not override this method.

See also: –

new

+ (id)**new**

Allocates a new instance of the receiving class, sends it an **init** message, and returns the initialized object.

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

Unlike **alloc**, **new** is sometimes reimplemented in subclasses to have it invoke a class-specific initialization method. If the **init...** method includes arguments, they’re typically reflected in a **new...** method as well. For example:

```
+ newArg:(int)tag arg:(struct info *)data
{
    return [[self alloc] initArg:tag arg:data];
}
```

However, there’s little point in implementing a **new...** method if it’s simply a shorthand for **alloc** 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:

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

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

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

poseAsClass:

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

Causes the receiving class to "pose as" its *aClass* superclass. 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 this doesn't 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.

setVersion:

+ (void)**setVersion:**(int)*aVersion*

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

See also: + **version**

superclass

+ (Class)**superclass**

Returns the class object for the receiver's superclass.

See also: + **class**, – **superclass** (NSObject protocol)

version

+ (int)**version**

Returns the version number assigned to the class. If no version has been set, the default is 0.

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

Caution should be taken when obtaining the version from within 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.

See also: + **setVersion:, versionForClassName:** (NSCoder)

nstance Methods

awakeAfterUsingCoder:

– (id)**awakeAfterUsingCoder:**(NSCoder *)*aDecoder*

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

See also: – **classForCoder**, – **replacementObjectForCoder:**, – **initWithCoder:**(NSCoding protocol)

classForArchiver

– (Class)**classForArchiver**

Overridden by subclasses to substitute a class other than its own during archiving. For example, the private subclasses of a class cluster substitute the name of their public superclass when being archived. This method allows specialized behavior for archiving—override **classForCoder** to add general coding behavior. This method is invoked by NSArchiver. NSObject's implementation returns the object returned by **classForCoder**.

See also: – **replacementObjectForArchiver:**

classForCoder

– (Class)**classForCoder**

Overridden by subclasses to substitute a class other than its own during coding. For example, the private subclasses of a class cluster substitute the name of their public superclass when being archived. This method is invoked by NSCoder. NSObject's implementation returns the receiver's class.

See also: – **awakeAfterUsingCoder:**, – **replacementObjectForCoder:**

classForPortCoder

– (Class)**classForPortCoder**

Overridden by subclasses to substitute a class other than its own for distribution encoding. This method allows specialized behavior for distributed objects—override **classForCoder** to add general coding behavior. This method is invoked by NSPortCoder. NSObject's implementation returns the class returned by **classForCoder**.

See also: – **replacementObjectForPortCoder:**

copy

– (id)**copy**

Convenience method for classes that adopt the NSCopying protocol. This method returns the object returned by the NSCopying protocol method **copyWithZone:** where the zone is NULL. An exception is raised if there is no implementation for **copyWithZone:**.

dealloc

– (void)**dealloc**

Deallocates the memory occupied by the receiver. Subsequent messages to the object will generate an error indicating that a message was sent to a deallocated object (provided that 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. See the introduction to the Foundation Kit for more details on the use of these methods.

Subclasses must implement their own versions of **dealloc** to allow the deallocation of any additional memory consumed by the object—such as dynamically allocated storage for data, or object instance variables that are 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];
}
```

See also: – **autorelease** (NSObject protocol), – **release** (NSObject protocol)

description

@protocol NSObject
– (NSString *)**description**

Returns a NSString that represents the contents of the receiver. The debugger's **print-object** command indirectly invokes this method to produce a textual description of an object. NSObject's implementation of this method simply prints the name of the receiver's class and the hexadecimal value of its **id**.

See also: + **description**

doesNotRecognizeSelector:

– (void)**doesNotRecognizeSelector:**(SEL)*aSelector*

Handles *aSelector* messages that the receiver doesn't recognize. The run-time system invokes this method whenever an object receives an *aSelector* message that it can't respond to or forward. This method, in turn, raises an NSInvalidArgumentException, and generates an error message.

doesNotRecognizeSelector: messages are generally sent only by the run-time 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** or **init** method by reimplementing it to include a **doesNotRecognizeSelector:** message as follows:

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

The **_cmd** variable identifies 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.

See also: – **forwardInvocation:**

forwardInvocation:

– (void)**forwardInvocation:**(NSInvocation *)*anInvocation*

Overridden by subclasses to forward messages to other objects. When an object is sent a message for which it has no corresponding method, the run-time system gives the receiver an opportunity to delegate the message to another receiver. It does this by creating an NSInvocation object representing the message and sending the receiver a **forwardInvocation:** message containing this NSInvocation 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.

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

- To locate an object that can respond to the message encoded in *anInvocation*. This need not be the same object for all messages.
- To send the message to that object using *anInvocation*. *anInvocation* will hold the result, and the run-time 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
{
    if ([friend respondsToSelector:[invocation selector]])
        [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 message that's forwarded is returned to the original sender. All types of return values can be delivered to the sender: **id**s, 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.

To respond to methods that your object does not itself recognize, you must override **methodSignatureFromSelector:** in addition to **forwardInvocation:**. The mechanism for forwarding messages uses information obtained from **methodSignatureFromSelector:** 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.

NSObject's implementation of **forwardInvocation:** simply invokes the **doesNotRecognizeSelector:** method; it doesn't forward any messages. Thus, if you choose not to implement **forwardInvocation:**, unrecognized messages will raise an exception.

hash

@protocol NSObject
– (unsigned)**hash**

Returns an integer that can be used as a table address in a hash table structure. NSObject's implementation returns a value based on the object's **id**. If two objects are equal (as determined by the **isEqual:** method), they must return 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.

init

– (id)**init**

Implemented by subclasses to initialize a new object (the receiver) immediately after memory for it has been allocated. An **init** message is generally coupled with an **alloc** or **allocWithZone:** 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 version of the **init** method defined in the NSObject class does no initialization; it simply returns **self**.

Subclass versions of this method should return the new object (**self**) after it has been successfully initialized. 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** or **allocWithZone:**, in subsequent code.

Every class must guarantee that the **init** method either returns a fully functional instance of the class or raises an exception. Typically this means overriding the 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**:

```
- 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 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:

```
- init;
- initArg:(int)tag;
- initArg:(int)tag arg:(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:

```
- init
{
    return [self initArg:-1];
}

- initArg:(int)tag
{
    return [self initArg:tag arg:NULL];
}

- initArg:(int)tag arg:(struct info *)data
{
    [super init. . .];
    /* class-specific initialization goes here */
}
```

In this example, the **initArg:arg:** 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 perform 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:

```
- initArg:(int)tag arg:(struct info *)data arg:anObject
{
    [super initArg:tag arg:data];
    /* class-specific initialization goes here */
}
```

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:

```
- initArg:(int)tag arg:(struct info *)data
{
    return [self initArg:tag arg:data arg:nil];
}
```

This 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.

isEqual:

@protocol NSObject
– (BOOL)**isEqual:**(id)*anObject*

Returns YES if the receiver and *anObject* are equal, NO otherwise. NSObject's implementation compares the **id** of *anObject* and the receiver to determine equality. Subclasses can override this method to redefine what it means for objects to be equal. For example, a container object might define two containers as equal if they contain the same contents. See the NSData, NSDictionary, NSArray, and NSString class specifications for examples of the use of this method. Note that equality as defined by this method is not necessarily reflexive. For example, A is equal to B, does not imply B is equal to A, especially if B is a subclass of A.

methodForSelector:

– (IMP)**methodForSelector:**(SEL)*aSelector*

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

aSelector must be a valid, non-NULL selector. If in doubt, use the **respondsToSelector:** method to check before passing the selector to **methodForSelector:**.

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 that **methodForSelector:** returns. It's sufficient for methods that return an object and take object arguments. However, if the *aSelector* method takes different argument types or returns anything but an **id**, its function counterpart will be inadequately prototyped. Lacking a prototype, the compiler will promote **float**s to **double**s and **char**s to **int**s, 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:** 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:** 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:** 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. This can result in errors.

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.

See also: + **instanceMethodForSelector:**

methodSignatureForSelector:

– (NSMethodSignature *)**methodSignatureForSelector:**(SEL)*aSelector*

Returns an NSMethodSignature object that contains a description of the *aSelector* method, or **nil** if the *aSelector* method can't be found. When the receiver is an instance, *aSelector* should be an instance method; when the receiver is a class, it should be a class method.

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.

See also: + **instanceMethodSignatureForSelector:**, – **forwardInvocation:**

mutableCopy

– (id)**mutableCopy**

Convenience method for classes that adopt the NSMutableCopying protocol. This method just calls the NSMutableCopying protocol method **mutableCopyWithZone:** with the zone as NULL. An exception is raised if there is no implementation for **mutableCopyWithZone:**.

performSelector:withObject:afterDelay:

– (void)**performSelector:**(SEL)*aSelector*
withObject:(id)*anArgument*
afterDelay:(NSTimeInterval)*delay*

Sends an *aSelector* message to the receiver sometime after *delay*. This method returns before the *aSelector* message is sent. The *aSelector* method should not have a significant return value and should take a single argument of type **id**; *anArgument* will be the argument passed in the message. Note that **self** and *anArgument* are retained until after the message is sent.

See also: – **cancelPreviousPerformRequestsWithTarget:selector:object:**

replacementObjectForArchiver:

– (id)**replacementObjectForArchiver:**(NSArchiver *)*anArchiver*

Overridden by subclasses to substitute another object for itself during archiving. This method is invoked by NSArchiver. NSObject's implementation returns the object returned by **replacementObjectForCoder:**.

See also: – **classForArchiver**

replacementObjectForCoder:

– (id)**replacementObjectForCoder:**(NSCoder *)*aCoder*

Overridden by subclasses to substitute another object for itself during encoding. For example, 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**.

See also: – **classForCoder**, – **awakeAfterUsingCoder:**

replacementObjectForPortCoder:

– (id)**replacementObjectForPortCoder:**(NSPortCoder *)*aCoder*

Overridden by subclasses to substitute another object or a copy for itself during distribution encoding. This method is invoked by NSPortCoder. NSObject's implementation returns a NSDistantObject for the object returned by **replacementObjectForCoder:**, enabling all objects to be distributed by proxy as the default. However, if **replacementObjectForCoder:** 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 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;
}
```

See also: – **classForPortCoder**