# Object

**Inherits From:** none *(Object is the root class)*

**Declared In:** objc/Object.h

## Class Description

Object is the root class of all ordinary Objective C inheritance hierarchies; it's the one class that has no superclass. From Object, other classes inherit a basic interface to the run-time system for the Objective C language. It's through Object that instances of all classes obtain their ability to behave as objects.

Among other things, the Object class provides inheriting classes with a framework for creating, initializing, freeing, copying, comparing, and archiving 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 **respondsTo:** message.

The Object class is an abstract class; programs use instances of classes that inherit from Object, but never of Object itself.

### Initializing an Object to Its Class

Every object is connected to the run-time system through its **isa** instance variable, inherited from the Object class. **isa** identifies the object's class; it points to 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.

Because all objects directly or indirectly inherit from the Object class, they all have this variable. The defining characteristic of an "object" is that its first instance variable is an **isa** pointer to a class structure.

The installation of the class structure—the initialization of **isa**—is one of the responsibilities of the **alloc**, **allocFromZone:**, and **new** methods, the same methods that

create (allocate memory for) new instances of a class. In other words, class initialization is part of the process of creating an object; it's not left to the methods, such as **init**, that initialize individual objects with their particular characteristics.

## Instance and Class Methods

Every object requires an interface to the run-time system, whether it's an instance object or a class object. For example, it should be possible to ask either an instance or a class about its position in the inheritance hierarchy or whether it can respond to a particular message.

So that this won't mean implementing every Object method twice, once as an instance method and again as a class method, the run-time system treats methods defined in the root class in a special way:

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

A class object has access to class methods—those defined in the class and those inherited from the classes above it in the inheritance hierarchy—but generally not to instance methods. However, the run-time system gives all class objects 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 Object's **respondsTo:** and **perform:with:** instance methods:

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

if ( [MyClass respondsTo:method] )
    [MyClass perform:method with:self];
```

When a class object receives a message, the run-time system looks first at the receiver's repertoire of class methods. If it fails to find a class method that can respond to the message, it looks at the set of instance methods defined in the root class. If the root class has an instance method that can respond (as Object does for **respondsTo:** and **perform:with:**), the run-time system uses that implementation and the message succeeds.

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

### Interface Conventions

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

- It means that every object can readily respond to certain standard messages, such as **awake** or **init**, even if the response is to do nothing  It's not necessary to check (using **respondsTo:**) 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.

Sometimes a method is merely declared in the Object class; it has no implementation, not even the empty one of returning **self**.  These "unimplemented" methods serve the same purpose—defining an interface convention—as Object's "empty" methods.  When implemented, they enable objects to respond to system-generated messages.

## Instance Variables

Class **isa**;

| | |
|---|---|
| isa | A pointer to the instance's class structure. |

## Method Types

| | |
|---|---|
| Initializing the class | + initialize |
| Creating, copying, and freeing instances | + alloc<br>+ allocFromZone:<br>+ new<br>– copy<br>– copyFromZone:<br>– zone<br>– free<br>+ free |
| Initializing a new instance | – init |

| | |
|---|---|
| Identifying classes | + name<br>+ class<br>– class<br>+ superclass<br>– superclass |
| Identifying and comparing instances | – isEqual:<br>– hash<br>– self<br>– name<br>– printForDebugger: |
| Testing inheritance relationships | – isKindOf:<br>– isKindOfClassNamed:<br>– isMemberOf:<br>– isMemberOfClassNamed: |
| Testing class functionality | – respondsTo:<br>+ instancesRespondTo: |
| Testing for protocol conformance | + conformsTo:<br>– conformsTo: |
| Sending messages determined at run time | – perform:<br>– perform:with:<br>– perform:with:with: |
| Forwarding messages | – forward::<br>– performv:: |
| Obtaining method information | – methodFor:<br>+ instanceMethodFor:<br>– descriptionForMethod:<br>+ descriptionForInstanceMethod: |
| Posing | + poseAs: |
| Enforcing intentions | – notImplemented:<br>– subclassResponsibility: |
| Error handling | – doesNotRecognize:<br>– error: |
| Dynamic loading | + finishLoading:<br>+ startUnloading |

| | |
|---|---|
| Archiving | – read:<br>– write:<br>– startArchiving:<br>– awake<br>– finishUnarchiving<br>+ setVersion:<br>+ version |

## Class Methods

### alloc

+ **alloc**

Returns a new instance of the receiving class. The **isa** instance variable of the new object is initialized to a data structure that describes the class; memory for all other instance variables is set to 0. A version of the **init** method should be used to complete the initialization process. For example:

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

Other classes shouldn't override **alloc** to add code that initializes the new instance. Instead, class-specific versions of the **init** method should be implemented for that purpose. Versions of the **new** method can also be implemented to combine allocation and initialization.

**Note:** The **alloc** method doesn't invoke **allocFromZone:**. The two methods work independently.

**See also:** + **allocFromZone:**, – **init**, + **new**

### allocFromZone:

+ **allocFromZone:**(NXZone *)*zone*

Returns a new instance of the receiving class. Memory for the new object is allocated from *zone*.

The **isa** instance variable of the new object is initialized to a data structure that describes the class; memory for its other instance variables is set to 0. A version of the **init** method should be used to complete the initialization process. For example:

```
id newObject = [[TheClass allocFromZone:someZone] init];
```

The **allocFromZone:** method shouldn't be overridden to include any initialization code. Instead, class-specific versions of the **init** method 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 can be used for this purpose; it returns the zone where the receiver is located. For example:

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

**See also:** + **alloc**, – **zone**, – **init**

## class

+ **class**

Returns **self**. Since this is a class method, it returns the class object.

When a class is the receiver of a message, it can be referred to by name. 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 **isKindOf:** method:

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

**See also:** – **name**, – **class**

## conformsTo:

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

Returns YES if the receiving class conforms to *aProtocol*, and NO if it doesn't.

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. Here, for example, MyClass adopts the imaginary AffiliationRequests and Normalization protocols:

```
@interface MyClass : Object <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>
```

When a class adopts a protocol, it must implement all the methods the protocol declares. If it 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. In the example above, MyClass must implement the methods in the AffiliationRequests and

Normalization protocols and, in addition, either inherit from a class that adopts the Joining protocol or implement the Joining methods itself.

When these conventions are followed and all the methods in adopted and incorporated protocols are in fact implemented, the **conformsTo:** test for a set of methods becomes roughly equivalent to the **respondsTo:** test for a single method.

However, **conformsTo:** judges conformance solely on the basis of the formal declarations in source code, as illustrated above. It doesn't check to see whether the methods declared in the protocol are actually implemented. It's the programmer's responsibility to see that they are.

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

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

The Protocol class is documented in Chapter 15, "Run-Time System."

**See also:** – **conformsTo:**

## descriptionForInstanceMethod:

+ (struct objc_method_description *)
    **descriptionForInstanceMethod:**(SEL)*aSelector*

Returns a pointer to a structure that describes the *aSelector* instance method, or NULL if the *aSelector* method can't be found. To ask the class for a description of a class method, or an instance for the description of an instance method, use the **descriptionForMethod:** instance method.

**See also:** – **descriptionForMethod:**

## finishLoading:

+ **finishLoading:**(struct mach_header *)*header*

Implemented by subclasses to integrate the class, or a category of the class, into a running program. A **finishLoading:** message is sent immediately after the class or category has been dynamically loaded into memory, but only if the newly loaded class or category implements a method that can respond. *header* is a pointer to the structure that describes the modules that were just loaded.

Once a dynamically loaded class is used, it will also receive an **initialize** message. However, because the **finishLoading:** message is sent immediately after the class is loaded,

it always precedes the **initialize** message, which is sent only when the class receives its first message from within the program.

A **finishLoading:** method is specific to the class or category where it's defined; it's not inherited by subclasses or shared with the rest of the class. Thus a class that has four categories can define a total of five **finishLoading:** methods, one in each category and one in the main class definition. The method that's performed is the one defined in the class or category just loaded.

There's no default **finishLoading:** method. The Object class declares a prototype for this method, but doesn't implement it.

**See also:** + **startUnloading**

## free

+ **free**

Returns **nil**. This method is implemented to prevent class objects, which are "owned" by the run-time system, from being accidentally freed. To free an instance, use the instance method **free**.

**See also:** – **free**

## initialize

+ **initialize**

Initializes the class before it's used (before it receives its first message). The run-time system generates 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 object receives the **initialize** message just once. Superclasses receive it before subclasses do.

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

```
[Application new]
```

the run-time system will generate these three **initialize** messages,

```
[Object initialize];
[Responder initialize];
[Application initialize];
```

since Application is a subclass of Responder and Responder is a subclass of Object. All the **initialize** messages precede the **new** message and are sent in the order of inheritance, as shown.

If your program later begins to use the Text class,

```
[Text instancesRespondTo:someSelector]
```

the run-time system will generate these additional **initialize** messages,

```
[View initialize];
[Text initialize];
```

since the Text class inherits from Object, Responder, and View. The **instancesRespondTo:** message is sent only after all these classes are initialized. Note that the **initialize** messages to Object and Responder aren't repeated; each class is initialized only once.

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

Because **initialize** methods are inherited, it's possible for the same method to be invoked many times, once for the class that defines it and once for each inheriting class. To prevent code from being repeated each time the method is invoked, it can be bracketed as shown in the example below:

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

Since the run-time system sends a class just one **initialize** message, the test shown in the example above should prevent code from being invoked more than once. However, if for some reason an application also generates **initialize** messages, a more explicit test may be needed:

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

**See also:** – **init**, – **class**

### instanceMethodFor:

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

Locates and returns the address of the implementation of the *aSelector* instance method. An error is generated if instances of the receiver can't respond to *aSelector* messages.

This method is used to ask the class object for the implementation of an instance method. To ask the class for the implementation of a class method, use the instance method **methodFor:** instead of this one.

**instanceMethodFor:**, and the function pointer it returns, are subject to the same constraints as those described for **methodFor:**.

**See also:** – **methodFor:**

### instancesRespondTo:

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

Returns YES if instances of the class are capable of responding to *aSelector* messages, and NO if they're not. To ask the class whether it, rather than its instances, can respond to a particular message, use the **respondsTo:** instance method instead of **instancesRespondTo:**.

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:** – **respondsTo:**, – **forward::**

### name

+ (const char *)**name**

Returns a null-terminated string containing the name of the class. This information is often used in error messages or debugging statements.

**See also:** – **name**, + **class**

### new

+ **new**

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

As defined in the Object class, **new** is essentially 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 the **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...**, like the one 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 one only if necessary. For example:

```
+ newArg:(int)tag arg:(struct info *)data
{
    id 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 **allocFromZone:** methods should never be augmented to include initialization code.

**See also:** – **init**, + **alloc**, + **allocFromZone:**

## poseAs:

+ **poseAs:***aClassObject*

Causes the receiving class to "pose as" its superclass, the *aClassObject* class. The receiver takes the place of *aClassObject* in the inheritance hierarchy; all messages sent to *aClassObject* will actually be delivered to the receiver. The receiver must be defined as a subclass of *aClassObject*. It can't declare any new instance variables of its own, but it can define new methods and override methods defined in the superclass. The **poseAs:** message should be sent before any messages are sent to *aClassObject* and before any instances of *aClassObject* 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 for 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.

If successful, this method returns **self**. If not, it generates an error message and aborts.

## setVersion:

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

Sets the class version number to *aVersion*, and returns **self**. 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**

## startUnloading

+ **startUnloading**

Implemented by subclasses to prepare for the class, or a category of the class, being unloaded from a running program. A **startUnloading** message is sent immediately before the class or category is unloaded, but only if the class or category about to be unloaded implements a method that can respond.

A **startUnloading** method is specific to the class or category where it's defined; it isn't inherited by subclasses or shared with the rest of the class. Thus a class that has four categories can define a total of five **startUnloading** methods, one in each category and one

in the main class definition. The method that's performed is the one defined in the class or category that will be unloaded.

There's no default **startUnloading** method. The object class declares a prototype for this method but doesn't implement it.

**See also:** + **finishLoading:**

### superclass

+ **superclass**

Returns the class object for the receiver's superclass.

**See also:** + **class**, – **superclass**

### version

+ (int)**version**

Returns the version number assigned to the class. If no version has been set, this will be 0.

**See also:** + **setVersion:**

## Instance Methods

### awake

– **awake**

Implemented by subclasses to reinitialize the receiving object after it has been unarchived (by **read:**). An **awake** message is automatically sent to every object after it has been unarchived and after all the objects it refers to are in a usable state.

The default version of the method defined here merely returns **self**.

A class can implement an **awake** method to provide for more initialization than can be done in the **read:** method. Each implementation of **awake** should limit the work it does to the scope of the class definition, and incorporate the initialization of classes farther up the inheritance hierarchy through a message to **super**. For example:

```
- awake
{
    [super awake];
    /* class-specific initialization goes here */
    return self;
}
```

All implementations of **awake** should return **self**.

**Note:** Not all objects loaded from a nib file (created by Interface Builder) are unarchived; some are newly instantiated. Those that are unarchived receive an **awake** message, but those that are instantiated do not. See the Interface Builder documentation in *NEXTSTEP Development Tools* for more information.

**See also:** – **read:**, – **finishUnarchiving**, – **awakeFromNib** (NXNibNotification protocol in the Application Kit), – **loadNibFile:owner:** (Application class in the Application Kit)

## class

– **class**

Returns the class object for the receiver's class.

**See also:** + **class**

## conformsTo:

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

Returns YES if the class of the receiver conforms to *aProtocol*, and NO if it doesn't. This method invokes the **conformsTo:** class method to do its work. 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:** + **conformsTo:**

## copy

– **copy**

Returns a new instance that's an exact copy of the receiver. This method creates only one new object. If the receiver has instance variables that point to other objects, the instance variables in the copy will point to the same objects. The values of the instance variables are copied, but the objects they point to are not.

This method does its work by invoking the **copyFromZone:** method and specifying that the copy should be allocated from the same memory zone as the receiver. If a subclass implements its own **copyFromZone:** method, this **copy** method will use it to copy instances of the subclass. Therefore, a class can support copying from both methods just by implementing a class-specific version of **copyFromZone:**.

**See also:** – **copyFromZone:**

## copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new instance that's an exact copy of the receiver. Memory for the new instance is allocated from *zone*.

This method creates only one new object. If the receiver has instance variables that point to other objects, the instance variables in the copy will point to the same objects. The values of the instance variables are copied, but the objects they point to are not.

Subclasses should implement their own versions of **copyFromZone:**, not **copy**, to define class-specific copying.

**See also:** – **copy**, – **zone**

## descriptionForMethod:

– (struct objc_method_description *)**descriptionForMethod:**(SEL)*aSelector*

Returns a pointer to a structure that describes the *aSelector* method, or NULL 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.

The **objc_method_description** structure is declared in **objc/Protocol.h**, and is mostly used in the implementation of protocols. It includes two fields—the selector for the method (which will be the same as *aSelector*) and a character string encoding the method's return and argument types:

```
struct objc_method_description {
    SEL name;
    char *types;
};
```

Type information is encoded according to the conventions of the **@encode()** directive, but the string also includes information about total argument size and individual argument offsets. For example, if **descriptionForMethod:** were asked for a description of itself, it would return this string in the **types** field:

```
^{objc_method_description=:*}12@8:12:16
```

This records the fact that **descriptionForMethod:** returns a pointer (‘^’) to a structure (“{...}”) and that it pushes a total of 12 bytes on the stack. The structure is called “objc_method_description” and it consists of a selector (‘:’) and a character pointer (‘*’). The first argument, **self**, is an object (‘@’) at an offset of 8 bytes from the stack pointer, the second argument, **_cmd**, is a selector (‘:’) at an offset of 12 bytes, and the third argument, *aSelector*, is also a selector but at an offset of 16 bytes. The first two arguments—**self** for the message receiver and **_cmd** for the method selector—are passed to every method implementation but are hidden by the Objective C language.

The type codes used for methods declared in a class or category are:

| Meaning | Code |
|---|---|
| **id** | ‘@’ |
| Class | ‘#’ |
| SEL | ‘:’ |
| **void** | ‘v’ |
| **char** | ‘c’ |
| **unsigned char** | ‘C’ |
| **short** | ‘s’ |
| **unsigned short** | ‘S’ |
| **int** | ‘i’ |
| **unsigned int** | ‘I’ |
| **long** | ‘l’ |
| **unsigned long** | ‘L’ |
| **float** | ‘f’ |
| **double** | ‘d’ |
| **char *** | ‘*’ |
| any other pointer | ‘^’ |
| an undefined type | ‘?’ |
| a bitfield | ‘b’ |
| begin an array | ‘[’ |
| end an array | ‘]’ |
| begin a union | ‘(’ |
| end a union | ‘)’ |
| begin a structure | ‘{’ |
| end a structure | ‘}’ |

The same codes are used for methods declared in a protocol, but with these additions for type modifiers:

| | |
|---|---|
| **const** | ‘r’ |
| **in** | ‘n’ |
| **inout** | ‘N’ |
| **out** | ‘o’ |
| **bycopy** | ‘O’ |
| **oneway** | ‘V’ |

**See also:** + **descriptionForInstanceMethod:**, – **descriptionForClassMethod:** (Protocol class in the Run-Time System), – **descriptionForInstanceMethod** (Protocol class in the Run-Time System)

## doesNotRecognize:

– **doesNotRecognize:**(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, invokes the **error:** method to generate an error message and abort the current process.

**doesNotRecognize:** messages should be sent only by the run-time system. Although they’re sometimes used in program code to prevent a method from being inherited, it’s better to use the **error:** method directly. For example, an Object subclass might renounce the **copy** method by reimplementing it to include an **error:** message as follows:

```
- copy
{
    [self error:"  %s objects should not be sent '%s' messages\n",
            [[self class] name], sel_getName(_cmd)];
}
```

This code prevents instances of the subclass from recognizing or forwarding **copy** messages—although the **respondsTo:** method will still report that the receiver has access to a **copy** method.

(The **_cmd** variable identifies the current selector; in the example above, it identifies the selector for the **copy** method. The **sel_getName()** function returns the method name corresponding to a selector code; in the example, it returns the name “copy”.)

**See also:** – **error:**, – **subclassResponsibility:**, + **name**

## error:

– **error:**(const char *)*aString*, ...

Generates a formatted error message, in the manner of **printf()**, from *aString* followed by a variable number of arguments. For example:

```
[self error:"index %d exceeds limit %d\n", index, limit];
```

The message specified by *aString* is preceded by this standard prefix (where *class* is the name of the receiver's class):

```
"error: class "
```

This method doesn't return. It calls the run-time **_error** function, which first generates the error message and then calls **abort()** to create a core file and terminate the process.

**See also:** – **subclassResponsibility:**, – **notImplemented:**, – **doesNotRecognize:**

## finishUnarchiving

– **finishUnarchiving**

Implemented by subclasses to replace an unarchived object with a new object if necessary. A **finishUnarchiving** message is sent to every object after it has been unarchived (using **read:**) and initialized (by **awake**), but only if a method has been implemented that can respond to the message.

The **finishUnarchiving** message gives the application an opportunity to test an unarchived and initialized object to see whether it's usable, and, if not, to replace it with another object that is. This method should return **nil** if the unarchived instance (**self**) is OK; otherwise, it should free the receiver and return another object to take its place.

There's no default implementation of the **finishUnarchiving** method. The Object class declares this method, but doesn't define it.

**See also:** – **read:**, – **awake**, – **startArchiving:**

## forward::

– **forward:**(SEL)*aSelector* **:**(marg_list)*argFrame*

Implemented by subclasses to forward messages to other objects. When an object is sent an *aSelector* message, and the run-time system can't find an implementation of the method for the receiving object, it sends the object a **forward::** message to give it an opportunity to delegate the message to another receiver. (If the delegated receiver can't respond to the message either, it too will be given a chance to forward it.)

The **forward::** 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.

A **forward::** message is generated only if the *aSelector* method isn't implemented by the receiving object's class or by any of the classes it inherits from.

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

- To locate an object that can respond to the *aSelector* message. This need not be the same object for all messages.
- To send the message to that object, using the **performv::** method.

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 **forward::** method could be as simple as this:

```
- forward:(SEL)aSelector :(marg_list)argFrame
{
    if ( [friend respondsTo:aSelector] )
        return [friend performv:aSelector :argFrame];
    [self doesNotRecognize:aSelector];
}
```

*argFrame* is a pointer to the arguments included in the original *aSelector* message. It's passed directly to **performv::** without change. (However, *argFrame* does not correctly capture variable arguments. Messages that include a variable argument list—for example, messages to perform Object's **error**: method—cannot be forwarded.)

The *aSelector* message will return the value returned by **forward::**. (Note in the example that **forward::** returns unchanged the value returned by **performv::**.) Since **forward::**

returns a pointer, specifically an **id**, the *aSelector* method must also be one that returns a pointer (or **void**). Methods that return other types cannot be reliably forwarded.

Implementations of the **forward::** method can do more than just forward messages. **forward::** 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 **forward::** method might also involve several other objects in the response to a given message, rather than forward it to just one.

The default version of **forward::** implemented in the Object class simply invokes the **doesNotRecognize:** method; it doesn't forward messages. Thus, if you choose not to implement **forward::**, unrecognized messages will generate an error and cause the task to abort.

**Note:** If it's necessary for a **forward::** method to reason about the arguments passed in *argFrame*, it can get information about what kinds of arguments they are by calling the **method_getNumberOfArguments()**, **method_getSizeOfArguments()**, and **method_getArgumentInfo()** run-time functions. It can then examine and alter argument values with the **marg_getValue()**, **marg_getRef()**, and **marg_setValue()** macros. These functions and macros are documented in Chapter 15, "Run-Time System.".

**See also:** – **performv::**, – **doesNotRecognize:**

## free

– **free**

Frees the memory occupied by the receiver and returns **nil**. Subsequent messages to the object will generate an error indicating that a message was sent to a freed object (provided that the freed memory hasn't been reused yet).

Subclasses must implement their own versions of **free** to deallocate any additional memory consumed by the object—such as dynamically allocated storage for data, or other objects that are tightly coupled to the freed object and are of no use without it. After performing the class-specific deallocation, the subclass method should incorporate superclass versions of **free** through a message to **super**:

```
- free {
    [companion free];
    free(privateMemory);
    vm_deallocate(task_self(), sharedMemory, memorySize);
    return [super free];
}
```

If, under special circumstances, a subclass version of **free** refuses to free the receiver, it should return **self** instead of **nil**. Object's default version of this method always frees the receiver and always returns **nil**. It calls **object_dispose()** to accomplish the deallocation.

## hash

– (unsigned int)**hash**

Returns an unsigned integer that's derived from the **id** of the receiver. The integer is guaranteed to always be the same for the same **id**.

**See also: – isEqual:**

## init

– **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 **allocFromZone:** message in the same line of code:

```
id 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 Object 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 free the object and return **nil**. In some cases, an **init** method might free 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 **allocFromZone:**, in subsequent code.

Every class must guarantee that the **init** method returns a fully functional instance of the class. 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
{
    [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 add arguments to the **init** method 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 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 Object class. Subclasses that do their own initialization should override it, as described above.

**See also:** + **new**, + **alloc**, + **allocFromZone:**

## isEqual:

– (BOOL)**isEqual:***anObject*

Returns YES if the receiver is the same as *anObject*, and NO if it isn't. This is determined by comparing the **id** of the receiver to the **id** of *anObject*.

Subclasses may need to override this method to provide a different test of equivalence. For example, in some contexts, two objects might be said to be the same if they're both the same kind of object and they both contain the same data:

```
- (BOOL)isEqual:anObject
{
    if ( anObject == self )
        return YES;
    if ( [anObject isKindOf:[self class]] ) {
        if ( !strcmp(stringData, [anObject stringData]) )
            return YES;
    }
    return NO;
}
```

### isKindOf:

– (BOOL)**isKindOf:***aClassObject*

Returns YES if the receiver is an instance of *aClassObject* or an instance of any class that inherits from *aClassObject*. Otherwise, it returns NO. For example, in this code **isKindOf:** would return YES because, in the Application Kit, the Menu class inherits from Window:

```
id  aMenu = [[Menu alloc] init];
if ( [aMenu isKindOf:[Window class]] )
    . . .
```

When the receiver is a class object, this method returns YES if *aClassObject* is the Object class, and NO otherwise.

**See also:** – **isMemberOf:**

### isKindOfClassNamed:

– (BOOL)**isKindOfClassNamed:**(const char *)*aClassName*

Returns YES if the receiver is an instance of *aClassName* or an instance of any class that inherits from *aClassName*. This method is the same as **isKindOf:**, except it takes the class name, rather than the class **id**, as its argument.

**See also:** – **isMemberOfClassNamed:**

### isMemberOf:

– (BOOL)**isMemberOf:***aClassObject*

Returns YES if the receiver is an instance of *aClassObject*. Otherwise, it returns NO. For example, in this code, **isMemberOf:** would return NO:

```
id  aMenu = [[Menu alloc] init];
if ([aMenu isMemberOf:[Window class]])
    . . .
```

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

**See also:** – **isKindOf:**

### isMemberOfClassNamed:

– (BOOL)**isMemberOfClassNamed:**(const char *)*aClassName*

Returns YES if the receiver is an instance of *aClassName*, and NO if it isn't. This method is the same as **isMemberOf:**, except it takes the class name, rather than the class **id**, as its argument.

**See also:** – **isKindOfClassNamed:**

### methodFor:

– (IMP)**methodFor:**(SEL)*aSelector*

Locates and returns the address of the receiver's implementation of 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, nonNULL selector. If in doubt, use the **respondsTo:** method to check before passing the selector to **methodFor:**.

IMP is defined (in the **objc/objc.h** header file) 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 **methodFor:** 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 called as a function than 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 pointer to a function that returns a BOOL and takes an **id** argument (in addition to the two "hidden" arguments). The value returned by **methodFor:** is then similarly cast to be a pointer to this same function type:

```
BOOL (*test)(id, SEL, id);
test = (BOOL (*)(id, SEL, id))[target methodFor:@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 **methodFor:** returns. The example below defines the **EqualIMP** type for just this purpose:

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

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

Either way, it's important to cast **methodFor:**'s return value to the appropriate function type. It's not sufficient to simply call the function returned by **methodFor:** and cast the result of that call to the desired type. This can result in errors.

Note that turning a method into a function by obtaining the address of its implementation "unhides" the **self** and **_cmd** arguments.

**See also:** + **instanceMethodFor:**

## name

– (const char *)**name**

Implemented by subclasses to return a name associated with the receiver.

By default, the string returned contains the name of the receiver's class. However, this method is commonly overridden to return a more object-specific name. You should therefore not rely on it to return the name of the class. To get the name of the class, use the class **name** method instead:

```
const char *classname = [[self class] name];
```

**See also:** + **name**, + **class**

## notImplemented:

– **notImplemented:**(SEL)*aSelector*

Used in the body of a method definition to indicate that the programmer intended to implement the method, but left it as a stub for the time being. *aSelector* is the selector for the unimplemented method; **notImplemented:** messages are sent to **self**. For example:

```
- methodNeeded
{
    [self notImplemented:_cmd];
}
```

When a **methodNeeded** message is received, **notImplemented:** will invoke the **error:** method to generate an appropriate error message and abort the process. (In this example, **_cmd** refers to the **methodNeeded** selector.)

**See also:** – **subclassResponsibility:**, – **error:**

## perform:

– **perform:**(SEL)*aSelector*

Sends an *aSelector* message to the receiver and returns the result of the message. This 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. If the method returns anything but an object, the return must be cast to the correct type. For example:

```
char *myClass;
myClass = (char *)[anObject perform:@selector(name)];
```

Casting generally works for pointers and for integral types that are the same size as pointers (such as **int** and **enum**). Whether it works for other integral types (such as **char**, **short**, or **long**) is machine dependent. Casting doesn't work if the return is a floating type (**float** or **double**) or a structure or union. This is because the C language doesn't permit a pointer (like **id**) to be cast to these types.

Therefore, **perform:** shouldn't be asked to perform any method that returns a floating type, structure, or union, and should be used very cautiously with methods that return integral types. An alternative is to get the address of the method implementation (using **methodFor:**) and call it as a function. For example:

```
SEL aSelector = @selector(backgroundGray);
float aGray = ( (float (*)(id, SEL))
                [anObject methodFor:aSelector] )(anObject, aSelector);
```

**See also:** – **perform:with:**, – **perform:with:with:**, – **methodFor:**

## perform:with:

– **perform:**(SEL)*aSelector* **with:***anObject*

Sends an *aSelector* message to the receiver with *anObject* as an argument. This method is the same as **perform:**, except that you can supply an argument for the *aSelector* message. *aSelector* should identify a method that takes a single argument of type **id**.

**See also:** – **perform:**, – **perform:with:afterDelay:cancelPrevious:** (Application Kit Object Additions)

## perform:with:with:

– **perform:**(SEL)*aSelector*
**with:***anObject*
**with:***anotherObject*

Sends the receiver an *aSelector* message with *anObject* and *anotherObject* as arguments. This method is the same as **perform:**, except that you can supply two arguments for the *aSelector* message. *aSelector* should identify a method that can take two arguments of type **id**.

**See also:** – **perform:**

## performv::

– **performv:**(SEL)*aSelector* **:**(marg_list)*argFrame*

Sends the receiver an *aSelector* message with the arguments in *argFrame*. **performv::** messages are used within implementations of the **forward::** method. Both arguments, *aSelector* and *argFrame*, are identical to the arguments the run-time system passes to **forward::**. They can be taken directly from that method and passed through without change to **performv::**.

**performv::** should be restricted to implementations of the **forward::** method. Because it doesn't restrict the number of arguments in the *aSelector* message or their type, it may seem like a more flexible way of sending messages than **perform:**, **perform:with:**, or **perform:with:with:**. However, it's not an appropriate substitute for those methods. First, it's more expensive than they are. The run-time system must parse the arguments in *argFrame* based on information stored for *aSelector.* Second, in future releases, **performv::** may not work in contexts other than the **forward::** method.

**See also:** – **forward::**, – **perform:**

## printForDebugger:

– (void)**printForDebugger:**(NXStream *)*stream*

Implemented by subclasses to write a useful description of the receiver to *stream*. Object's default version of this method provides the class name and the hexadecimal address of the receiver, formatted as follows:

<*classname*: 0x*address*>

Debuggers can use this method to ask objects to identify themselves.

## read:

– **read:**(NXTypedStream *)*stream*

Implemented by subclasses to read the receiver's instance variables from the typed stream *stream*. You need to implement a **read:** method for any class you create, if you want its instances (or instance of classes that inherit from it) to be archivable.

The method you implement should unarchive the instance variables defined in the class in a manner that matches they way they were archived by **write:**. In each class, the **read:** method should begin with a message to **super**:

```
- read:(NXTypedStream *)stream
{
    [super read:stream];
    /* class-specific code goes here */
    return self;
}
```

This ensures that all inherited instance variables will also be unarchived.

All implementations of the **read:** method should return **self**. Also, don't reassign the value of **self** within a **read:** method.

After an object has been read, it's sent an **awake** message so that it can reinitialize itself, and may also be sent a **finishUnarchiving** message.

**See also:** – **awake**, – **finishUnarchiving**, – **write:**

## respondsTo:

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

Returns YES if the receiver implements or inherits a method that can respond to *aSelector* messages, and NO if it doesn't. 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:** – **forward::**, + **instancesRespondTo:**

## self

– **self**

Returns the receiver.

**See also:** + **class**

## startArchiving:

– **startArchiving:**(NXTypedStream *)*stream*

Implemented by subclasses to prepare an object for being archived—that is, for being written to the typed stream *stream*. A **startArchiving:** message is sent to an object just before it's archived—but only if it implements a method that can respond. The message

gives the object an opportunity to do anything necessary to get itself, or the stream, ready before a **write:** message begins the archiving process.

There's no default implementation of the **startArchiving:** method. The Object class declares the method, but doesn't define it.

**See also:** – **awake**, – **finishUnarchiving**, – **write:**

## subclassResponsibility:

– **subclassResponsibility:**(SEL)*aSelector*

Used in an abstract class to indicate that its subclasses are expected to implement *aSelector* methods. If a subclass fails to implement the method, it will inherit it from the abstract superclass. That version of the method generates an error when it's invoked. To avoid the error, subclasses must override the superclass method.

For example, if subclasses are expected to implement **doSomething** methods, the superclass would define the method this way:

```
- doSomething
{
    [self subclassResponsibility:_cmd];
}
```

When this version of **doSomething** is invoked, **subclassResponsibility:** will—by in turn invoking Object's **error:** method—abort the process and generate an appropriate error message.

(The **_cmd** variable identifies the current method selector, just as **self** identifies the current receiver. In the example above, it identifies the selector for the **doSomething** method.)

Subclass implementations of the *aSelector* method shouldn't include messages to **super** to incorporate the superclass version. If they do, they'll also generate an error.

**See also:** – **doesNotRecognize:**, – **notImplemented:**, – **error:**

## superclass

– **superclass**

Returns the class object for the receiver's superclass.

**See also:** + **superclass**

## write:

– **write:**(NXTypedStream *)*stream*

Implemented by subclasses to write the receiver's instance variables to the typed stream *stream*. You need to implement a **write:** method for any class you create, if you want to be able to archive its instances (or instances of classes that inherit from it).

The method you implement should archive only the instance variables defined in the class, but should begin with a message to **super** so that all inherited instance variables will also be archived:

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    /* class-specific archiving code goes here */
    return self;
}
```

All implementations of the **write:** method should return **self**.

During the archiving process, **write:** methods may be performed twice, so they shouldn't do anything other than write instance variables to a typed stream.

**See also:** – **read:**, – **startArchiving:**

## zone

– (NXZone *)**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, which is returned by **NXDefaultMallocZone()**.

**See also:** + **allocFromZone:**, + **alloc**, + **copyFromZone:**